STRUCTURE QUERY LANGUAGE

UNIT – III

Reference:

- ✓ ***SQL and PL/SQL Using ORACLE by Ivan Bayross***
- ✓ ***Oracle 11g with PL/SQL Approach by Sham Tickoo & Sunil Raina***
- ✓ ***SQL and PL/SQL for Oracle 11g (BLACK BOOK) by Dr. P.S. Deshpande***

Structured Query Language

- SQL (Structured Query Language) is a database sub-language for querying and modifying relational databases.
- It was developed by IBM Research in the mid 70's and standardized by ANSI in 1986.
- Relational Model defines two *root* languages for accessing a relational database -- Relational Algebra and Relational Calculus.
- Relational Algebra is a low-level, operator-oriented language. Creating a query in Relational Algebra involves combining relational operators using algebraic notation.
- Relational Calculus is a high-level, declarative language. Creating a query in Relational Calculus involves describing what results are desired.

Functions of a DBMS

- **Data definition:** SQL lets a user define the structure and organization of the stored data and relationships among the stored data items.
- **Data retrieval:** SQL allows a user or an application program to retrieve stored data from the database and use it.
- **Data manipulation:** SQL allows a user or an application program to update the database by adding new data, removing old data, and modifying previously stored data.
- **Access control:** SQL can be used to restrict a user's ability to retrieve, add, and modify data, protecting stored data against unauthorized access.
- **Data sharing:** SQL is used to coordinate data sharing by concurrent users, ensuring that they do not interfere with one another.
- **Data integrity:** SQL defines integrity constraints in the database, protecting it from corruption due to inconsistent updates or system failures.
- SQL is thus a comprehensive language for controlling and interacting with a database management system.

Characteristics of SQL

SQL is both an easy-to-understand language and a comprehensive tool for managing data. Here are some of the major features of SQL and the market forces that have made it successful:

- **Vendor independence**
- **Relational foundation**
- **High-level, English-like structure**
- **Interactive, ad hoc queries**
- **Programmatic database access**
- **Multiple views of data**
- **Complete database language**

SQL Common Data Types

Datatype	Specification	Remarks
Char	CHAR(SIZE)	255 characters
Varchar2	Vharchar2(Size)	4000 bytes
Varchar	Varchar(Size)	2000 bytes
Date	Date	Stores year, month,hour
Number	Number(Size)	10^{126} To 10^{-130}

Types of SQL commands

SQL statements are often divided into three categories:

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- TCL (Transaction Control Language)

DDL (Data Definition Language)

DDL (Data Definition Language). These SQL statements define the structure of a database, including rows, columns, tables, indexes, and database specifics such as file locations. DDL SQL statements are more part of the DBMS and have large differences between the SQL variations. DML SQL commands include the following:

- **CREATE** to make a new database, table, index, or stored query.
- **DROP** to destroy an existing database, table, index, or view.
- **ALTER** to modify the structure of a Database/ database Objects

DML (Data Manipulation Language)

- These SQL statements are used to retrieve and manipulate data. This category encompasses the most fundamental commands including **DELETE, INSERT, SELECT, and UPDATE**. DML SQL statements have only minor differences between SQL variations. DML SQL commands include the following:
  - DELETE to remove rows.
  - INSERT to add a row.
  - SELECT to retrieve row.
  - UPDATE to change data in specified columns.

TCL (Transaction Control Language)

Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

- **COMMIT** - save work done
- **SAVEPOINT** - identify a point in a transaction to which you can later roll back
- **ROLLBACK -** restore database to original since the last COMMIT
- **SET TRANSACTION -** Change transaction options like isolation level and what rollback segment to use

Tables

- In SQL2, can use the CREATE TABLE command for specifying the primary key attributes, secondary keys, and referential integrity constraints (foreign keys).
- Key attributes can be specified via the PRIMARY KEY and UNIQUE phrases

```
CREATE TABLE DEPT (
  DNAME VARCHAR(10)NOT NULL,
  DNUMBER NUMBER NOT NULL,
  Name CHAR(9),Post    CHAR(20),
  PRIMARY KEY (DNUMBER),
  UNIQUE (DNAME) );
```

DROP TABLE

- Used to remove a relation (base table) and its definition
- The relation can no longer be used in queries, updates, or any other commands since its description no longer exists
- Example:

Syntax: **DROP TABLE <tablename>;**

Example :**DROP TABLE STUDENT;**

ALTER TABLE

- Used to add an attribute to one of the base relations
  - The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is not allowed for such an attribute
  - Example:

```
ALTER TABLE EMPLOYEE ADD JOB VARCHAR(12);
```

- The database users must still enter a value for the new attribute JOB for each EMPLOYEE tuple.
  - This can be done using the UPDATE command.

ALTER TABLE

- Used to remove an attribute from the relation.

Syntax: ALTER TABLE <table name> DROP COLUMN <column name >

Example: ALTER TABLE studemt DROP COLUMN contact_no;

MODIFYING EXISTING COLUMNS

Syntax: ALTER TABLE<table name> MODIFY (<Column name> <new datatype> (<new size>));

Example :ALTER TABLE student MODIFY (Name Varchar2(30));

Displaying the Table Structure

Syntax: DESCRIBE <table name>;

Example : DESCRIBE student;

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table *r* (*A*1 *D*1, *A*2 *D*2, ..., *An Dn,*

  (integrity-constraint1), ..., (integrity-constraintk))

  - *r* is the name of the relation

Example:

create table *branch*

(*branch_name* char(15) **not null,***branch_city* char(30),

assets integer);

CHECK CONSTRAINT

Example:

CREATE TABLE student_mstr(Std_id varchar2(10)

CHECK (Std_id LIKE ‘M%’), name varchar2(20)

CHECK (name =UPPER(name)), address varchar2(30));

DEFAULT VALUE

Example:

CREATE TABLE bank_mstr (name varchar2(20), cust_id number(10), Curbal number(8,2) **DEFAULT 0**);

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)

- Consider a person who needs to know a customer’s loan number but has no need to see the loan amount. This person should see a relation described, in SQL, by

  (**select** *customer_name, loan_number*

  from *borrower, loan*

  where *borrower.loan_number = loan.loan_number*)

- A **view** provides a mechanism to hide certain data from the view of certain users. Any relation that is not of the conceptual model but is made visible to a user as a **“virtual relation”** is called a **view**.

