
Database Management Systems (DBMS) Lab File

Database Management System (MC-302)

Delhi Technological University

19th May 2020



Anish Sachdeva
DTU/2K16/MC/13

Index

DDL Statements & Commands	1
Create Database	1
Create Table Within the Database with Constraints	2
Alter Table Commands	3
Add	3
Drop	4
Rename a Column	5
DML Statements & Commands	6
Insert Tuples	6
Delete Tuples	10
Update Tuples	11
Synopsis of Project With ER Diagram	12
Program for Following SQL Commands	14
IN / NOT IN	14
Aggregate Functions	15
Group by + Having clause	17
Order By	18
Join Statements & Commands	19
Natural Join	19
Inner Join	19
Outer Join	20
Introduction To PL/SQL	22
Variables in PL/SQL	22
Variable Declaration	22
Initializing Variables In PL/SQL	22
Variable Scope in PL/SQL	23
Assigning SQL Query Results to PL/SQL Variables	24
I/O in PL/SQL	27
Database Pipes	27
Creating the DBMS_PIPE Package	27

Public Pipes	27
Writing and Reading	28
Private Pipes	28
Errors	29
CREATE_PIPE	29
Packages in PL/SQL	30
What is a Package?	30
Reasons to Use Packages	31
Package Specification	32
Exceptions In PL/SQL	34
Exception Types	35
Trapping Exceptions	36
Trapping predefined TimesTen errors	36
Triggers in PL/SQL	39
Benefits of Triggers	39
Creating Triggers	39
Example	40
Triggering a Trigger	42
Transactions in SQL	43
Properties of Transactions	43
Transaction Control	43
Transactional Control Commands	44
Commit	44
Example	44
Rollback	45
Example	45
Savepoint	46
Example	47

DDL Statements & Commands

Create Database

-- Creating address table

```
CREATE TABLE address (  
  id INTEGER PRIMARY KEY AUTOINCREMENT ,  
  zip_code INTEGER NOT NULL ,  
  street VARCHAR(255) ,  
  city VARCHAR(255)  
);
```

-- Creating the Organization Table

```
CREATE TABLE organization (  
  id INTEGER PRIMARY KEY AUTOINCREMENT ,  
  name VARCHAR(255) NOT NULL ,  
  address_id INTEGER ,  
  FOREIGN KEY (address_id) REFERENCES address(id)  
);
```

-- Create the Product table

```
CREATE TABLE product (  
  id INTEGER PRIMARY KEY AUTOINCREMENT ,  
  name varchar(255) NOT NULL ,  
  cost NUMERIC NOT NULL ,  
  organization_id INTEGER ,  
  FOREIGN KEY (organization_id) REFERENCES organization(id)  
);
```

-- Creating Customer Database

```
create table customer (  
  id INTEGER primary key autoincrement ,  
  firstName varchar(255) not null,  
  lastName varchar(255) not null,
```

```
    age INTEGER
);
-- Creating the Orders Table
CREATE TABLE "order" (
    id INTEGER PRIMARY KEY AUTOINCREMENT ,
    customer_id INTEGER,
    product_id INTEGER,
    date DATE ,
    FOREIGN KEY (customer_id) REFERENCES customer(id),
    FOREIGN KEY (product_id) REFERENCES product(id)
);
```

Create Table Within the Database with Constraints

```
-- Creating the Orders Table
CREATE TABLE "order" (
    id INTEGER PRIMARY KEY AUTOINCREMENT ,
    customer_id INTEGER,
    product_id INTEGER,
    date DATE DEFAULT(CURRENT_DATE),
    FOREIGN KEY (customer_id) REFERENCES customer(id),
    FOREIGN KEY (product_id) REFERENCES product(id)
);

-- Creating Customer Database
create table customer (
    id INTEGER primary key autoincrement ,
    firstName varchar(255) not null,
    lastName varchar(255) not null,
    age INTEGER ,
    check (age >= 10)
);
```

Alter Table Commands

Add

-- Adding an email column in customer table

ALTER TABLE customer

ADD COLUMN email VARCHAR(255);

	id	firstName	lastName	age	email
1	1	anish	sachdeva	22	<null>
2	2	piyush	gupta	22	<null>
3	3	akshita	chander	22	<null>
4	4	maddy	iota	22	<null>
5	5	gaurav	garg	22	<null>
6	6	smiti	thappar	22	<null>
7	7	apurva	puri	22	<null>
8	8	aviral	sharma	22	<null>
9	9	harshil	chauhan	22	<null>
10	10	ashish	sharma	22	<null>

-- Adding Date of Birth in Customer table

ALTER TABLE customer

ADD COLUMN date_of_birth DATE;

	id	firstName	lastName	age	date_of_b...	email
1	1	anish	sachdeva	22	<null>	<null>
2	2	piyush	gupta	22	<null>	<null>
3	3	akshita	chander	22	<null>	<null>
4	4	maddy	iota	22	<null>	<null>
5	5	gaurav	garg	22	<null>	<null>
6	6	smiti	thappar	22	<null>	<null>
7	7	apurva	puri	22	<null>	<null>
8	8	aviral	sharma	22	<null>	<null>
9	9	harshil	chauhan	22	<null>	<null>
10	10	ashish	sharma	22	<null>	<null>

