Chapter 14

Advanced SQL

Transparencies

Chapter - Objectives

- How to create and delete views using SQL.
- How the DBMS performs operations on views.
- Under what conditions views are updatable.
- Advantages and disadvantages of views.
- Purpose of integrity enhancement feature of SQL.
- How to define integrity constraints using SQL.
- How to use the integrity enhancement feature in the CREATE and ALTER TABLE statements.
- How the ISO transaction model works.

Chapter - Objectives

- How to use the GRANT and REVOKE statements as a level of security.
- SQL statements can be embedded in high-level programming languages.
- Difference between static and dynamic embedded SQL.
- How to write programs that use embedded SQL.
- How to use the Open Database Connectivity (ODBC) de facto standard

Views

View

Dynamic result of one or more relational operations operating on the base relations to produce another relation.

- Virtual relation that does not actually exist in the database but is produced upon request, at time of request.

Views

- **Contents of a view are defined as a query on one or more base relations.**
- **Any operations on view are automatically translated into operations on relations from which it is derived.**

SQL - CREATE VIEW

CREATE VIEW view_name [(column_name [,...])]
AS subselect
[WITH [CASCADED | LOCAL] CHECK OPTION]

- **Can assign a name to each column in view.**
- **If list of column names is specified, it must have same number of items as number of columns produced by *subselect*. If omitted, each column takes name of corresponding column in *subselect*.**

SQL - CREATE VIEW

- List must be specified if there is any ambiguity in a column name.
- The *subselect* is known as the defining query.
- WITH CHECK OPTION ensures that if a row fails to satisfy WHERE clause of defining query, it is not added to underlying base table.
- Need SELECT privilege on all of tables referenced in the subselect and USAGE privilege on any domains used in referenced columns.

Example 14.1 - Create Horizontal View

Create a view so that the manager at branch B3 can only see details for staff who work in his or her office.

```
CREATE VIEW manager3_staff
AS    SELECT *
      FROM staff
      WHERE bno = 'B3';
```

Example 14.1 - Create Horizontal View

Creating view Manager3_Staff with same column names as Staff:

Table 14.1 Data for view Manager3_Staff.

sno	*fname*	*lname*	*address*	*tel_no*	*position*	*sex*	*dob*	*salary*	*nin*	*bno*
SG37	Ann	Beech	81 George St, Glasgow PA1 2JR	0141-848-3345	Snr Asst	F	10-Nov-60	12000.00	WL432514C	B3
SG14	David	Ford	63 Ashby St, Partick, Glasgow G11	0141-339-2177	Deputy	M	24-Mar-58	18000.00	WL220658D	B3
SG5	Susan	Brand	5 Gt Western Rd, Glasgow G12	0141-334-2001	Manager	F	3-Jun-40	24000.00	WK588932E	B3

(3 rows)

Example 14.2 - Create Vertical View

Create view of staff details at branch B3 excluding salaries.

```
CREATE VIEW staff3
AS      SELECT sno, fname, lname, address,
                    tel_no, position, sex
  FROM staff
  WHERE bno = 'B3';
```

Example 14.2 - Create Vertical View

Or:

```
CREATE VIEW staff3
AS SELECT sno, fname, lname,address,
                tel_no, position, sex
    FROM manager3_staff;
```


Example 14.2 - Create Vertical View

- **Creating view Staff3 with same columns as Staff, excluding Salary, DOB, NIN, and Bno:**

Table 14.2 Data for view Staff3.

sno	*fname*	*lname*	*address*	*tel_no*	*position*	*sex*
SG37	Ann	Beech	81 George St, Glasgow PA1 2JR	0141-848-3345	Snr Asst	F
SG14	David	Ford	63 Ashby St, Partick, Glasgow G11	0141-339-2177	Deputy	M
SG5	Susan	Brand	5 Gt Western Rd, Glasgow G12	0141-334-2001	Manager	F

(3 rows)

Example 14.3 - Grouped and Joined Views

Create view of staff who manage properties for rent, including branch number they work at, staff number, and number of properties they manage.

```
CREATE VIEW staff_prop_cnt (branch_no, staff_no, cnt)
 AS SELECT s.bno, s.sno, COUNT(*)
    FROM staff s, property_for_rent p
    WHERE s.sno = p.sno
    GROUP BY s.bno, s.sno;
```


Example 14.3 - Grouped and Joined Views

Table 14.3 Data for view Staff_Prop_Cnt.

branch_no	*staff_no*	*cnt*
B3	SG14	2
B3	SG37	2
B5	SL41	1
B7	SA9	1

(4 rows)

SQL - DROP VIEW

DROP VIEW view_name
[RESTRICT | CASCADE]

- **Causes definition of view to be deleted from the database.**
- **For example:**

DROP VIEW manager3_staff;

SQL - DROP VIEW

- **With CASCADE, all related dependent objects are deleted; i.e. any views defined on view being dropped.**
- **With RESTRICT (default), if any other objects depend for their existence on continued existence of view being dropped, command is rejected.**

Restrictions on Views

SQL-92 imposes several restrictions on creation and use of views.