- **Syntax: CREATE VIEW <Viewname > As**

  Select <columnname1>, <columnname2> FROM <table name> where <columnname>=<expression list>;

Example

create view v1 as

(**select** branch_name, customer_name **from** depositor);

Example : SELECT customer_name from V1;

Example: DROP VIEW v1;

Indexes

- One of the greatest benefits of holding information in a database is the ability to quickly retrieve it. When querying a database, it is possible to apply criteria to ask for a specific set of rows. For example, returning all employees that use a specific vehicle or returning employees within a particular range of payroll IDs.

- If the DBMS needed to scan through all of the data within a table in order to retrieve the desired information, the process would be very slow, particularly for tables with millions of rows. To improve retrieval performance a table can have one or more indexes. Each index provides a fast "look-up" facility for rows, as an index in a book allows all references to a topic to be located without reading every page.

How do Indexes Work?

When an index is created, it records the location of values in a table that are associated with the column that is indexed. Entries are added to the index when new data is added to the table.

When a query is executed against the database and a condition is specified on a column in the WHERE clause that is indexed, the index first searched for the values specified in the WHERE clause.

Aggregate functions

These functions operate on the multi-set of values of a column of a relation, and return a value

- **avg:** average value
- **min:** minimum value
- **max:** maximum value
- **sum:** sum of values
- **count:** number of values

Queries & Subqueries

- Find the average account balance at the Ghaziabad branch.

    select avg *(balance)* **from** *account*

    where *branch_name* = ‘Ghaziabad’

- Find the number of depositors in the bank.

    select count (*) **from** *customer*

- Find the number of tuples in the *customer* relation.

    select count (distinct *customer_name)*

    from *depositor*

Group By & Having Clause

Group By clause is used in the Select statement to collect data from multiple records and group the results that have matching values for one or more columns.

E.g. Write a SQL query which will return the job description of employees along with the total salary:

```
SELECT job, SUM(salary) from EMP_DATA GROUP BY job;
```

E.g. Write a SQL query which will return the job description of employees with the minimum salary in each job description

```
SELECT job, MIN(salary) from EMP_DATA GROUP BY job;
```

E.g. Write a SQL query which will return the job description of employees with total number of employees in each job description

```
SELECT job, COUNT(*) from EMP_DATA GROUP BY job;
```

Group By & Having Clause

HAVING clause is used in the Select statement to FILTER the data returned by the GROUP BY clause.

E.g. Write a SQL query which will return maximum salary of the employees than 4000.

SELECT Deptno, MAX(salary) from EMP_DATA GROUP BY Deptno HAVING MAX(salary)>4000;

E.g. Write a SQL query which will return minimum salary of the employees than 1000.

SELECT Deptno, MIN(salary) from EMP_DATA GROUP BY Deptno HAVING MAX(salary)>1000;

Group By & Having Clause

- **Find the number of depositors for each branch.**

  select *branch_name,* **count (distinct** *customer_name)*

  from *depositor, account*

  where *depositor.account_number = account.account_number*

  group by *branch_name*

- **Find the names of all branches where the average account balance is more than 10000.**

  select *branch_name,* **avg** (*balance*)

  from Bank_detai**l group by** *branch_name*

  having avg (*balance*) >10000

Nested Sub-query

- SQL provides a mechanism for the nesting of sub-queries.
- A subquery is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Examples

- Find all customers who have both an account and a loan at the bank.

  select distinct *customer_name*

  from *borrower*

  where *customer_name* **in** (**select** *customer_name*

  from *depositor*)

- Find all customers who have a loan at the bank but do not have an account at the bank

  select distinct *customer_name*

  from *borrower*

  where *customer_name* **not in** (**select** *customer_name*

  from *depositor*)

Database Operations

- Insert
- Update
- Delete

INSERT Operation

- Add a new tuple to *account*

  insert into *account*
  values (‘A-9732’, ‘Ghaziabad’,1200)
  or equivalently
  insert into *account* (
  branch_name, balance, account_number)
  values (‘Ghaziabad’, 1200, ‘A-9732’)

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values (‘A-777’,‘Ghaziabad’, *null*)

UPDATES Operation

- Increase all accounts with balances over Rs. 10,000 by 600, all other accounts receive 500.

- Write two **update** statements:

  update *account* **set** *balance* = *balance +600*

  where *balance* > 10000

  update *account* **set** *balance* = *balance* + 500

  where *balance* = 10000

DELETE Operation

- Delete all account tuples at the Ghaziabad branch:

  delete from *account*

  where *branch_name* = ‘*Ghaziabad*’

- Delete all accounts at every branch located in the city ‘Kanpur’:

  delete from *account*

  where *branch_name* **in** (**select** *branch_name*

  from *branch* **where** *branch_city* = ‘*Kanpur*’)

- Delete the record of all accounts with balances below the average at the bank.

  delete from *account* **where** *balance* < (**select avg** (*balance*) **from** *account*)

Joins

- An SQL JOIN clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two tables by using values common to each.

SQL specifies four types of JOINs:

✓ INNER JOIN

✓ OUTER JOIN

✓ CROSS JOIN

✓ SELF JOIN

Inner Join

Inner joins are also known as Equi Joins. There are the most common joins used in SQL . They are known as equi joins because the where statement generally compares two columns from two tables with the equivalence operator =.

This type of join can be used in situation where selecting only two those rows that have values in common in the columns specified in the ON clause, is required. In short INNER JOIN returns all rows from both tables where there is a match.

Syntax: SELECT <column name1> FROM <tablename>

INNER JOIN <tablename2> ON <tablename1>.<columnname1>= <Tablenem2>.<columnname2>

WHERE <condition>

Example:

```
SELECT E.Emp_No, E.Name,B.Name, E.Dept,E.Desig
From Emp_master E INNER JOIN Branch_master B
ON B.Branch_no=E.Branch_No;
```

OUTER JOIN

Outer Joins are similar to inner joins, but give a bit more flexibility when selecting data from related tables. This type of join can be used in situation where it is desired , to select all rows from the table on the left (or right) regardless of whether the other table has values in common.