Drop

-- Removing email column from customer table

ALTER TABLE customer **DROP COLUMN** email;

	id	firstName	lastName	age	email
1	1	anish	sachdeva	22	<null>
2	2	piyush	gupta	22	<null>
3	3	akshita	chander	22	<null>
4	4	maddy	iota	22	<null>
5	5	gaurav	garg	22	<null>
6	6	smiti	thappar	22	<null>
7	7	apurva	puri	22	<null>
8	8	aviral	sharma	22	<null>
9	9	harshil	chauhan	22	<null>
10	10	ashish	sharma	22	<null>

-- Removing Date Of Birth column in customer table

ALTER TABLE customer **DROP COLUMN** date_of_birth;




	id	firstName	lastName	age	email
1	1	anish	sachdeva	22	<null>
2	2	piyush	gupta	22	<null>
3	3	akshita	chander	22	<null>
4	4	maddy	iota	22	<null>
5	5	gaurav	garg	22	<null>
6	6	smiti	thappar	22	<null>
7	7	apurva	puri	22	<null>
8	8	aviral	sharma	22	<null>
9	9	harshil	chauhan	22	<null>
10	10	ashish	sharma	22	<null>

Rename a Column

-- Renaming the column firstName --> first

ALTER TABLE customer

RENAME COLUMN firstName **TO** first;

	 id	 first	 lastName	 age
1	1	anish	sachdeva	22
2	2	piyush	gupta	22
3	3	akshita	chander	22
4	4	maddy	iota	22
5	5	gaurav	garg	22
6	6	smiti	thappar	22
7	7	apurva	puri	22
8	8	aviral	sharma	22
9	9	harshil	chauhan	22
10	10	ashish	sharma	22

-- Renaming Column lastName --> last

ALTER TABLE customer

RENAME COLUMN lastName **TO** last;

	 id	 first	 last	 age
1	1	anish	sachdeva	22
2	2	piyush	gupta	22
3	3	akshita	chander	21
4	4	maddy	iota	21
5	5	gaurav	garg	21
6	6	smiti	thappar	22
7	7	apurva	puri	23
8	8	aviral	sharma	23
9	9	harshil	chauhan	23
10	10	ashish	sharma	24

DML Statements & Commands

Insert Tuples

-- Adding Values Into Customer

```
INSERT INTO customer(firstName, lastName, age, date_of_birth, email) VALUES
('anish', 'sachdeva', 22, '07/04/1998', 'anishviewer@gmail.com'),
('piyush', 'gupta', 22, null, null),
('akshita', 'chander', 21, null, 'akshita@gmail.com'),
('maddy', 'iota', 21, null, 'maddy@gmail.com'),
('gaurav', 'garg', 21, null, 'g_garg@gmail.com'),
('smiti', 'thappar', 22, null, 'simms@gmail.com'),
('apurva', 'puri', 23, null, 'apurvapuriiii@gmail.com'),
('aviral', 'sharma', 23, null, 'asharma@gmail.com'),
('harshil', 'chauhan', 23, null, 'hchu@gmail.com'),
('ashish', 'sharma', 24, null, 'asharmagang@gmail.com');
```

	id	firstName	lastName	age	date_of_birth	email
1	1	anish	sachdeva	22	07/04/1998	anishviewer@gmail.com
2	2	piyush	gupta	22	<null>	<null>
3	3	akshita	chander	21	<null>	akshita@gmail.com
4	4	maddy	iota	21	<null>	maddy@gmail.com
5	5	gaurav	garg	21	<null>	g_garg@gmail.com
6	6	smiti	thappar	22	<null>	simms@gmail.com
7	7	apurva	puri	23	<null>	apurvapuriiii@gmail.com
8	8	aviral	sharma	23	<null>	asharma@gmail.com
9	9	harshil	chauhan	23	<null>	hchu@gmail.com
10	10	ashish	sharma	24	<null>	asharmagang@gmail.com

-- Inserting values into address table

```
INSERT INTO address(zip_code, street, city)
VALUES (110034, 'main street', 'bangalore'),
       (11345, 'amazing street', 'delhi'),
       (11653, 'redmond street', 'hyderabad');
```

	id	zip_code	street	city
1	1	110034	main street	bangalore
2	2	11345	amazing street	delhi
3	3	11653	redmond street	hyderabad

-- Inserting values into organization

```
INSERT INTO organization (name, address_id) VALUES
('microsoft', 3),
('google', 3),
('apple', 1),
('amazon', 2);
```

	id	name	address_id
1	1	microsoft	3
2	2	google	3
3	3	apple	1
4	4	amazon	2

-- Inserting values into the product

```
INSERT INTO product (name, cost, organization_id) VALUES
('xbox one s', 400, 1),
('xbox one x', 500, 1),
('surface book', 3000, 1),
('pixel 4a', 450, 2),
('pixel book', 1200, 1),
('chrome book', 800, 1),
('iphone 12', 1200, 1),
('apple tv 4k', 400, 1),
('macbook pro 13', 1500, 1),
```

```
('macbook pro 16', 2300, 1);
```