(a) If column in view is based on an aggregate function:

- **Column may appear only in SELECT and ORDER BY clauses of queries that access view.**
- **Column may not be used in WHERE nor be an argument to an aggregate function in any query based on view.**

Restrictions on Views

- **For example, following query would fail:**

```
SELECT COUNT(cnt)
FROM staff_prop_cnt;
```

- **Similarly, following query would also fail:**

```
SELECT *
FROM staff_prop_cnt
WHERE cnt > 2;
```

Restrictions on Views

(b) Grouped view may never be joined with a base table or a view.

- **For example, Staff_Prop_Cnt view is a grouped view, so any attempt to join this view with another table or view fails.**

View Resolution

Count number of properties managed by each member at branch B3.

```
SELECT staff_no, cnt
FROM staff_prop_cnt
WHERE branch_no = 'B3'
ORDER BY staff_no;
```

View Resolution

(a) View column names in SELECT list are translated into their corresponding column names in the defining query:

SELECT s.sno, COUNT(*)

(b) View names in FROM are replaced with corresponding FROM lists of defining query:

FROM staff s, property_for_rent p

View Resolution

(c) WHERE from user query is combined with WHERE of defining query using AND:

WHERE s.sno = p.sno AND bno = 'B3'

(d) GROUP BY and HAVING clauses copied from defining query:

GROUP BY s.sno, s.bno

(e) ORDER BY copied from query with view column name translated into defining query column name

ORDER BY s.sno

View Resolution

(f) Final merged query is now executed to produce the result:

```
SELECT s.sno, COUNT(*)
FROM staff s, property_for_rent p
WHERE s.sno = p.sno AND bno = 'B3'
GROUP BY s.sno, s.bno
ORDER BY s.sno;
```


View Updatability

- **All updates to base table reflected in all views that encompass base table.**
- **Similarly, may expect that if view is updated then base table(s) will reflect change.**

View Updatability

- **However, consider view Staff_Prop_Cnt:**

```
CREATE VIEW staff_prop_cnt (branch_no,
                            staff_no, cnt)
AS      SELECT s.bno, s.sno, COUNT(*)
 FROM staff s, property_for_rent p
 WHERE s.sno = p.sno
 GROUP BY s.bno, s.sno;
```


View Updatability

- **giving table:**

Table 14.5 The view Staff_Prop_Cnt.

branch_no	*staff_no*	*cnt*
B3	SG14	2
B3	SG37	2
B5	SL41	1
B7	SA9	1

(4 rows)

View Updatability

- If tried to insert record showing that at branch B3, SG5 manages 2 properties:

```
INSERT INTO staff_prop_cnt
VALUES ('B3', 'SG5', 2);
```

- Have to insert 2 records into Property_for_Rent showing which properties SG5 manages. However, do not know which properties they are; i.e. do not know primary keys!

View Updatability

- If change definition of view and replace count with actual property numbers:

```
CREATE VIEW staff_prop_list (branch_no,
                             staff_no, property_no)
AS SELECT s.bno, s.sno, p.pno
    FROM staff s, property_for_rent p
    WHERE s.sno = p.sno;
```

View Updatability

- and try to insert the record:

```
INSERT INTO staff_prop_list
VALUES ('B3', 'SG5', 'PG19');
```

- Still problem, because in Property_for_Rent all columns except Area/Pcode/Sno are not allowed nulls.
- However, have no way of giving remaining non-null columns values.

View Updatability

- SQL-92 specifies the views that must be updatable in system that conforms to standard.
- Definition given is that a view is updatable iff:
  - DISTINCT is not specified.
  - Every element in SELECT list of defining query is a column name and no column appears more than once.
  - FROM clause specifies only one table, excluding any views based on a join, union, intersection or difference.

View Updatability

- WHERE clause does not include any nested SELECTs that reference the table in FROM clause.
- There is no GROUP BY or HAVING clause in the defining query.

◆ Also, every row added through view must not violate integrity constraints of base table.

Updatable View

For view to be updatable, DBMS must be able to trace any row or column back to its row or column in the source table.

WITH CHECK OPTION

- **Rows exist in a view because they satisfy WHERE condition of defining query.**
- **If a row changes and no longer satisfies condition, it disappears from the view.**
- **New rows appear within view when insert/update on view cause them to satisfy WHERE condition.**
- **Rows that enter or leave a view are called *migrating rows*.**
- **WITH CHECK OPTION prohibits a row migrating out of the view.**

WITH CHECK OPTION

- **LOCAL/CASCADED apply to view hierarchies.**
- **With LOCAL, any row insert/update on view and any view directly or indirectly defined on this view must not cause row to disappear from view unless row also disappears from derived view/table.**
- **With CASCADED (default), any row insert/update on this view and on any view directly or indirectly defined on this view must not cause row to disappear from the view.**

Example 14.4 - WITH CHECK OPTION

```
CREATE VIEW manager3_staff
AS      SELECT *
        FROM staff
        WHERE bno = 'B3'
WITH CHECK OPTION;
```


