Database Management System

MODULE-3 Constraints, Views and SQL

VSIT | Vidyalankar School of Information Technology
NAAC ACCREDITED COLLEGE

Vidyalankar School of Information Technology
Wadala (E), Mumbai
www.vsit.edu.in

Compiled by: Rohini Desai
rohini.g@vsit.edu.in

VSIT | Vidyalankar School of Information Technology

NAAC Accredited College

Certificate

This is to certify that the e-book titled "Database Management Systems" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.

Signature
Prof. Rohini Desai
Assistant Professor
Department of IT

Date: 06-06-2019

⚠ *DISCLAIMER: The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

Unit III: Constraints, Views and SQL

Contents

- Constraints
- types of constrains
- Integrity constraints
- Views: Introduction to views
- data independence
- security
- updates on views
- comparison between tables and views
- SQL: data definition
- aggregate function
- Null Values
- nested sub queries
- Joined relations
- Triggers

Recommended Books

- Database System and Concepts A Silberschatz, H Korth, S Sudarshan McGrawHill
- Database Systems Rob Coronel Cengage Learning Twelfth Edition
- Programming with PL/SQL for Beginners H. Dand, R. Patil and T. Sambare
- Introduction to Database System C.J.Date

Unit III	Pre-requisites	Sem. II	Sem. III	Sem. IV	Sem. V	Sem. VI
Constraints, Views and SQL	-	-	-	-	-	-

CONSTRAINTS & ITS TYPES

Required Data

The simplest data integrity constraint requires that a column contain a non−NULL value. The ANSI/ISO standard and most commercial SQL products support this constraint by allowing you to declare that a column is NOT NULL when the table containing the column is first created. The NOT NULL constraint is specified as part of the CREATE TABLE statement.

When a column is declared NOT NULL, the DBMS enforces the constraint by ensuring the following:

- Every INSERT statement that adds a new row or rows to the table must specify a non-NULL data value for the column. An attempt to insert a row containing a NULL value (either explicitly or implicitly) results in an error.
- Every UPDATE statement that updates the column must assign it a non-NULL data value. Again, an attempt to update the column to a NULL value results in an error.

One disadvantage of the NOT NULL constraint is that it must usually be specified when a table is first created. Typically, you cannot go back to a previously created table and disallow NULL values for a column. Usually, this disadvantage is not serious because it's obvious when the table is first created which columns should allow NULLS and which should not There is also a potential logical problem with adding the NOT NULL constraint to an existing table. If one or more rows of that table already contain NULL values, then what should the DBMS do with those rows? They represent valid real-world objects, but they now violate the (new) required data constraint.

The inability to add a NOT NULL constraint to an existing table is also partly a result of the way most DBMS brands implement NULL values internally. Usually a DBMS reserves an extra byte in every stored row of data for each column that permits NULL values. The extra byte serves as a null indicator for the column and is set to some specified value to indicate a NULL value. When a column is defined as NOT NULL, the indicator byte is not present, saving disk storage space. Dynamically adding and removing NOT NULL constraints would thus require on-the-fly reconfiguration of the stored rows on the disk, which is not practical in large database.

SALESREPS (child) rows with matching office numbers. Similarly, each SALESREPS (child row has exactly one OFFICES (parent) row with a matching office number.

Suppose you tried to insert a new row into the SALESREPS table that contained an invalid office number, as in this example :

```
INSERT INTO SALESREPS (EMPL_NUM, NAME, REP_OFFICE, AGE,
                       HIRE_DATE, SALES)
VALUES (115, 'George Smith', 31, 37, '01-APR-90', 0.00)
```

On the surface, there's nothing wrong with this INSERT statement. In fact, many SQL implementations will successfully add the row. The database will show George Smith works in office number 31, even though no office number 31 is listed in the OFFICES table. The newly inserted row clearly breaks the parent/child relationship between the

OFFICES and SALESREPS tables. In fact, the office number in the INSERT statement is probably an error—the user may have intended office number 11, 21, or 13.

It seems clear that every legal value in the REP_OFFICE column should be forced to match some value that appears in the OFFICE column. This rule is known as a referential integrity constraint. It ensures the integrity of the parent/child relationships created by foreign keys and primary keys.