	id	name	cost	organization_id
1	1	xbox one s	400	1
2	2	xbox one x	500	1
3	3	surface book	3000	1
4	4	pixel 4a	450	2
5	5	pixel book	1200	1
6	6	chrome book	800	1
7	7	iphone 12	1200	1
8	8	apple tv 4k	400	1
9	9	macbook pro 13	1500	1
10	10	macbook pro 16	2300	1

```
-- Insert into the Orders table
```

```
INSERT INTO "order" (customer_id, product_id, date) VALUES
```

```
(1, 1, '07/04/2020'),
```

```
(1, 2, '07/04/2020'),
```

```
(1, 3, '07/04/2020'),
```

```
(2, 1, '07/04/2020'),
```

```
(2, 2, '07/04/2020'),
```

```
(3, 1, '07/04/2020'),
```

```
(4, 2, '07/04/2020'),
```

```
(4, 5, '07/04/2020'),
```

```
(5, 7, '07/04/2020'),
```

```
(6, 9, '07/04/2020'),
```

```
(6, 10, '07/04/2020'),
```

```
(7, 4, '07/04/2020'),
```

```
(8, 6, '07/04/2020'),
```

```
(9, 10, '07/04/2020'),
```

```
(9, 1, '07/04/2020');
```

	id	customer_id	product_id	date
1	1	1	1	07/04/2020
2	2	1	2	07/04/2020
3	3	1	3	07/04/2020
4	4	2	1	07/04/2020
5	5	2	2	07/04/2020
6	6	3	1	07/04/2020
7	7	4	2	07/04/2020
8	8	4	5	07/04/2020
9	9	5	7	07/04/2020
10	10	6	9	07/04/2020
11	11	6	10	07/04/2020
12	12	7	4	07/04/2020
13	13	8	6	07/04/2020
14	14	9	10	07/04/2020
15	15	9	1	07/04/2020

Delete Tuples

-- Delete all entries in customer where the email id is null

DELETE FROM customer

WHERE email IS NULL;

	id	firstName	lastName	age	date_of_birth	email
1	1	anish	sachdeva	22	07/04/1998	anishviewer@gmail.com
2	3	akshita	chander	21	<null>	akshita@gmail.com
3	4	maddy	iota	21	<null>	maddy@gmail.com
4	5	gaurav	garg	21	<null>	g_garg@gmail.com
5	6	smiiti	thappar	22	<null>	simms@gmail.com
6	7	apurva	puri	23	<null>	apurvapuriii@gmail.com
7	8	aviral	sharma	23	<null>	asharma@gmail.com
8	9	harshil	chauhan	23	<null>	hchu@gmail.com
9	10	ashish	sharma	24	<null>	asharmagang@gmail.com

-- Delete all entries in customer where the age is less than 22

DELETE FROM customer

WHERE age < 22;

	id	firstName	lastName	age	date_of_birth	email
1	1	anish	sachdeva	22	07/04/1998	anishviewer@gmail.com
2	6	smiiti	thappar	22	<null>	simms@gmail.com
3	7	apurva	puri	23	<null>	apurvapuriii@gmail.com
4	8	aviral	sharma	23	<null>	asharma@gmail.com
5	9	harshil	chauhan	23	<null>	hchu@gmail.com
6	10	ashish	sharma	24	<null>	asharmagang@gmail.com

Update Tuples

-- change price of pixel 4a to --> 400

UPDATE product

SET cost = 400

WHERE name = 'pixel 4a';

	id	name	cost	organization_id
1	1	xbox one s	400	1
2	2	xbox one x	500	1
3	3	surface book	3000	1
4	4	pixel 4a	400	2
5	5	pixel book	1200	1
6	6	chrome book	800	1
7	7	iphone 12	1200	1
8	8	apple tv 4k	400	1
9	9	macbook pro 13	1500	1
10	10	macbook pro 16	2300	1

-- revert price of pixel 4a --> 450

UPDATE product

SET cost = 450

WHERE name = 'pixel 4a';

	id	name	cost	organization_id
1	1	xbox one s	400	1
2	2	xbox one x	500	1
3	3	surface book	3000	1
4	4	pixel 4a	450	2
5	5	pixel book	1200	1
6	6	chrome book	800	1
7	7	iphone 12	1200	1
8	8	apple tv 4k	400	1
9	9	macbook pro 13	1500	1
10	10	macbook pro 16	2300	1

Synopsis of Project with ER Diagram

There are 5 relationships in the project which are address, organization, product, customer and order. The address relationship stores addresses with properties : street, pin code, city and a unique id.

The organization relationship contains a unique organization id, name and the address of this organization's head office is mapped in the address table using a column for the address id. Many organizations can have the same address and hence have a many-to-one relationship with the address relationship.

The product relationship has the following properties: id, name, cost and organization_id. The id is a unique identifier for all products and acts the primary key for this relationship. The name is maximum 255 character string value and the organization_id is a foreign key that refers to the organization relationship and implies that a particular product has been manufactured or is being sold by that particular organization.