Example 14.4 - WITH CHECK OPTION

- Cannot update ranch number of row B3 to B5 as this would cause row to migrate from view.
- Specification of WITH CHECK OPTION would prevent following insert through row:

```
INSERT INTO manager3_staff
VALUES('SL15', 'Mary' , 'Black',
       '2 Hillcrest, London, NW2',
       '0181-554-3426', 'Assistant', 'F',
       '21-Jun-67', 8000, 'WM787850T', 'B2');
```

Example 14.4 - WITH CHECK OPTION

- If Manager3_Staff is defined not on Staff directly but on another view of Staff:

```
CREATE VIEW low_salary
 AS SELECT * FROM Staff WHERE salary > 9000;
CREATE VIEW high_salary
 AS SELECT * FROM low_salary
     WHERE salary > 10000
 WITH LOCAL CHECK OPTION;
CREATE VIEW manager3_staff
 AS SELECT * FROM high_salary WHERE bno = 'B3';
```


Example 14.4 - WITH CHECK OPTION

```
UPDATE manager3_staff
SET salary = 9500
WHERE sno = 'SG37';
```

- Update would fail: although update would cause row to disappear from High_Salary, row would not disappear from Low_Salary.
- However, if update tried to set salary to 8000, update would succeed as row would no longer be part of Low_Salary.

Example 14.4 - WITH CHECK OPTION

- If High_Salary had specified WITH CASCADED CHECK OPTION, setting salary to 9500 or 8000 would be rejected because row would disappear from High_Salary.
- To prevent anomalies like this, each view should be created using WITH CASCADED CHECK OPTION.

Advantages of Views

- Data Independence
- Currency
- Security
- Reduced Complexity
- Convenience
- Customization
- Data Integrity

Disadvantages of Views

- Update Restriction
- Structure Restriction
- Performance

Integrity Enhancement Feature

- Consider five types of integrity constraints:
  - Required data.
  - Domain constraints.
  - Entity integrity.
  - Referential integrity.
  - Enterprise constraints.

Integrity Enhancement Feature

Required Data

position VARCHAR(10) NOT NULL

Domain Constraints

(a) CHECK

```
sex   CHAR      NOT NULL
      CHECK (sex IN ('M', 'F'))
```


Integrity Enhancement Feature

(b) CREATE DOMAIN

CREATE DOMAIN domain_name [AS] data_type
[DEFAULT default_option]
[CHECK (search_condition)]

For example:

```
CREATE DOMAIN sex_type AS CHAR
      CHECK (VALUE IN ('M', 'F'));
sex   SEX_TYPE      NOT NULL
```

Integrity Enhancement Feature

- ***search_condition*** **can involve a table lookup:**

```
CREATE DOMAIN branch_no AS VCHAR(3)
CHECK (VALUE IN (SELECT bno
                 FROM branch));
```

- **Domains can be removed using DROP DOMAIN:**

```
DROP DOMAIN domain_name
 [RESTRICT | CASCADE]
```


IEF - Entity Integrity

- **Primary key of a table must contain a unique, non-null value for each row.**
- **SQL-92 supports FOREIGN KEY clause in CREATE and ALTER TABLE statements:**

```
PRIMARY KEY(sno)
PRIMARY KEY(rno, pno)
```

- **PRIMARY KEY clause can be specified only once per table. Can still ensure uniqueness for alternate keys using UNIQUE.**

IEF - Referential Integrity

- FK is column or set of columns that links each row in child table containing foreign FK row of parent table containing matching PK.
- Referential integrity means that, if FK contains a value, that value must refer to existing row in parent table.
- SQL-92 supports definition of FKs with FOREIGN KEY clause in CREATE and ALTER TABLE:

  FOREIGN KEY(bno) REFERENCES branch

IEF - Referential Integrity

- Any INSERT/UPDATE that attempts to create FK value in child table without matching candidate key value in parent is rejected.
- Action taken that attempts to update/delete a candidate key value in parent table with matching rows in child is dependent on referential action specified using ON UPDATE/ and ON DELETE subclauses:
  - CASCADE - SET NULL,
  - SET DEFAULT - NO ACTION.

IEF - Referential Integrity

CASCADE: Delete row from parent and delete matching rows in child, and so on in cascading manner.

SET NULL: Delete row from parent and set FK column(s) in child to NULL. Only valid if FK columns are NOT NULL.

SET DEFAULT: Delete row from parent and set each component of FK in child to specified default. Only valid if DEFAULT specified for FK columns

NO ACTION: Reject delete from parent. Default.