Referential integrity has been a key part of the relational model since it was first proposed by Codd. However, referential integrity constraints were not included in IBM's prototype System/R DBMS, nor in early releases of DB2 or SQL/DS. IBM added referential integrity support to DB2 in 1989, and referential integrity was added to the SQL1 standard after its initial release. Most DBMS vendors today support referential integrity constraints.

INTEGRITY CONSTRAINTS

Column Check Constraints (SQL2)

A SQL2 check constraint is a search condition, like the search condition in a WHERE clause, that produces a true/false value. When a check constraint is specified for a column, the DBMS automatically checks the value of that column each time a new row is inserted or a row is updated to ensure that the search condition is true. If not, the INSERT or UPDATE statement fails. A column check constraint is specified as part of the column definition within the CREATE TABLE statement.

Consider this excerpt from a CREATE TABLE statement, modified from the definition of the demo database to include three check constraints :

```
CREATE TABLE SALESREPS
        ( EMPL_NUM INTEGER NOT NULL
                        CHECK (EMPL_NUM BETWEEN 101 AND
199).
                AGE INTEGER
                        CHECK (AGE >= 21),
                □
                □
                □
                QUOTA MONEY
                        CHECK (MONEY >= 0.0)
                □
                □
                □
```

The first constraint (on the EMPL_NUM column) requires that valid employee numbers be three-digit numbers between 101 and 199. The second constraint (on the AGE column) similarly prevents hiring of minors. The third constraint (on the QUOTA column) prevents a salesperson from having a quota target less than $ 0.00.

All three of these column check constraints are very simple examples of the capability specified by the SQL2 standard. In general, the parentheses following the keyword CHECK can contain any valid search condition that makes sense in the context of a column definition. With this flexibility, a check constraint can compare values from two different columns of the table, or even compare a proposed data value against other values from the database. These capabilities are more fully described in the “Advanced constraint Capabilities” section later in this chapter.

Domain (SQL2)

A SQL2 domain generalizes the check−constraint concept and allows you to easily apply the same check constraint to many different columns within a database. A domain is a collection of legal data values. You specify a domain and assign it a domain name using the SQL2 CREATE DOMAIN statement. As with the check−constraint definition, a search condition is used to define the range of legal data values. For example, here is a SQL2 CREATE DOMAIN statement to create the domain VALID_EMPLOYEE_ID, which includes all legal employee numbers:

```
CREATE DOMAIN VALID_EMPLOYEE_ID INTEGER
        CHECK (VALUE BETWEEN 101 AND 199)
```

After the VALID_EMPLOYEE_ID domain has been defined, it may be used to define columns in database tables instead of a data type. Using this capability, the example REATE TABLE statement for the SALESREPS table would appear as :

```
CREAT TABLE SALESREPS
(EMPL_NUM VALID_EMPLOYEE_ID,
        AGE INTEGER
                CHECK (AGE >= 21),
    
    
    
QUOTA MONEY
        CHECK (MONEY >= 0.0)
    
    
    
```

The advantage of using the domain is that if other columns in other tables also contain employee numbers, the domain name can be use repeatedly, simplifying the table definitions. The OFFICES table contains such a column :

```
CREATE TABLE OFFICES
        (OFFICE INTEGER NOT NULL,
                CITY VARCHAR (15) NOT NULL,
        REGION VARCHAR (10) NOT NULL,
                MGR VALID_EMPLOYEE_ID,
        TARGET MONEY,
                SALES MONEY NOT NULL
            
            
            
```

Another advantage of domains is that the definition of valid data (such as valid employee numbers, in this example) is stored in one central place within the database. If the definition changes later (for example, if the company grows and employee numbers in the range 200−299 must be allowed), it is much easier to change one domain definition than to change many column constraints scattered throughout the database. In a large enterprise database, there may literally be hundreds of defined domains, and the benefits of SQL2 domains for change management can be very substantial.

https://www.youtube.com/watch?v=D2qU29YNgr4

VIEWS

INTRODUCTION

The tables of a database define the structure and organization of its data. However SQL also lets you look at the stored data in other ways by defining alternative views of the data. A view is a SQL query that is permanently stored in the database and assigned a name. The results of the stored query are visible through the view, and SQL lets you access these query results as if they were, in fact, a real table in the database.