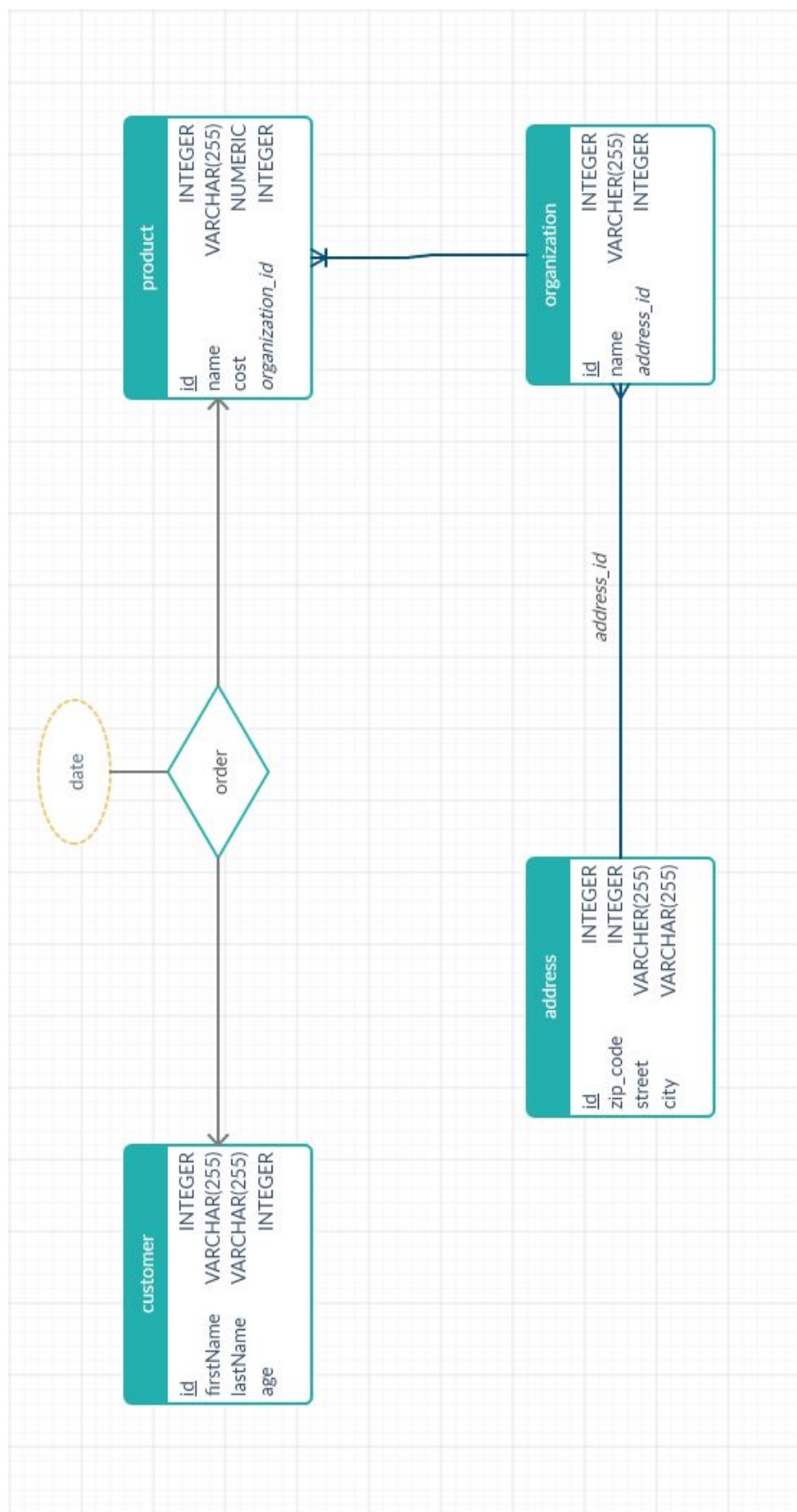
The cost in the product table is a numeric value and stores the cost of the particular product being sold on this particular application by the manufacturer in us dollars.

The customer relationship stores all customers that are part of this application irrespective of the fact whether they have made any orders or not. The customer relationship has the following properties : id, firstName, lastName, age, email and date_of_birth.

Here there is an additional constraint on age that all customers must have age ≥ 10 . The field id acts as the primary key of this relationship and is also set on auto increment so that values are assigned automatically as new customers are added to this table.

The last relationship in this system is the order relationship which is used to connect the customer and the product relationship. The order relationship has the following fields: id, customer_id, product_id, date. Here the field id is the unique primary key of this relationship and is set to auto increment.

The customer_id is a foreign key that refers to the id attribute of the customer relationship. The product_id is a foreign key that refers to the id attribute of the product relationship.



Program for Following SQL Commands

IN / NOT IN

-- All products that have been created by apple and google

```
SELECT *
FROM product
WHERE product.organization_id IN (
    SELECT id
    FROM organization
    WHERE organization.name = 'google' OR
        organization.name = 'apple'
);
```

	id	name	cost	organization_id
1	4	pixel 4a	450	2
2	5	pixel book	1200	2
3	6	chrome book	800	2
4	7	iphone 12	1200	3
5	8	apple tv 4k	400	3
6	9	macbook pro 13	1500	3
7	10	macbook pro 16	2300	3

-- All orders for products that are not from apple

```
SELECT * FROM order
WHERE order.product_id NOT IN (
    SELECT product.id FROM product
    WHERE product.organization_id = (
        SELECT organization.id
        FROM organization
```

```

WHERE organization.name = 'apple'
)
);

```

	id	customer_id	product_id	date
1	1	1	1	07/04/2020
2	2	1	2	07/04/2020
3	3	1	3	08/04/2020
4	4	2	1	10/04/2020
5	5	2	2	11/04/2020
6	6	3	1	11/04/2020
7	7	4	2	12/04/2020
8	8	4	5	13/04/2020
9	12	7	4	25/04/2020
10	13	8	6	30/04/2020
11	15	9	1	03/05/2020