IEF - Referential Integrity

FOREIGN KEY (sno) REFERENCES staff ON DELETE SET NULL

FOREIGN KEY (ono) REFERENCES owner ON UPDATE CASCADE

IEF - Enterprise Constraints

- **Could use CHECK/UNIQUE in CREATE and ALTER TABLE.**
- **Also have:**

  CREATE ASSERTION assertion_name
  CHECK (search_condition)

- **which is very similar to the CHECK clause.**

IEF - Enterprise Constraints

```
CREATE ASSERTION staff_not_handling_too_much
CHECK (NOT EXISTS (SELECT sno
                   FROM property_for_rent
                   GROUP BY sno
                   HAVING COUNT(*) > 10))
```

Example 14.5 - CREATE TABLE

```
CREATE DOMAIN owner_number AS VARCHAR(5)
  CHECK (VALUE IN (SELECT ono FROM owner));
CREATE DOMAIN prop_number AS VARCHAR(5);
CREATE DOMAIN property_rooms AS SMALLINT;
      CHECK(VALUE BETWEEN 1 AND 15);
CREATE DOMAIN property_rent AS DECIMAL(6,2)
      CHECK(VALUE BETWEEN 0 AND 9999);
```


Example 14.5 - CREATE TABLE

```
CREATE TABLE property_for_rent (
   pno        PROP_NUMBER   NOT NULL,
   rooms      PROPERTY_ROOMS     NOT NULL
                                 DEFAULT 4,
   rent       PROPERTY_RENT      NOT NULL,
   ono        OWNER_NUMBER       NOT NULL,
   bno        BRANCH_NUMBER      NOT NULL,
   PRIMARY KEY (pno),
   FOREIGN KEY (ono) REFERENCES owner ON
   DELETE NO ACTION ON UPDATE CASCADE);
```

ALTER TABLE

- Add a new column to a table.
- Drop a column from a table.
- Add a new table constraint.
- Drop a table constraint.
- Set a default for a column.
- Drop a default for a column.

Example 14.6 - ALTER TABLE

Change Staff table by removing default of 'Assistant' for Position column and setting default for Sex column to female ('F').

```
ALTER TABLE staff
        ALTER position DROP DEFAULT;
ALTER TABLE staff
        ALTER sex SET DEFAULT 'F';
```

Example 14.6 - ALTER TABLE

Removing constraint that staff not allowed to handle more than 10 properties at a time from Property_for_Rent.

```
ALTER TABLE property_for_rent
    DROP CONSTRAINT staff_not_handling_too_much;
```


Example 14.6 - ALTER TABLE

Add new column to Renter representing preferred area for accommodation.

```
ALTER TABLE renter
    ADD pref_area VARCHAR(15);
```

Transactions

- SQL-92 defines transaction model based on COMMIT and ROLLBACK.
- Transaction is logical unit of work with one or more SQL statements guaranteed to be atomic with respect to recovery.
- An SQL transaction automatically begins with a *transaction-initiating* SQL statement (e.g., SELECT, INSERT). Changes made by transaction are not visible to other concurrently executing transactions until transaction completes.

Transactions

- Transaction can complete in one of four ways:
  - COMMIT ends transaction successfully, making changes permanent.
  - ROLLBACK aborts transaction, backing out any changes made by transaction.
  - For programmatic SQL, successful program termination ends final transaction successfully, even if a COMMIT has not been executed.
  - For programmatic SQL, abnormal program end aborts transaction.

Transactions

- New transaction starts with next transaction-initiating statement.
- SQL transactions cannot be nested.
- SET TRANSACTION configures transaction:

```
SET TRANSACTION
  [READ ONLY | READ WRITE] |
  [ISOLATION LEVEL READ UNCOMMITTED |
  READ COMMITTED|REPEATABLE READ
  |SERIALIZABLE ]
```


Immediate and Deferred Integrity Constraints

- Do not always want constraints to be checked immediately, but instead at transaction commit.
- Constraint may be defined as INITIALLY IMMEDIATE or INITIALLY DEFERRED, indicating mode constraint assumes at start of each transaction.
- In former case, also possible to specify whether mode can be changed subsequently using qualifier [NOT] DEFERRABLE.
- Default mode is INITIALLY IMMEDIATE.

Immediate and Deferred Integrity Constraints

- **SET CONSTRAINTS statement used to set mode for specified constraints for current transaction:**

SET CONSTRAINTS
{ALL | constraint_name [, . . .]}
{DEFERRED ¦ IMMEDIATE}

Access Control - Authorization Identifiers and Ownership

- **Authorization identifier is normal SQL identifier used to establish identity of a user. Usually, has an associated password.**
- **Used to determine which objects user may reference and what operations may be performed on those objects.**
- **Each object created in SQL has an owner, as defined in AUTHORIZATION clause of schema to which the object belongs.**
- **Owner is only person who may know about it.**

Privileges

- Actions user permitted to carry out on given base table or view:

SELECT Retrieve data from a table.

INSERT Insert new rows into a table.

UPDATE Modify rows of data in a table.

DELETE Delete rows of data from a table.

REFERENCES Reference columns of named table in integrity constraints.

USAGE Use domains, collations, character sets, and translations.

Privileges

- Can restrict INSERT/UPDATE/REFERENCES to named columns.
- Owner of table must grant other users the necessary privileges using GRANT statement.
- To create view, user must have SELECT privilege on all tables that make up view and REFERENCES privilege on the named columns.