Example:

```
SELECT  E.LNAME, E.DEPT,C.CONTACT FROM
EMP_MASTER E LEFT JOIN CONCT_MASTER C ON
E.EMP_NO=C.CODE_NO;
```

or

```
SELECT E.LNAME, E.DEPT,C.CONTACT FROM
EMP_MASTER E ,CONCT_MASTER C WHERE
E.EMP_NO=C.CODE_NO(+);
```

OUTER JOIN

The LEFT JOIN can be used which returns all the rows from the first table (EMP_MASTER) even if there are no matches in the second table (CONCT_MASTER) .

RIGHT JOIN

Example: SELECT E.LNAME, E.DEPT,C.CONTACT FROM CONCT_MASTER C RIGHT JOIN EMP_MASTER E ON C.CODE_NO=E.EMP_NO;

or

SELECT E.LNAME, E.DEPT,C.CONTACT FROM CONCT_MASTER C , EMP_MASTER E WHERE C.CODE_NO(+)=E.EMP_NO;

Natural Join

- A **natural join** offers a further specialization of equi-joins. The join predicate arises implicitly by comparing all columns in both tables that have the same column-name in the joined tables. The resulting joined table contains only one column for each pair of equally-named columns.

- Example:

SELECT * **FROM** employee **NATURAL JOIN** department ;

Cross Join

- A CROSS JOIN returns what's known as a Cartesian product . This means that the join combines every row from the left table with the every row in the right table.
- This type of join can be used in the situations where it is desired , of select all possible combinations of rows and columns from both the tables.
- This type of join is usually not preferred as it may run for a very long time and produce a huge result set that may not be useful.

```
SELECT * FROM employee, department;
```

Self Join

- A self-join is joining a table to itself.

```
SELECT      F.EmployeeID, F.LastName, S.EmployeeID,
             S.LastName, F.Country
FROM    Employee F, Employee S
WHERE       F.Country = S.Country
ORDER BY    F.EmployeeID, S.EmployeeID;
```

Set Operations

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations: $\cup, \cap, -$

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

Contd…

- Find all customers who have a loan, an account, or both:

  (**select** *customer_name* **from** *depositor*)
  UNION
  (select *customer_name* **from** *borrower)*

- Find all customers who have both a loan and an account.

  (**select** *customer_name* **from** *depositor*)
  intersect
  (select *customer_name* **from** *borrower)*

- Find all customers who have an account but no loan.

  (select *customer_name* **from** *depositor*)
  except
  (select *customer_name* **from** *borrower)*

Left Outer Join

Use this when you only want to return rows that have matching data in the left table, even if there's no matching rows in the right table.

SELECT * FROM Individual AS Ind LEFT JOIN Publisher AS Pb ON Ind.IndividualId = Pb.IndividualId

Left Table

Id	FirstName	LastName	UserName
1	Fred	Flinstone	freddo
2	Homer	Simpson	homey
3	Homer	Brown	notsofamous
4	Ozzy	Ozzbourne	sabbath
5	Homer	Gain	noplacelike

Id	FirstName	LastName	UserName
1	Fred	Flinstone	freddo
2	Homer	Simpson	homey
3	Homer	Brown	notsofamous
4	Ozzy	Ozzbourne	sabbath
5	Homer	Gain	noplacelike

Left Table

IndividualId	AccessLevel
1	Administrator
2	Contributor
3	Contributor
4	Contributor
10	Administrator

Right Table

Result Table

IndividualId	FirstName	LastName	UserName	IndividualId	AccessLevel
1	Fred	Flinstone	freddo	1	Administrator
2	Homer	Simpson	homey	2	Contributor
3	Homer	Brown	notsofamous	3	Contributor
4	Ozzy	Osbourne	sabbath	4	Contributor
5	Homer	Gain	noplacelike	NULL	NULL

Right Outer Join

Use this when you only want to return rows that have matching data in the right table, even if there's no matching rows in the left table.

```
SELECT * FROM Individual AS Ind
RIGHT JOIN Publisher AS Pb
ON Ind.IndividualId = Pb.IndividualId;
```

Or

```
SELECT  Ind.FIRSTNAME, Ind.LASTNAME,Pb.Username,Pb.Accesslevel FROM
Individual  Ind, Publisher Pb WHERE  Ind.IndividualId (+)= Pb.IndividualId;
```

Result Table:

IndividualId	FirstName	LastName	UserName	IndividualId	AccessLevel
1	Fred	Flinstone	freddo	1	Administrator
2	Homer	Simpson	homey	2	Contributor
3	Homer	Brown	notsofamous	3	Contributor
4	Ozzy	Osbourne	sabbath	4	Contributor
NULL	NULL	NULL	NULL	10	Administrator

Cursors in SQL

- The Oracle Engine uses a work area for its internal processing in order to execute an SQL statement . This work area is private to SQL's operations and is called a Cursor.
- The data that is stored in the cursor is called the Active Data Set.
- Conceptually the size of the cursor in memory is the size required to hold the number of rows in the Active Data Set.
- For every SQL statement execution certain area in memory is allocated. This private SQL area is called **context area or cursor**. A cursor acts as a handle or pointer into the context area. A PL/SQL program controls the context area using the cursor. Cursor represents a structure in memory
- When we declare a cursor, we get a pointer variable, which does not point any thing. When the cursor is opened, memory is allocated and the cursor structure is created. The cursor variable now points the cursor. When the cursor is closed the memory allocated for the cursor is released.
- Cursors allow the programmer to retrieve data from a table and perform actions on that data one row at a time. There are two types of cursors:
  - **Implicit Cursors and**
  - **Explicit cursors**

Implicit Cursors

- For SQL queries returning single row PL/SQL declares implicit cursors. Implicit cursors are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL. Implicit cursors are easy to code, and they retrieve exactly one row. PL/SQL implicitly declares cursors for all DML statements. The most commonly raised exceptions here are NO_DATA_FOUND or TOO_MANY_ROWS.

- **Implicit cursor attributes can be used to access information about the status of the last insert, update, delete or single-row select statements.**

Example :

```
BEGIN
UPDATE emp_master SET Branch_no=&branch_no where Emp_no=&emp_no;

If SQL%FOUND then
Dbms_output.put_line('Employee successfully transferred');

ENDIF;

If SQL%NOTFOUND then
Dbms_output.put_line('employee no. does not exist');
ENDIF
END;
```

Explicit Cursors

- Explicit cursors are used in queries that return multiple rows. The set of rows fetched by a query is called active set. The size of the active set meets the search criteria in the select statement. Explicit cursor is declared in the DECLARE section of PL/SQL program.