Aggregate Functions

-- Select maximum cost product

```

SELECT *
FROM product
WHERE cost = (
    SELECT MAX(cost) as maximum_cost
    FROM product
);

```

	id	name	cost	organization_id
1	3	surface book	3000	1

-- Select Minimum Costing Product

```

SELECT *
FROM product
WHERE cost = (
    SELECT MIN(cost) as maximum_cost

```

```

FROM product
);

```

	id	name	cost	organization_id
1	1	xbox one s	400	1
2	8	apple tv 4k	400	3

-- Average cost of all products available on this platform

```

SELECT avg(cost) as avergae_product_cost
FROM product;

```

	avergae_product_cost
1	1175

-- Number of products available on the market

```

SELECT count(*) as number_of_products
FROM product;

```

	number_of_products
1	10

-- Number of total orders placed

```

SELECT count(*) as total_orders
FROM "order";

```

	total_orders
1	15

-- Number of orders placed by customer no 2

```

SELECT count(*) as total_orders
FROM "order"
WHERE customer_id = 2;

```

	total_orders
1	2

Group by + Having clause

-- Seeing customers who have made more than one order

```
SELECT "order".customer_id,
       c.firstName,
       c.lastName,
       count(*) as number_of_orders
FROM "order"
JOIN customer c on "order".customer_id = c.id
GROUP BY "order".customer_id
having number_of_orders > 1;
```

	customer_id	firstName	lastName	number_of_orders
1	1	anish	sachdeva	3
2	2	piyush	gupta	2
3	4	maddy	iota	2
4	6	smiti	thappar	2
5	9	harshil	chauhan	2

-- Average cost of all products per organization and total number

-- of products they have on the market and displaying those

-- entries where average cost is less than 1,500

```
SELECT product.organization_id,
       avg(cost) as average_cost ,
       count(*) as total_products_on_market
FROM product
GROUP BY organization_id
HAVING average_cost < 1500;
```

	organization_id	average_cost	total_products_on_market
1	1	1300	3
2	2	816.6666666666666	3
3	3	1350	4

Order By

-- Ordering customers by their age in ascending order and name in
 -- descending order by last name and ascending order by first name

```
SELECT *
FROM customer
ORDER BY age asc, lastName desc, firstName asc ;
```

	id	firstName	lastName	age	date_of_birth	email
1	4	maddy	iota	21	<null>	maddy@gmail.com
2	5	gaurav	garg	21	<null>	g_garg@gmail.com
3	3	akshita	chander	21	<null>	akshita@gmail.com
4	6	smiiti	thappar	22	<null>	simms@gmail.com
5	1	anish	sachdeva	22	07/04/1998	anishviewer@gmail.com
6	2	piyush	gupta	22	<null>	<null>
7	8	aviral	sharma	23	<null>	asharma@gmail.com
8	7	apurva	puri	23	<null>	apurvapuriii@gmail.com
9	9	harshil	chauhan	23	<null>	hchu@gmail.com
10	10	ashish	sharma	24	<null>	asharmagang@gmail.com

-- Number of Orders placed per customer, ordered by number of orders
 -- placed

```
SELECT customer_id,
       firstName,
       lastName,
       count(*) as number_orders_placed
FROM "order"
JOIN customer c on "order".customer_id = c.id
GROUP BY customer_id
ORDER BY number_orders_placed;
```

	customer_id	firstName	lastName	number_orders_placed
1	3	akshita	chander	1
2	5	gaurav	garg	1
3	7	apurva	puri	1
4	8	aviral	sharma	1
5	9	harshil	chauhan	2
6	2	piyush	gupta	2
7	4	maddy	iota	2
8	6	smiiti	thappar	2
9	1	anish	sachdeva	3

Join Statements & Commands

Natural Join

-- All orders placed per customer (Natural Join)

```
SELECT customer.id as customer_id,
       customer.firstName,
       customer.lastName,
       "order".id as order_id,
       "order".product_id as product_id
FROM "order", customer
WHERE "order".customer_id = customer.id;
```

	customer_id	firstName	lastName	order_id	product_id
1	1	anish	sachdeva	1	1
2	1	anish	sachdeva	2	2
3	1	anish	sachdeva	3	3
4	2	piyush	gupta	4	1
5	2	piyush	gupta	5	2
6	3	akshita	chander	6	1
7	4	maddy	iota	7	2
8	4	maddy	iota	8	5
9	5	gaurav	garg	9	7
10	6	smiti	thappar	10	9
11	6	smiti	thappar	11	10
12	7	apurva	puri	12	4
13	8	aviral	sharma	13	6
14	9	harshil	chauhan	14	10
15	9	harshil	chauhan	15	1

Inner Join

-- All orders placed per customer (Inner Join)

```
SELECT customer.id as customer_id,
       customer.firstName,
       customer.lastName,
       "order".id as order_id,
```

```

"order".product_id as product_id
FROM "order"
INNER JOIN customer ON "order".customer_id = customer.id;

```

Performing the above query we get similar result as Natural Join.

	customer_id	firstName	lastName	order_id	product_id
1	1	anish	sachdeva	1	1
2	1	anish	sachdeva	2	2
3	1	anish	sachdeva	3	3
4	2	piyush	gupta	4	1
5	2	piyush	gupta	5	2
6	3	akshita	chander	6	1
7	4	maddy	iota	7	2
8	4	maddy	iota	8	5
9	5	gaurav	garg	9	7
10	6	smiti	thappar	10	9
11	6	smiti	thappar	11	10
12	7	apurva	puri	12	4
13	8	aviral	sharma	13	6
14	9	harshil	chauhan	14	10
15	9	harshil	chauhan	15	1