GRANT

GRANT {privilege_list | ALL PRIVILEGES}
ON object_name
TO {authorization_id_list | PUBLIC}
[WITH GRANT OPTION]

- ***privilege_list*** **consists of one or more of the above privileges separated by commas.**
- **ALL PRIVILEGES grants all privileges to a user.**

GRANT

- **PUBLIC allows access to be granted to all present and future authorized users.**
- ***object_name*** **can be a base table, view, domain, character set, collation or translation.**
- **WITH GRANT OPTION allows privileges to be passed on.**

Example 14.7 - GRANT All Privileges

Give Manager full privileges to Staff table.

```
GRANT ALL PRIVILEGES
ON staff
TO manager WITH GRANT OPTION;
```


Example 14.8 - GRANT Specific Privileges

Give Admin SELECT and UPDATE on column Salary of Staff.

```
GRANT SELECT, UPDATE (salary)
ON staff  TO admin;
```

Give users Personnel and Deputy SELECT on Staff table.

```
GRANT SELECT
ON staff TO personnel, deputy;
```

Give all users SELECT on Branch table.

```
GRANT SELECT
ON branch TO PUBLIC;
```

REVOKE

- **REVOKE takes away privileges granted with GRANT.**

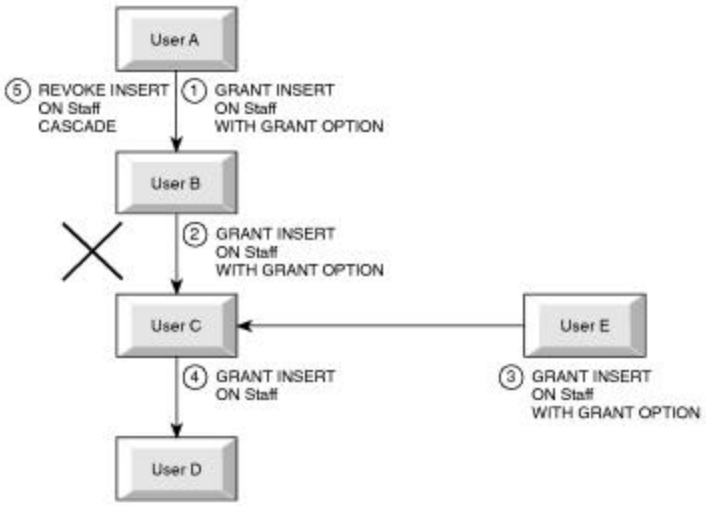
```
REVOKE [GRANT OPTION FOR]
  {privilege_list | ALL PRIVILEGES}
ON object_name
FROM {authorization_id_list | PUBLIC}
   [RESTRICT | CASCADE]
```

- **ALL PRIVILEGES refers to all privileges granted to a user by user revoking privileges.**

REVOKE

- **GRANT OPTION FOR allows privileges passed on via WITH GRANT OPTION of GRANT to be revoked separately from the privileges themselves.**
- **REVOKE fails if it results in an abandoned object, such as a view, unless the CASCADE keyword has been specified.**
- **Privileges granted to this user by other users are not affected.**

REVOKE



Example 14.11 - REVOKE Specific Privileges from PUBLIC

Revoke privilege SELECT on Branch table from all users.

```
REVOKE SELECT
ON branch FROM PUBLIC;
```

Revoke all privileges given to Deputy on Staff table.

```
REVOKE ALL PRIVILEGES
ON staff FROM deputy;
```

Embedded SQL

- **SQL can be *embedded* in high-level procedural language.**
- **In many cases, language is identical, although SELECT statement differs.**
- **Two types of programmatic SQL:**
  - ***Embedded SQL statements*.**
    - **SQL-92 supports Ada, C, COBOL, FORTRAN, MUMPS, Pascal, and PL/1.**
  - ***Application program interface (API)*.**

Example 14.13 - CREATE TABLE

```
EXEC SQL CREATE TABLE viewing     (
        pno varchar(5) not null,
        rno varchar(5) not null,
        date date not null,
        comment varchar(40));
if (sqlca.sqlcode >= 0)
   printf("Creation successful\n");
```

Embedded SQL

- Embedded SQL starts with identifier, usually EXEC SQL ['@SQL(' in MUMPS].
- Ends with terminator dependent on host language:
  - Ada, 'C', and PL/1: terminator is semicolon (;)
  - COBOL: terminator is END-EXEC
  - Fortran: ends when no more continuation lines.
- Embedded SQL can appear anywhere an executable host language statement can appear.

SQL Communications Area (SQLCA)

- Used to report runtime errors to the application program.
- Most important part is SQLCODE variable:
  - 0 - statement executed successfully.
  - < 0 - an error occurred.
  - > 0 - statement executed successfully, but an exception occurred, such as no more rows returned by SELECT.