Syntax: ***CURSOR <cursor-name> IS <select statement>***

Sample Code:

```
DECLARE
CURSOR emp_cur IS SELECT ename FROM EMP;
BEGIN
---
END;
```

Processing multiple rows is similar to file processing. For processing a file you need to open it, process records and then close.

Similarly user-defined explicit cursor needs to be opened, before reading the rows, after which it is closed. Like how file pointer marks current position in file processing, cursor marks the current position in the active set.

Opening Cursor

Syntax: OPEN <cursor-name>;

Example : OPEN emp_cur;

- When a cursor is opened the active set is determined, the rows satisfying the where clause in the select statement are added to the active set. A pointer is established and points to the first row in the active set.

- **Fetching from the cursor:** To get the next row from the cursor we need to use fetch statement.

  Syntax: FETCH <cursor-name> INTO <variables>;

  Example: FETCH emp_cur INTO ena;

  FETCH statement retrieves one row at a time. Bulk collect clause need to be used to fetch more than one row at a time.

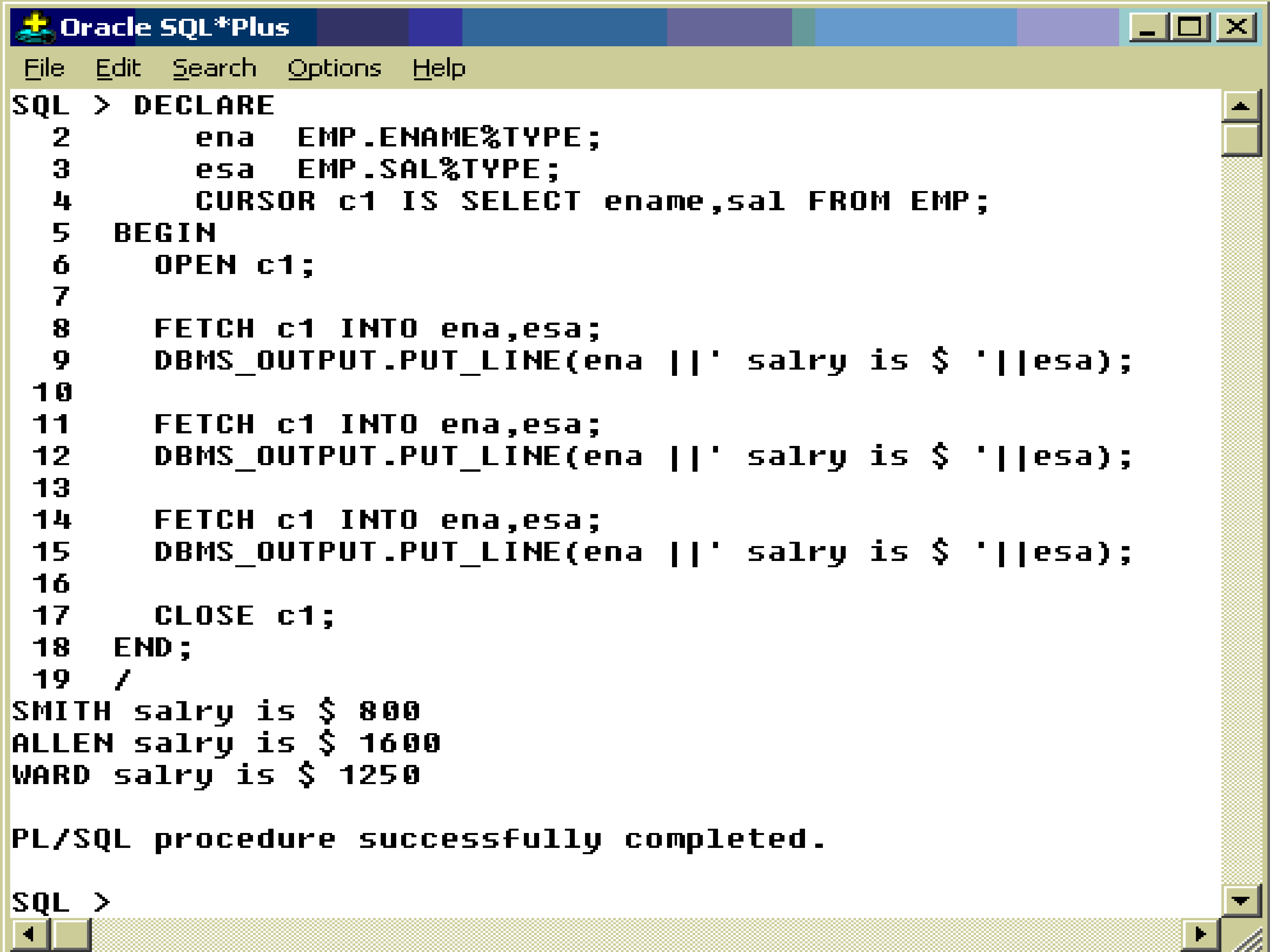
  Closing the cursor: After retrieving all the rows from active set the cursor should be closed. Resources allocated for the cursor are now freed. Once the cursor is closed the execution of fetch statement will lead to errors.

  CLOSE <cursor-name>;

Explicit Cursor Attributes

- Every cursor defined by the user has 4 attributes. When appended to the cursor name these attributes let the user access useful information about the execution of a multi-row query.The attributes are:

1. **%NOTFOUND:** It is a Boolean attribute, which evaluates to true, if the last fetch failed. i.e. when there are no rows left in the cursor to fetch.
2. **%FOUND:** Boolean variable, which evaluates to true if the last fetch, succeeded.
3. **%ROWCOUNT:** It's a numeric attribute, which returns number of rows fetched by the cursor so far.
4. **%ISOPEN:** A Boolean variable, which evaluates to true if the cursor is opened otherwise to false



```
SQL > DECLARE
  2        ena   EMP.ENAME%TYPE;
  3        esa   EMP.SAL%TYPE;
  4        CURSOR c1 IS SELECT ename,sal FROM EMP;
  5   BEGIN
  6      OPEN c1;
  7
  8      FETCH c1 INTO ena,esa;
  9      DBMS_OUTPUT.PUT_LINE(ena ||' salry is $ '||esa);
 10
 11      FETCH c1 INTO ena,esa;
 12      DBMS_OUTPUT.PUT_LINE(ena ||' salry is $ '||esa);
 13
 14      FETCH c1 INTO ena,esa;
 15      DBMS_OUTPUT.PUT_LINE(ena ||' salry is $ '||esa);
 16
 17      CLOSE c1;
 18   END;
 19   /
SMITH salry is $ 800
ALLEN salry is $ 1600
WARD salry is $ 1250

PL/SQL procedure successfully completed.

SQL >
```

PL/SQL

- **PL/SQL (Procedural Language/Structured Query Language)** is Oracle Corporation's proprietary procedural extension to the SQL database language, used in the Oracle database. Some other SQL database management systems offer similar extensions to the SQL language. PL/SQL's syntax strongly resembles that of Ada, and just like some Ada compilers of the 1980s.