Views are an important part of SQL for several reasons:

- Views let you tailor the appearance of a database so that different users see it from different perspectives.
- Views let you restrict access to data, allowing different users to see only certain rows or certain columns of a table.
- Views simplify database access by presenting the structure of the stored data in the way that is most natural for each user.

This chapter describes how to create views and how to use views to simplify processing and enhance the security of a database.

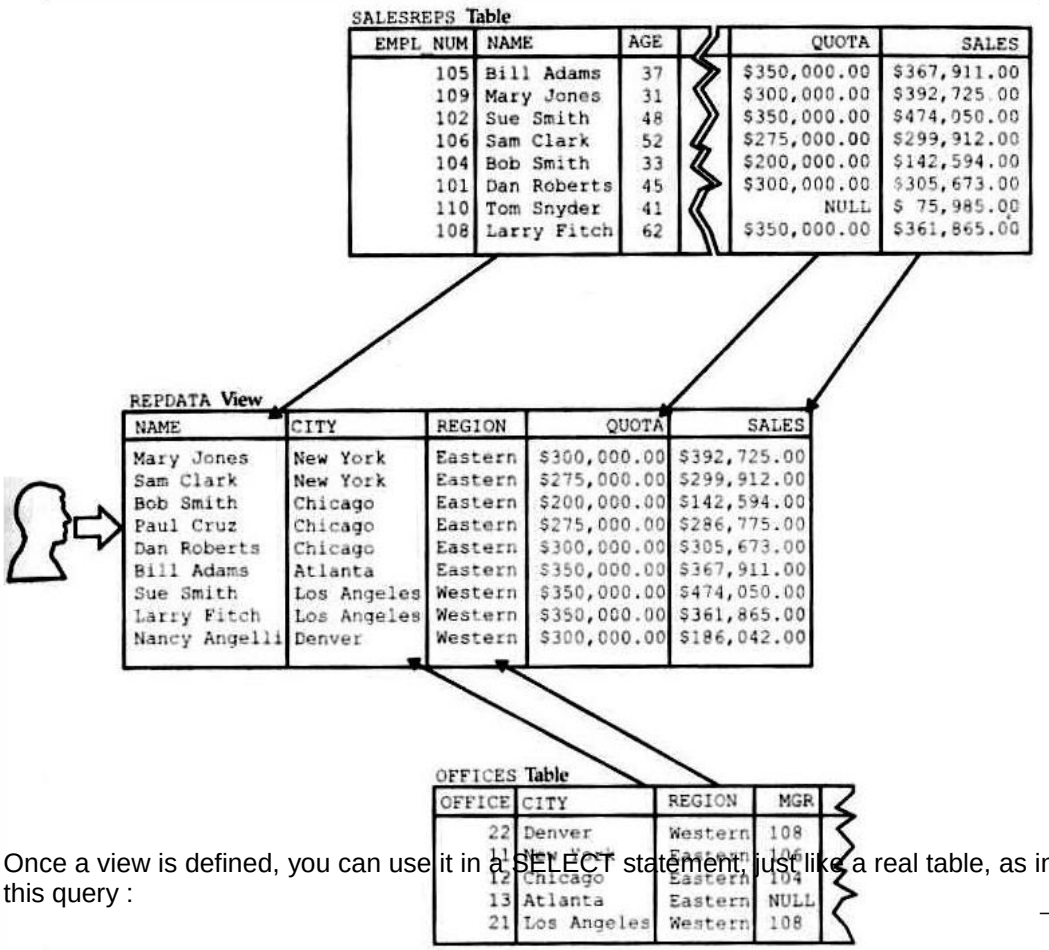
What is a View?

A view is a virtual table in the database whose contents are defined by a query, as shown in Fig. 1. To the database user, the view appears just like a real table, with a set of named columns and rows of data. But unlike a real table, a view does not exist in the database as a stored set of data values. Instead, the rows and columns of data visible through the view are the query results produced by the query that defines the view. SQL creates the illusion of the view by giving the view a name like a table name and storing the definition of the view in the database.