SQLCA for Ingres

```
/* Program to create the VIEWING table */
#include <stdio.h>
#include <stdlib.h>
EXEC SQL INCLUDE sqlca;

main ( )
{
/* Connect to database */
EXEC SQL CONNECT 'estatedb';
if (sqlca.sqlcode < 0) exit(–1);

/* Display message for user and create the table */
printf("Creating VIEWING table\n");
EXEC SQL CREATE TABLE viewing (pno varchar(5) not null,
                               rno varchar (5) not null,
                               date date not null,
                               comment varchar(40));
if (sqlca.sqlcode > = 0)                    /* Check success */
     printf("Creation successful\n");
else
     printf("Creation unsuccessful\n");

/* Commit the transaction */
EXEC SQL COMMIT;

/* Finally, disconnect from the database */
EXEC SQL DISCONNECT;
}
```


WHENEVER Statement

- **Every embedded SQL statement can potentially generate an error.**
- **WHENEVER is directive to precompiler to generate code to handle errors after every SQL statement:**

EXEC SQL WHENEVER

<condition> <action>

WHENEVER Statement

- ***condition*** **can be:**
  - **SQLERROR - generate code to handle errors (SQLCODE < 0).**
  - **SQLWARNING - generate code to handle warnings.**
  - **NOT FOUND - generate code to handle specific warning that a retrieval operation has found no more records (SQLCODE = 100).**

WHENEVER Statement

- ***action*** **can be:**
  - **CONTINUE - ignore condition and proceed to next statement.**
  - **GOTO *label* or GO TO label - transfer control to specified *label*.**

WHENEVER Statement

```
EXEC SQL WHENEVER SQLERROR GOTO error1;
EXEC SQL INSERT INTO viewing VALUES ('CR76',
    'PA14', DATE'1995-05-12', 'Not enough space');
```

- would be converted to:

```
EXEC SQL INSERT INTO viewing VALUES ('CR76',
    'PA14', DATE'1995-05-12', 'Not enough space');
if (sqlca.sqlcode < 0) goto error1;
```


Host Language Variables

- Program variable declared in host language.
- Used in embedded SQL to transfer data from database into program and vice versa.
- Can be used anywhere a constant can appear.
- Cannot be used to represent database objects, such as table names or column names.
- To use host variable, prefixed it by a colon (:).

Host Language Variables

```
EXEC SQL UPDATE staff
        SET salary = salary + :increment
        WHERE sno = 'SL21';
```

- Need to declare host language variables to SQL, as well as to host language:

```
EXEC SQL BEGIN DECLARE SECTION;
        float increment;
EXEC SQL END DECLARE SECTION;
```


Indicator Variables

- Indicates presence of null:
  - 0 associated host variable contains valid value.
  - <0 associated host variable should be assumed to contain a null; actual contents of host variable irrelevant.
  - >0 associated host variable contains valid value.
- Used immediately following associated host variable with a colon (:) separating 2 variables.

Indicator Variables - Example

```
EXEC SQL BEGIN DECLARE SECTION;
        char  address[51];
        short address_ind;
EXEC SQL END DECLARE SECTION;
   address_ind = -1;
   EXEC SQL UPDATE staff
                SET address = :address :address_ind
                WHERE sno = 'SL21';
```


Singleton SELECT - Retrieves Single Row

```
EXEC SQL SELECT lname, tel_no
        INTO :last_name, :tel_no :telno_ind,
        FROM staff
        WHERE sno = 'SL21';
```

- Must be 1:1 correspondence between expressions in SELECT list and host variables in INTO clause.
- If successful, SQLCODE set to 0; if there are no rows that satisfies WHERE, SQLCODE set to NOT FOUND.

Cursors

- If query can return arbitrary number of rows, need to use *cursors*.
- Cursor allows host language to access rows of query one at a time.
- Cursor acts as a pointer to a row of query result. Cursor can be advanced by one to access next row.
- Cursor must be declared and opened before it can be used and it must be closed to deactivate it after it is no longer required.

Cursors - DECLARE CURSOR

- Once opened, rows of query result can be retrieved one at a time using FETCH:

```
EXEC SQL DECLARE
  property_cursor CURSOR FOR
        SELECT pno, area, city
        FROM property_for_rent
        WHERE sno = 'SL41';
```

Cursors - OPEN

- **OPEN statement opens specified cursor and positions it before first row of query result:**

```
EXEC SQL OPEN
        property_cursor FOR READONLY;
```

- **FOR READONLY indicates data will not be updated during fetch.**

Cursors - FETCH and CLOSE

- **FETCH retrieves next row of query result table:**

```
EXEC SQL FETCH property_cursor
        INTO :property_no, :area :area_ind
```

- **FETCH is usually placed in a loop. When there are no more rows to be returned, SQLCODE is set to NOT FOUND.**

```
EXEC SQL CLOSE property_cursor;
```

Dynamic Embedded SQL

- **With *static* embedded SQL, cannot use host variables where database object names required.**
- ***Dynamic SQL* allows this.**
- **Idea is to place complete SQL statement in a host variable, which is passed to DBMS to be executed.**
- **If SQL statements do not involve multi-row queries, use EXECUTE IMMEDIATE statement:**

```
EXEC SQL EXECUTE IMMEDIATE host_variable
```


Dynamic Embedded SQL

For example:

```
sprintf(buffer, "update staff set salary = salary + %f
                 where sno = 'SL21' ", increment);
EXEC SQL EXECUTE IMMEDIATE :buffer;
```

PREPARE and EXECUTE

- DBMS must parse, validate, and optimize each EXECUTE IMMEDIATE statement, build execution plan, and execute plan.
- OK if SQL statement is only executed once in program; otherwise inefficient.
- Dynamic SQL provides alternative: PREPARE and EXECUTE.
- PREPARE tells DBMS to ready dynamically built statement for later execution.