- The key strength of PL/SQL is its tight integration with the Oracle database.

- PL/SQL is one of three languages embedded in the Oracle Database, the other two being SQL and Java.

- PL/SQL is development tool that not only supports SQL data manipulation but also provides facilities of conditional checking, branching and looping.

- PL/SQL sends an entire block of SQL statement to the Oracle engine all in one go.
- PL/SQL also permits dealing with errors as required , and facilitates displaying user-friendly messages, when errors are encountered.
- PL/SQL allows declaration and use of variables in blocks of code . These variables can be used to stored results of a query for later processing.
- Via PL/SQL , all sorts of calculations can be done quickly and efficiently without the use of Oracle engine. This considerably improves transaction performance.

PL/SQL offers several pre-defined packages for specific purposes. Such PL/SQL packages include:

- **DBMS_OUTPUT** - for output operations to non-database destinations
- **DBMS_JOB** - for running specific procedures/functions at a particular time (i.e. scheduling)
- **DBMS_SESSION** - provides access to SQL ALTER SESSION and SET ROLE statements, and other session information.
- **DBMS_METADATA -** for extracting meta data from the data dictionary (such as DDL statements)

and many more - Oracle Corporation customarily adds more packages and/or extends package functionality with each successive release of Oracle Database.

Basic code structure

- PL/SQL scripts, and have the following structure:

```
DECLARE TYPE
 BEGIN
     Statements
EXCEPTION
     EXCEPTION handlers
END ;
```

Example

```
DECLARE
        number1 NUMBER:=4;
        number2 NUMBER:= 17;    -- value default
        text1 VARCHAR2(12) := 'Hello world';
        text2 DATE := SYSDATE; -- current date and time
BEGIN
        DBMS_OUTPUT.PUT_LINE( number1);
        DBMS_OUTPUT.PUT_LINE( number2);
        DBMS_OUTPUT.PUT_LINE( text1);
        DBMS_OUTPUT.PUT_LINE( text2);
END;
```

Major Data Types in PL/SQL

- The symbol **:=** functions as an assignment operator to store a value in a variable.
- The major data types in PL/SQL include:
  - NUMBER
  - CHAR
  - VARCHAR2
  - DATE
  - TIMESTAMP
  - TEXT *etc.*

Functions

Functions in PL/SQL are a collection of SQL and PL/SQL statements that perform a task and should return a value to the calling environment.

CREATE OR REPLACE FUNCTION <function_name>
[(input/output variable declarations)]
 RETURN return_type <IS|AS> [declaration block]
  BEGIN
<PL/**SQL** block **WITH RETURN** statement>
[**EXCEPTION EXCEPTION** block]

 END;

Procedures

- Procedures are the same as Functions, in that they are also used to perform some task with the difference being that procedures cannot be used in a SQL statement and although they can have multiple out parameters they do not return a value. This is not always true for when an NULL function is used.

Triggers

A *trigger* is a set of actions that run automatically when a specified change operation is performed on a specified table.

The change operation can be an SQL INSERT, UPDATE, or DELETE statement, or an insert, update, or delete high level language statement in an application program.

Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

SQL triggers

The SQL CREATE TRIGGER statement provides a way for the database management system to actively control, monitor, and manage a group of tables whenever an insert, update, or delete operation is performed.

The statements specified in the SQL trigger are executed each time an SQL insert, update, or delete operation is performed.

An SQL trigger may call stored procedures or user-defined functions to perform additional processing when the trigger is executed.

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - **E.g. create trigger** *T1* **after update of** *balance* **on** *account*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as :** for inserts and updates
-

Statement level Trigger

```
CREATE OR REPLACE  TRIGGER T1 after update or
delete on EMPLOYEE
BEGIN
if updating then
insert into audit_table values (‘Value1’,’UPDATE’,sysdate);
Endif;

if deleting then
insert into audit_table values (‘Value1’,’DELETE’,sysdate);
Endif;
END;
```

Audit_table	
Name	**Type**
TABLE_NAME	VARCHAR2(10)
DML_OPERATION	VARCHAR2(6)
DATE_OF_DML	DATE

TYPES OF TRIGGRES

Row Trigger : A row trigger is fired each time in the table is affected by triggering statement.

E.g. UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.

Statement Trigger:

It is fired once on behalf of the triggering statement , independent of the number of rows the triggering statement affects.

Before Vs After Triggers

When defining a trigger it is necessary to specify the trigger timing, i.e. specifying when the trigger action is to be executed in relation to the triggering statement.

TYPES OF TRIGGRES

Before Trigger : it executes the trigger action before the triggering statement.