Outer Join

-- All orders placed per customer (Outer Join)

```

SELECT customer.id as customer_id,
       customer.firstName,
       customer.lastName,
       "order".id as order_id,
       "order".product_id as product_id
FROM customer
OUTER JOIN "order" ON "order".customer_id = customer.id;

```

	customer_id	firstName	lastName	order_id	product_id
1	1	anish	sachdeva	1	1
2	1	anish	sachdeva	2	2
3	1	anish	sachdeva	3	3
4	2	piyush	gupta	4	1
5	2	piyush	gupta	5	2
6	3	akshita	chander	6	1
7	4	maddy	iota	7	2
8	4	maddy	iota	8	5
9	5	gaurav	garg	9	7
10	6	smiti	thappar	10	9
11	6	smiti	thappar	11	10
12	7	apurva	puri	12	4
13	8	aviral	sharma	13	6
14	9	harshil	chauhan	15	1
15	9	harshil	chauhan	14	10
16	10	ashish	sharma	<null>	<null>

Introduction to PL/SQL

Variables in PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

Variable Declaration

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is:

```
Variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, variable_name is a valid identifier in PL/SQL, datatype must be a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below.

```
sales number(10, 2);
```

```
pi CONSTANT double precision := 3.1415;
```

```
name varchar2(25);
```

```
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a constrained declaration. Constrained declarations require less memory than unconstrained declarations. For example:

```
sales number(10, 2);
```

```
name varchar2(25);
```

```
address varchar2(100);
```

Initializing Variables In PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following:

1. The **DEFAULT** Keyword
2. The **assignment** operator

For Example:

```
counter binary_integer := 0;
```

```
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a NULL value using the NOT NULL constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables:

```
DECLARE
    a integer := 10;
    b integer := 20;
    c integer;
    f real;
BEGIN
    c := a + b;
    dbms_output.put_line('Value of c: ' || c);
    f := 70.0/3.0;
    dbms_output.put_line('Value of f: ' || f);
END;
/
```

When the above code is executed, it produces the following result:

```
Value of c: 30
```

```
Value of f: 23.333333333333333333
```

```
PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block.

However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope:

1. **Local Variables:** Variables declared in an inner block and not accessible to outer blocks.
2. **Global Variables:** Variables declared in the outermost or a package.

Following example shows the usage of Local and Global variables in its simple form:

```
DECLARE
    -- Global variables
    num1 number := 95;
    num2 number := 85;
BEGIN
    dbms_output.put_line('Outer Variable num1: ' || num1);
    dbms_output.put_line('Outer Variable num2: ' || num2);

    DECLARE
        -- Local variables
        num1 number := 195;
        num2 number := 185;
    BEGIN
        dbms_output.put_line('Inner Variable num1: ' || num1);
        dbms_output.put_line('Inner Variable num2: ' || num2);
    END;
END;
/
```

When the above code is executed, it produces the following result:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185
PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You can use the SELECT INTO statement of SQL to assign values to PL/SQL variables. For each item in the SELECT list, there must be a corresponding, type-compatible variable in

the INTO list. The following example illustrates the concept. Let us create a table named CUSTOMERS:

```
CREATE TABLE CUSTOMERS (  
    ID INT NOT NULL,  
    NAME VARCHAR (20) NOT NULL,  
    AGE INT NOT NULL,  
    ADDRESS CHAR (25),  
    SALARY DECIMAL (18, 2),  
    PRIMARY KEY (ID)  
);
```

Table Created

Let us now insert some values in the table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables using the SELECT INTO clause of SQL:

```
DECLARE
```

```
c_id customers.id%type := 1;
c_name customers.name%type;
c_addr customers.address%type;
c_sal customers.salary%type;
BEGIN
    SELECT name, address, salary INTO c_name, c_addr, c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line
    ('Customer ' || c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

When the above code is executed, it produces the following result:

```
Customer Ramesh from Ahmedabad earns 2000
```

```
PL/SQL procedure completed successfully
```

I/O in PL/SQL

Many, perhaps most, of the PL/SQL programs you write need to interact only with the underlying Oracle database using SQL. However, there will inevitably be times when you will want to send information from PL/SQL to the external environment or read information from some external source (screen, file, etc.) into PL/SQL. This chapter explores some of the most common mechanisms for I/O in PL/SQL, including the following built-in packages:

1. DBMS_PIPE, to send and receive information between sessions, asynchronously.
2. DBMS_OUTPUT, to send messages from a PL/SQL program to other PL/SQL programs in the same session, or to a display window running SQL*Plus.
3. UTL_FILE, which allows a PL/SQL program to read information from a disk file, and write information to a file.

Database Pipes

The DBMS_PIPE package allows two or more sessions in the same instance to communicate. Oracle *pipes* are similar in concept to the pipes used in UNIX, but Oracle pipes are not implemented using the operating system pipe mechanisms. Information sent through Oracle pipes is buffered in the system global area (SGA). All information in pipes is lost when the instance is shut down.

Depending upon your security requirements, you may choose to use either a *public pipe* or a *private pipe*.

Attention: Pipes are independent of transactions. Be careful using pipes when transaction control can be affected.

Creating the DBMS_PIPE Package

To create the DBMS_PIPE package, submit the DBMSPIPE.SQL and PRVTPPIPE.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script.

Public Pipes

You can create a public pipe either implicitly or explicitly. For *implicit* public pipes, the pipe is automatically created when referenced for the first time, and it disappears when it no longer contains data. Because the pipe descriptor is stored in the SGA, there is some space usage overhead until the empty pipe is aged out of the cache.

You can create an *explicit* public pipe by calling the CREATE_PIPE function with the PRIVATE flag set to FALSE. You must deallocate explicitly-created pipes by calling the REMOVE_PIPE function.

The domain of a public pipe is the schema in which it was created, either explicitly or implicitly.

Writing and Reading

Each public pipe works asynchronously. Any number of schema users can write to a public pipe, as long as they have EXECUTE permission on the DBMS_PIPE package, and know the name of the public pipe.

Any schema user with the appropriate privileges and knowledge can read information from a public pipe. However, once buffered information is read by one user, it is emptied from the buffer, and is not available for other readers of the same pipe.

The sending session builds a message using one or more calls to the PACK_MESSAGE procedure. This procedure adds the message to the session's local message buffer. The information in this buffer is sent by calling the SEND_MESSAGE procedure, designating the pipe name to be used to send the message. When SEND_MESSAGE is called, all messages that have been stacked in the local buffer are sent.

A process that wants to receive a message calls the RECEIVE_MESSAGE procedure, designating the pipe name from which to receive the message. The process then calls the UNPACK_MESSAGE procedure to access each of the items in the message.

Private Pipes

You must explicitly create a private pipe by calling the CREATE_PIPE function. Once created, the private pipe persists in shared memory until you explicitly deallocate it by calling the REMOVE_PIPE function. A private pipe is also deallocated when the database instance is shut down.

You cannot create a private pipe if an implicit pipe exists in memory and has the same name as the private pipe you are trying to create. In this case CREATE_PIPE returns an error.

Access to a private pipe is restricted to the following:

- sessions running under the same userid as the creator of the pipe

- stored subprograms executing in the same userid privilege domain as the pipe creator
- users connected as SYSDBA or INTERNAL

An attempt by any other user to send or receive messages on the pipe, or to remove the pipe, results in an immediate error. Any attempt by another user to create a pipe with the same name also causes an error.

As with public pipes, you must first build your message using calls to `PACK_MESSAGE` before calling `SEND_MESSAGE`. Similarly you must call `RECEIVE_MESSAGE` to retrieve the message before accessing the items in the message by calling `UNPACK_MESSAGE`.

Errors

DBMS_PIPE package routines can return the following errors:

ORA-23321: Pipename may not be null

ORA-23322: Insufficient privilege to access pipe

ORA-23321 can be returned by `CREATE_PIPE`, or any subprogram that takes a pipe name as a parameter. ORA-23322 can be returned by any subprogram that references a private pipe in its parameter list.

CREATE_PIPE

Call `CREATE_PIPE` to explicitly create a public or private pipe. If the `PRIVATE` flag is `TRUE`, the pipe creator is assigned as the owner of the private pipe. Explicitly created pipes can only be removed by calling `REMOVE_PIPE`, or by shutting down the instance.

Warning: Do not use a pipe name beginning with `ORA$`; these names are reserved for use by Oracle Corporation.

Packages in PL/SQL

What is a Package?

A **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

A package always has a **specification**, which declares the **public items** that can be referenced from outside the package.

If the public items include cursors or subprograms, then the package must also have a **body**. The body must define queries for public cursors and code for public subprograms. The body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an **initialization part**, whose statements initialize variables and do other one-time setup steps, and an exception-handling part. You can change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box.

In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using a **call specification**, which maps the external subprogram name, parameter types, and return type to their SQL counterparts.

The **AUTHID clause** of the package specification determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.

The **ACCESSIBLE BY clause** of the package specification lets you specify a white list of PL/SQL units that can access the package. You use this clause in situations like these:

- You implement a PL/SQL application as several packages—one package that provides the application programming interface (API) and helper packages to do the

work. You want clients to have access to the API, but not to the helper packages. Therefore, you omit the ACCESSIBLE BY clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.

- You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict use of the package to the intended units, you list them in the ACCESSIBLE BY clause in the package specification.

Reasons to Use Packages

Packages support the development and maintenance of reliable, reusable code with the following features:

- **Modularity**

Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules. You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined. This practice aids application development.

- **Easier Application Design**

When designing an application, all you need initially is the interface information in the package specifications. You can code and compile specifications without their bodies. Next, you can compile standalone subprograms that reference the packages. You need not fully define the package bodies until you are ready to complete the application.

- **Hidden Implementation Details**

Packages let you share your interface information in the package specification, and hide the implementation details in the package body. Hiding the implementation details in the body has these advantages:

- You can change the implementation details without affecting the application interface.
- Application users cannot develop code that depends on implementation details that you might want to change.
- **Added Functionality**

Package public variables and cursors can persist for the life of a session. They can be shared by all subprograms that run in the environment. They let you maintain data across transactions without storing it in the database. (For the situations in which package public variables and cursors do not persist for the life of a session, see "Package State".)
- **Better Performance**


The first time you invoke a package subprogram, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O.

Packages prevent cascading dependencies and unnecessary recompiling. For example, if you change the body of a package function, Oracle Database does not recompile other subprograms that invoke the function, because these subprograms depend only on the parameters and return value that are declared in the specification.
- **Easier to Grant Roles**

You can grant roles on the package, instead of granting roles on each object in the package.

Package Specification

A **package specification** declares **public items**. The scope of a public item is the schema of the package. A public item is visible everywhere in the schema. To reference a public item that is in scope but not visible, qualify it with the package name.



Each public item declaration has all information needed to use the item. For example, suppose that a package specification declares the function factorial this way:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

Exceptions in PL/SQL

An exception is a PL/SQL error that is raised during program execution, either implicitly by TimesTen or explicitly by your program. Handle an exception by trapping it with a handler or propagating it to the calling environment.

For example, if your `SELECT` statement returns multiple rows, TimesTen returns an error (exception) at runtime. As the following example shows, you would see TimesTen error 8507, then the associated `ORA` error message. (`ORA` messages, originally defined for Oracle Database, are similarly implemented by TimesTen.)