PREPARE and EXECUTE

- Prepared statement assigned name. When statement is subsequently executed, program need only specify this name:

```
EXEC SQL PREPARE statement_name
        FROM host_variable
EXEC SQL EXECUTE statement_name
        [ USING host_variable [, ..] |
        USING DESCRIPTOR descriptor_name ]
```

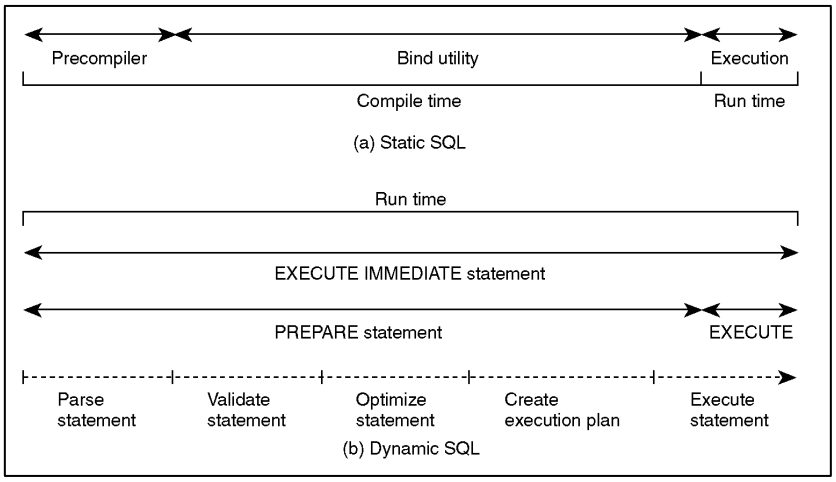
Parameter Markers

- **USING allows portions of prepared statement to be unspecified, replaced by *parameter markers* (?).**
- **Marker can appear anywhere in *host_variable* of PREPARE that constant can appear.**
- **Tells DBMS value will be supplied later, in EXECUTE statement.**

Parameter Markers

```
sprintf(buffer, "update staff set salary = ?
                 where sno = ?");
EXEC SQL PREPARE stmt FROM :buffer;
EXEC SQL EXECUTE stmt
                 USING :new_salary, :staff_no;
```

Static versus Dynamic SQL



SQL Descriptor Area (SQLDA)

- **Alternative way to pass parameters to EXECUTE statement is through SQLDA.**
- **Used when number of parameters and their data types unknown when statement formulated.**
- **SQLDA can also be used to dynamically retrieve data when do not know number of columns to be retrieved or the types of the columns.**

SQLDA for Ingres

```
/*
** SQLDA - Structure to hold data descriptions, used by embedded programs and INGRES
**  runtime during execution of dynamic SQL statements.
*/
typedef struct sqlvar_{
    short           sqltype;        /* type of column or variable */
    short           sqllen;         /* length of column of variable */
    char            *sqldata;       /* pointer to variable described by type and length */
    short           *sqlind;        /* pointer to indicator variable associated with host variable */
    struct {
        short       sqlname1;       /* length of name */
        char        sqlnamec[34];   /* name of result column from describe */
    } sqlname;
} IISQLVAR;

#define     IISQLDA_TYPE(sq_struct_tag, sq_sqlda_name, sq_num_vars) \
struct sq_struct_tag {                  \
    char            sqldaid[8];         \   /* contains fixed text "SQLDA   " */
    long            sqldabc;            \   /* length of SQLDA structure */
    short           sqln;               \   /* number of allocated sqlvar elements */
    short           sqld;               \   /* number of results columns associated with statement */
    IISQLVAR        sqlvar[sqln];       \   /* array of data */
} sq_sqlda_name;

#define IISQ_MAX_COLS       300
typedef IISQLDA_TYPE(sqlda_, IISQLDA, IISQ_MAX_COLS);
#define IISQDA_HEAD_SIZE    16
#define IISQDA_VAR_SIZE     sizeof(IISQLVAR)
```


SQL Descriptor Area (SQLDA)

- **SQLDA is divided into two parts:**
  - ***Fixed part,* identifying structure as SQLDA and specifying size of this instantiation of SQLDA. Used only for SELECT.**
  - ***Variable part,* containing data relating to each parameter passed to or received from DBMS.**

Fields in variable part of SQLDA

Sqltype: Code corresponding to data type of parameter passed in.

Sqllen: Length of associated data type in bytes.