After Trigger: it executes the trigger action after the triggering statement is executed

Syntax : CREATE OR REPLACE TRIGGER <trigger name>

```
    <BEFORE, AFTER>
{DELETE, INSERT, UPDATE [of Columnname]}
On <tablename>
Declare
:
Begin
<PL/SQL
End;
```

Creating Trigger

CREATE TRIGGER T1

AFTER UPDATE or DELETE on Cust_Master

FOR EACH ROW

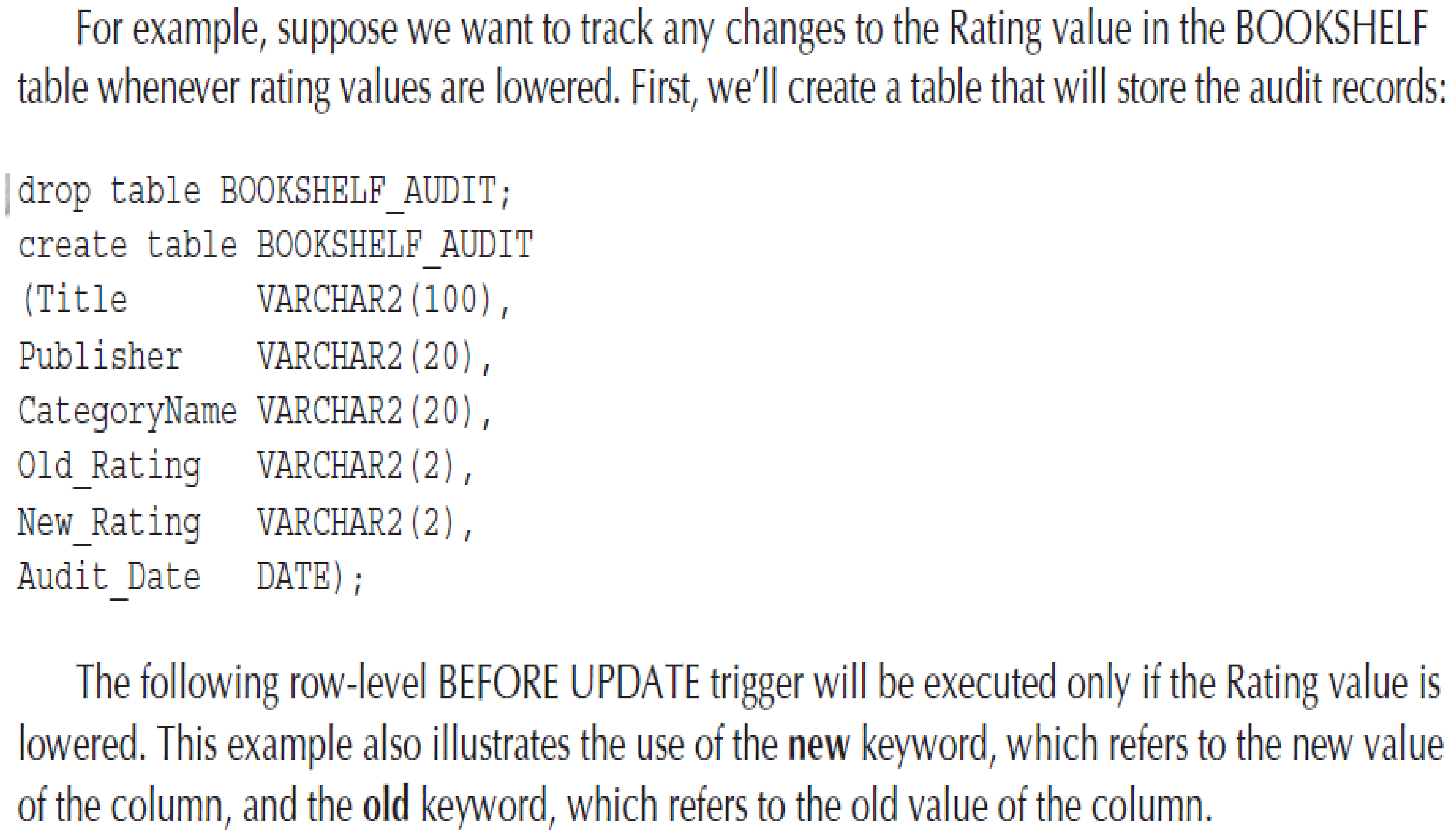
DECLARE

BEGIN

PL/SQL statement

END;

Example:



For example, suppose we want to track any changes to the Rating value in the BOOKSHELF table whenever rating values are lowered. First, we'll create a table that will store the audit records:

```
drop table BOOKSHELF_AUDIT;
create table BOOKSHELF_AUDIT
(Title        VARCHAR2(100),
Publisher     VARCHAR2(20),
CategoryName VARCHAR2(20),
Old_Rating    VARCHAR2(2),
New_Rating    VARCHAR2(2),
Audit_Date    DATE);
```

The following row-level BEFORE UPDATE trigger will be executed only if the Rating value is lowered. This example also illustrates the use of the **new** keyword, which refers to the new value of the column, and the **old** keyword, which refers to the old value of the column.

Sql> CREATE TABLE BOOKSHELF (TITLE,PUBLISHER,CATEGORYNAME,RATING)

```
create or replace trigger BOOKSHELF_BEF_UPD_ROW
before update on BOOKSHELF
for each row
when (new.Rating < old.Rating)
begin
  insert into BOOKSHELF_AUDIT
    (Title, Publisher, CategoryName,
     Old_Rating, New_Rating, Audit_Date)
  values
    (:old.Title, :old.Publisher, :old.CategoryName,
     :old.Rating, :new.Rating, Sysdate);
end;
/
```

```
Create or replace trigger tt22 before update on bookshelf for each row when (new.rate
< old.rate)
Begin
Insert into bookshelf_aud values ( :old.title,:old.rate,:new.rate,sysdate);
End;
```

Trigger Code

```
create trigger T1 after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin
  insert into borrower
        (select customer-name, account-number
        from depositor
        where nrow.account-number =
                depositor.account-number);
  insert into loan values
        (nrow.account-number, nrow.branch-name,nrow.balance);
  update account set balance = 0
  where account.account-number = nrow.account-number
end
```

Security and User Authorization in SQL

Threats to the database

Threats to the database include:

•**Unauthorised modification:** Changing data values for reasons of sabotage, crime or ignorance which may be enabled by inadequate security mechanisms, or sharing of passwords or password guessing etc.

•**Unauthorised disclosure:** When information that should not have been disclosed has been disclosed. A general issue of crucial importance, which can be accidental or deliberate.

- **Loss of availability:**

Some times called denial of service**.** When the database is not available it incurs a loss .So any threat that gives rise to time offline, even to check whether something has occurred, is to be avoided.

Categories of specific regulatory threats to database systems

- **Commercial sensitivity:**
Most financial losses through fraud arise from employees. Access controls provide both protection against criminal acts and evidence of attempts (successful or otherwise) to carry out acts detrimental to the organisation, whether fraud, extraction of sensitive data or loss of availability.
- **Personal privacy and data protection:** Internationally, personal data is normally subject to legislative controls. Personal data is data about an identifiable individual. Often the individual has to be alive but the method of identification is not prescribed. So a postal code for a home may in some cases identify an individual, if only one person is living at an address with the postal code. Such data needs careful handling and control.