```
Command> DECLARE
```

```
> v_lname VARCHAR2 (15);
```

```
> BEGIN
```

```
> SELECT last_name INTO v_lname
```

```
> FROM employees
```

```
> WHERE first_name = 'John';
```

```
> DBMS_OUTPUT.PUT_LINE ('Last name is : ' || v_lname);
```

```
> END;
```

```
> /
```

```
8507: ORA-01422: exact fetch returns more than requested number of rows
```

```
8507: ORA-06512: at line 4
```

The command failed.

You can handle such exceptions in your PL/SQL block so that your program completes successfully.

For example:

```
Command> DECLARE
```

```
> v_lname VARCHAR2 (15);
```

```
> BEGIN

>   SELECT last_name INTO v_lname

>   FROM employees

>   WHERE first_name = 'John';

>   DBMS_OUTPUT.PUT_LINE ('Last name is : ' || v_lname);

> EXCEPTION

>   WHEN TOO_MANY_ROWS THEN

>   DBMS_OUTPUT.PUT_LINE (' Your SELECT statement retrieved multiple

>   rows. Consider using a cursor. ');

> END;

> /
```

```
Your SELECT statement retrieved multiple rows. Consider using a cursor.
```

PL/SQL procedure successfully completed.

Exception Types

There are three types of exceptions:

- Predefined exceptions are error conditions that are defined by PL/SQL.
- Non-predefined exceptions include any standard TimesTen errors.
- User-defined exceptions are exceptions specific to your application.

In TimesTen, these three types of exceptions are used in the same way as in Oracle Database.

Exception	Description
Predefined TimesTen error	One of approximately 20 errors that occur most often in PL/SQL code
Non-predefined TimesTen error	Any other standard TimesTen error
User-defined error	Error defined and raised by the application


Trapping Exceptions

Trapping predefined TimesTen errors

Trap a predefined TimesTen error by referencing its predefined name in your exception-handling routine. PL/SQL declares predefined exceptions in the `STANDARD` package.

Table below lists predefined exceptions supported by TimesTen, the associated `ORA` error numbers and `SQLCODE` values, and descriptions of the exceptions.

Exception name	Oracle Database error number	SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPENED	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
NO_DATA_FOUND	ORA-01403	+100
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
STORAGE_ERROR	ORA-06500	-6500



SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Triggers in PL/SQL

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes –

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating 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)
```

```

DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;

```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name – Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will 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 will 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, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will 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.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previously:

```
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	
+-----+-----+-----+-----+-----+										

The following program creates 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;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Trigger created.
```

The following points need to be considered here –

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-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.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

```
INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result –

```
Old salary:
```

```
New salary: 7500
```

```
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table –

```
UPDATE customers
```

```
SET salary = salary + 500
```

```
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result –

```
Old salary: 1500
```

```
New salary: 2000
```

```
Salary difference: 500
```

Transactions in SQL

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID.

- Atomicity – ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- Consistency – ensures that the database properly changes states upon a successfully committed transaction.
- Isolation – enables transactions to operate independently of and transparent to each other.
- Durability – ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control

The following commands are used to control transactions.

- COMMIT – to save the changes.
- ROLLBACK – to roll back the changes.
- SAVEPOINT – creates points within the groups of transactions in which to ROLLBACK.
- SET TRANSACTION – Places a name on a transaction.

Transactional Control Commands

Transactional control commands are only used with the DML Commands such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

Commit

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows:

```
COMMIT;
```

Example

Consider the CUSTOMERS table having the following records:

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
7	Muffy	24	Indore	10000.00

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
WHERE AGE = 25;
```

```
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Rollback

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows:

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records:

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

7	Muffy	24	Indore	10000.00
---	-------	----	--------	----------

+-----+				
---------	--	--	--	--

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
```

```
WHERE AGE = 25;
```

```
SQL> ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

+-----+				
---------	--	--	--	--

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

7	Muffy	24	Indore	10000.00
---	-------	----	--------	----------

+-----+				
---------	--	--	--	--

Savepoint

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

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
7	Muffy	24	Indore	10000.00

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
```

```
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
SQL> ROLLBACK TO SP2;
```

```
Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2.

```
SQL> SELECT * FROM CUSTOMERS;
```

ID	NAME	AGE	ADDRESS	SALARY
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
7	Muffy	24	Indore	10000.00

```
6 rows selected
```