***Sqldata** Pointer to a data area within program that contains parameter value.

***Sqlind** Pointer to an indicator variable associated with parameter value.

- Remaining fields used when retrieving data from database.

Important Fields in Fixed Part of SQLDA

Sqln: Number of elements allocated to variable part of SQLDA. Must be set by program before using SQLDA.

Sqld: Actual number of columns in SELECT. Set by DBMS.

- If DESCRIBE returns 0 for SQLD, statement is not SELECT.

Fields in Variable Part of SQLDA for Retrieval

- **Sqltype Code corresponding to data type of column being retrieved.**
- **Sqllen Length of associated data type in bytes.**
- ***Sqldata Pointer to a data area within program that will receive column result.**
- ***Sqlind Pointer to indicator variable associated with column result.**
- **Sqlname Length and name of associated column.**

DESCRIBE

- **Returns names, data types, and lengths of columns specified in query into an SQLDA.**
- **For non-select, sets SQLD to 0.**

```
EXEC SQL DESCRIBE statement_name
        USING descriptor_name
```

- ***statement_name* is name of prepared statement and *descriptor_name* is name of an initialized SQLDA.**

DESCRIBE

```
sprintf(query, "select pno, comment from viewing");
    EXEC SQL PREPARE stmt FROM :query;
    EXEC SQL DESCRIBE stmt INTO :sqlda;
```

- In this case, following information will be filled in:

sqld	= 2		
sqlvar[0].sqltype	= 20	sqlvar[1].sqltype	= −21
sqlvar[0].sqllen	= 5	sqlvar[1].sqllen	= 40
sqlvar[0].sqlname.sqlname1	= 3	sqlvar[1].sqlname.sqlname1	= 7
sqlvar[0].sqlname.sqlnamec	= PNO	sqlvar[1].sqlname.sqlnamec	= COMMENT

Multi-Row Selects

- Again, use cursors to retrieve data from a query result table that has an arbitrary number of rows.

```
EXEC SQL DECLARE cursor_name
        CURSOR FOR select_statement
EXEC SQL OPEN cursor_name [FOR READONLY]
    [USING host_variable [,...] |
        USING DESCRIPTOR descriptor_name ]
EXEC SQL FETCH cursor_name
        USING DESCRIPTOR descriptor_name
EXEC SQL CLOSE cursor_name
```

Multi-Row Selects

- **OPEN allows values for parameter markers to be substituted using one or more *host_variables* in:**
  - **USING clause or**
  - **passing values via *descriptor_name* (SQLDA) in a USING DESCRIPTOR clause.**
- **Main difference is with FETCH, which now uses *descriptor_name* to receive rows of query result table.**
- **Before FETCH, program must provide data areas to receive retrieved data and indicator variables, and set up SQLLEN, SQLDATA, and SQLIND accordingly.**

Open Database Connectivity (ODBC)

- **With an API, rather than embedding raw SQL within program, DBMS vendor provides API.**
- **API consists of set of library functions for many common types of database accesses.**
- **One problem with this approach has been lack of interoperability.**
- **To standardize this approach, Microsoft produced ODBC standard.**
- **ODBC provides common interface for accessing heterogeneous SQL databases, based on SQL.**

Open Database Connectivity (ODBC)

- Interface (built on 'C') provides high degree of interoperability: single application can access different SQL DBMSs through common code.
- Enables developer to build and distribute c-s application without targeting specific DBMS.
- Database drivers are then added to link application to user's choice of DBMS.
- ODBC is now emerging as de facto industry standard.

ODBC's Flexibility

- Applications not tied to proprietary vendor API.
- SQL statements can be explicitly included in source code or constructed dynamically.
- An application can ignore underlying data communications protocols.
- Data can be sent and received in format that is convenient to application.
- Design aligned with X/Open and ISO CLI.
- There are ODBC drivers available today for more than 50 of most popular DBMSs.

ODBC Interface

- Library of functions that allow application to connect to DBMS, execute SQL statements, and retrieve results.
- A standard way to connect and log on to a DBMS.
- A standard representation of data types.
- A standard set of error codes.
- SQL syntax based on the X/Open and ISO Call-Level Interface (CLI) specifications.

ODBC Architecture

- ODBC architecture has four components:
  - Application
  - Driver Manager
  - Driver and Database Agent
  - Data Source.

ODBC Architecture

Application - performs processing and calls ODBC functions to submit SQL statements to DBMS and to retrieve results from DBMS.

Driver Manager - loads drivers on behalf of application. Driver Manager, provided by Microsoft, is Dynamic-Link Library (DLL).

Driver and Database Agent - process ODBC function calls, submit SQL requests to specific data source, and return results to application.

ODBC Architecture

- **If necessary, driver modifies application's request so that it conforms to syntax supported by associated DBMS.**
- **With multiple drivers, these tasks performed by the driver; no database agent exists.**
- **With single driver, agent designed for each associated DBMS and runs on database server side.**

Data Source - consists of data user wants to access and its associated DBMS, and its host operating system, and network platform, if any.

ODBC Architecture