Categories of specific regulatory threats to database systems

• **Computer misuse:**

There is also generally legislation on the misuse of computers. Misuse includes the violation of access controls and attempts to cause damage by changing the database state or introducing worms and viruses to interfere with proper operation. These offences are often extraditable. ***So an unauthorised access in India using computers in France to access databases in Germany which refer to databases in America could lead to extradition to France or Germany or the USA.***

•**Audit requirements:**

These are operational constraints built around the need to know who did what, who tried to do what, and where and when everything happened. They involve the detection of events (including CONNECT and GRANT transactions), providing evidence for detection, assurance as well as either defence or prosecution. There are issues related to computer-generated evidence.

Authentication and Authorization

Access to IT resources generally requires a log-in process that is trusted to be secure. Most of what follows is directly about Relational client-server systems. Other system models differ to a greater or lesser extent, though the underlying principles remain true.

Authentication

- ✓ The client has to establish the identity of the server and the server has to establish the identity of the client. This is done often by means of shared secrets (either a password/user-id combination, or shared biographic and/or biometric data).
- ✓ It can also be achieved by a system of higher authority which has previously established authentication.
- ✓ In client-server systems where data (not necessarily the database) is distributed, the authentication may be acceptable from a peer system. Note that authentication may be transmissible from system to system. The result, as far as the DBMS is concerned, is an authorization-identifier.

Authentication does not give any privileges for particular tasks. It only establishes that the DBMS trusts that the user is who he claimed to be and that the user trusts that the DBMS is also the intended system. Authentication is a prerequisite for authorization.

Authorization

Authorization relates to the permissions granted to an authorized user to carry out particular transactions, and hence to change the state of the database(write item transactions) and/or receive data from the database (read-item transactions). The result of authorization, which needs to be on a transactional basis, is a vector: Authorization.

Authorization

Forms of authorization on (parts of) the database:

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data

Security Specification in SQL

- The grant statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> to <user list>

- <user list> is:
  - a user-id
  - *public*, which allows all valid users the privilege granted
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

grant select on *branch* **to** *U_1, U_2, U_3*

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **references**: ability to declare foreign keys when creating relations.
- **all privileges**: used as a short form for all the allowable privileges

Privilege To Grant Privileges

- **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.
  - Example:

    grant select on ***branch*** **to** U_1 **with grant option**

    gives U_1 the **select** privileges on branch and allows U_1 to grant this privilege to others

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke<privilege list>

  on <relation name or view name> **from** <user list> [**restrict**|**cascade**]

- Example:

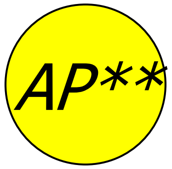
  revoke select on *branch* **from** U_1, U_2, U_3 **cascade**

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**.
- We can prevent cascading by specifying **restrict**:

  revoke select on *branch* **from** U_1, U_2, U_3 **restrict**

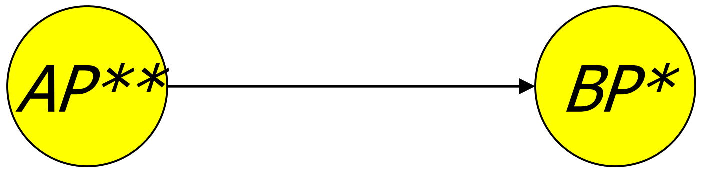
  With **restrict**, the **revoke** command fails if cascading revokes are required.

Example: Grant Diagram



A owns the
object on
which P is
a privilege

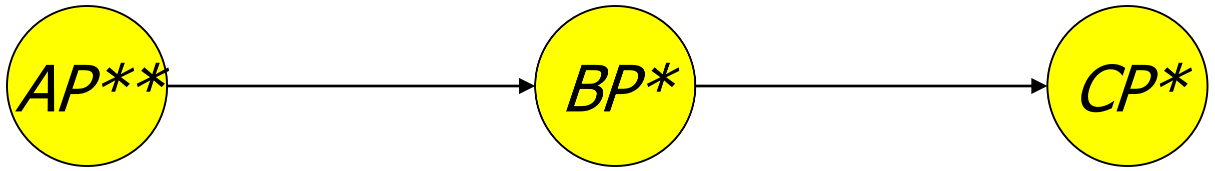
Example: Grant Diagram



A owns the object on which P is a privilege

A:
GRANT P
TO B WITH
GRANT OPTION

Example: Grant Diagram

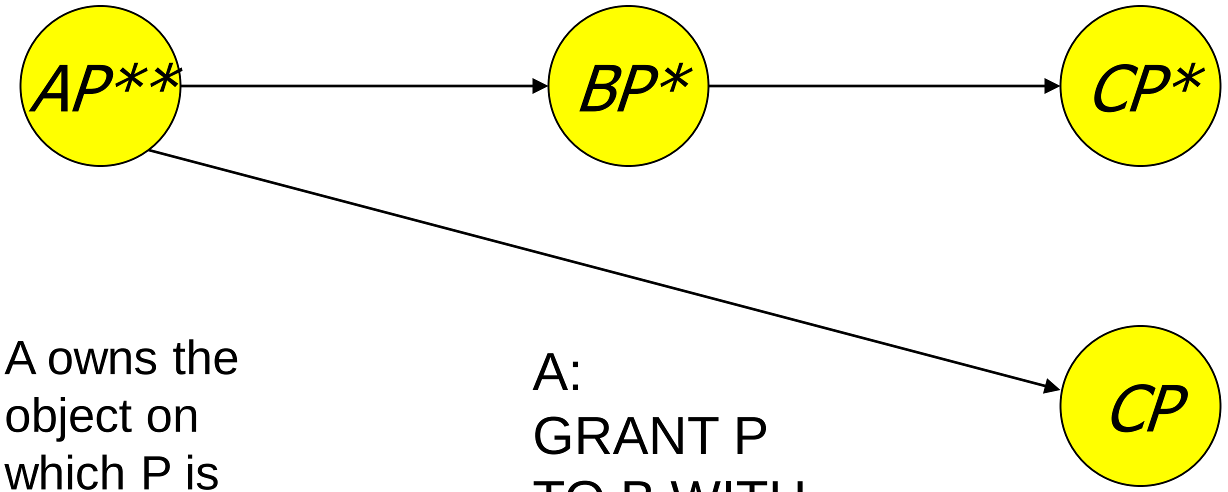


B:
GRANT P
TO C WITH
GRANT OPTION

A owns the
object on
which P is
a privilege

A:
GRANT P
TO B WITH
GRANT OPTION

Example: Grant Diagram



B:
GRANT P
TO C WITH
GRANT OPTION

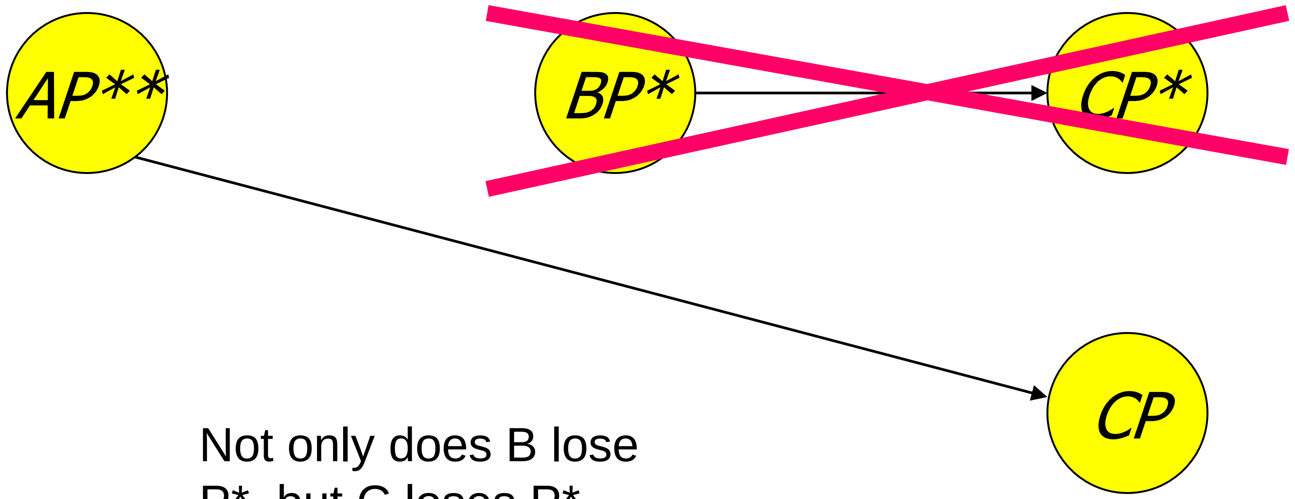
A owns the
object on
which P is
a privilege

A:
GRANT P
TO B WITH
GRANT OPTION

A:
GRANT P
TO C

Example: Grant Diagram

A executes
REVOKE P FROM B CASCADE;

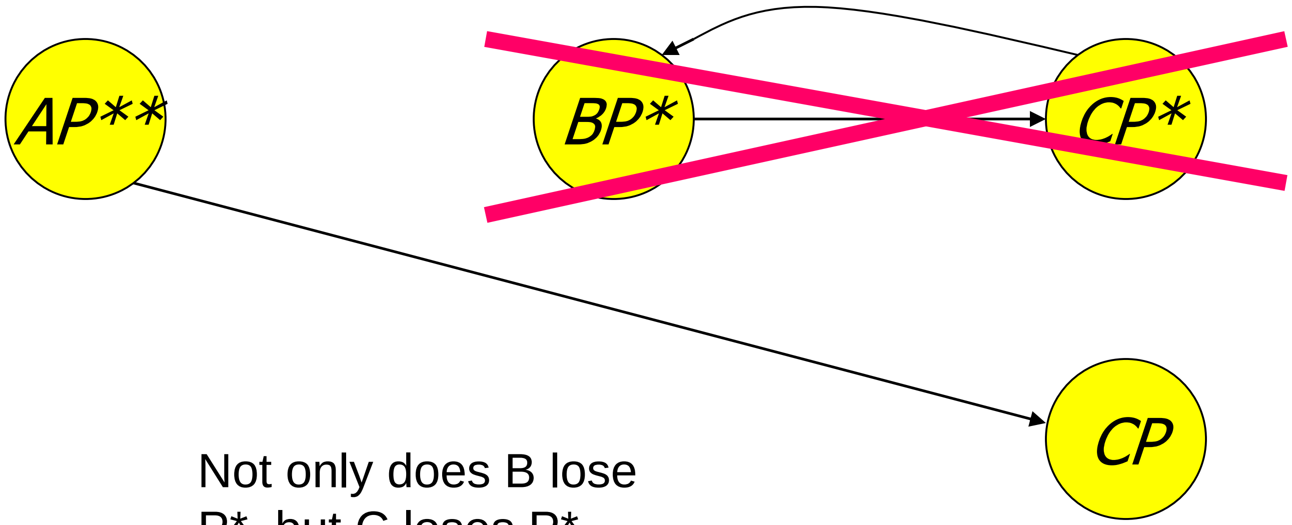


Not only does B lose
P*, but C loses P*.
Delete BP* and CP*.

However, C still
has P without grant
option because of
the direct grant.

Example: Grant Diagram

A executes
REVOKE P FROM B CASCADE;



Even had C passed P to B, both nodes are still cut off.

Not only does B lose P*, but C loses P*. Delete BP* and CP*.

However, C still has P without grant option because of the direct grant.

Clusters

Clustering is an important concept for improving Oracle performance. Whenever the database is accessed, any reduction in input/output always helps in improving its overall performance.

The concept of cluster is where member records are stored physically near parent records. For Oracle clusters can be used to define common, one to many access paths, and the member rows can be stored on the same database block as their owner row.

Clusters can be used to store data form different tables in the same physical data blocks, they are appropriate to use if the records from those tables are frequently queried together. By storing them in the same data blocks, the number of database block reads needed to fullfill such queries decreases, there by improving performance.

Packages

- Packages are groups of conceptually linked Functions, Procedures,Variable,Constants & Cursors etc.

- The use of packages promotes re-use of code. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary.

- The specification (spec for short) is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use.

- The body fully defines cursors and subprograms, and so implements the spec.