The view shown in figure 1 is typical. It has been given the name REPDATA and is defined by this two-table query:

```
SELECT NAME, CITY, REGION, QUOTA, SALESREPS. SALES
     FROM SALESREPS, OFFICES
WHERE REP_OFFICE = OFFICE
```

The data in the view comes from the SALESREPS and OFFICES tables. These tables are called the source table for the view because they are the source of the data that is visible through the view. This view contains one row of information for each salesperson, extended with the name of the city and region where the salesperson works. As shown in the figure, the view appears as a table, and its contents look just like the query results that you would obtain if you actually ran the query.



Once a view is defined, you can use it in a SELECT statement, just like a real table, as in this query :

Fig. 1 : A typical view with two source tables

List the salespeople who are over quota, showing the name, city and region for each salesperson.

```
SELECT NAME, CITY, REGION
        FROM REPDATA
WHERE SALES > QUOTA
```

NAME	CITY	REGION
Mary Jones	New York	Eastern
Sam Clark	New York	Eastern
Dan Roberts	Chicago	Eastern
Paul Cruz	Chicago	Eastern
Bill Adams	Atlanta	Eastern
Sue Smith	Los Angeles	Western
Larry Fitch	Los Angeles	Western

The name of the view, REPDATA, appears in the FROM clause just like a table name, and the columns of the view are referenced in the SELECT statement just like the columns of a real table. For some views, you can also use the INSERT, DELETE, and UPDATE statements to modify the data visible through the view, as if it were a real table. Thus, for all practical purposes, the view can be used in SQL statements as if it were a real table.

How the DBMS Handles Views?

When the DBMS encounters a reference to a view in a SQL statement, it finds the definition of the view stored in the database. Then the DBMS translates the request that references the view into an equivalent request against the source tables of the view and carries out the equivalent request. In this way, the DBMS maintains the illusion of the view while maintaining the integrity of the source tables.

For simple views, the DBMS may construct each row of the view on the fly, drawing the data for the row from the source table(s). For more complex views, the DBMS must actually materialize the view; that is, the DBMS must actually carry out the query that defines the view and store its results in a temporary table. The DBMS fills your requests for view access from this temporary table and discards the table when it is no longer needed. Regardless of how the DBMS actually handles a particular view, the result is the same for the user—the view can be referenced in SQL statements exactly as if it were a real table in the database.

Advantages of Views

Views provide a variety of benefits and can be useful in many different types of databases. In a personal computer database, views are usually a convenience, defined to simplify database requests. In a production database installation, views play a central role in defining the structure of the database for its users and enforcing its security. Views provide these major benefits:

- Security. Each user can be given permission to access the database only through a small set of views that contain the specific data the user is authorized to see, thus restricting the user's access to stored data.
- Query simplicity. A view can draw data from several different tables and present it as a single table, turning multitable queries into single-table queries against the view.

- Structural simplicity Views can give a user a personalized view of the database structure, presenting the database as a set of virtual tables that make sense for that user.
- Insulation from change A view can present a consistent, unchanged image of the structure of the database, even if the underlying source tables are split, restructured, or renamed.
- Data integrity. If data is accessed and entered through I view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

Disadvantages of Views

While views provide substantial advantages, there are also two major disadvantages to using a view instead of a real table:

- **Performance.** Views create the appearance of a table, but the DBMS must still translate queries against the view into queries against the underlying source tables It the view is defined by a complex multitable query, then even a simple query against the view becomes a complicated join, and it may take a long time to complete.
- **Update restrictions.** When a user tries to update rows of a view, the DBMS must translate the request into an update on rows of the underlying source tables This is possible for simple views, but more complex views cannot be updated; they are read-only.

These disadvantages mean that you cannot indiscriminately define views and use them instead of the source tables. Instead, you must in each case consider the advantages provided by using a view and weigh them against the disadvantages.

Creating a View (CREATE VIEW)

The CREATE VIEW statement, shown in figure 2, is used to create a view. The statement assigns a name to the view and specifies the query that defines the view. To create the view successfully, you must have permission to access all of the tables referenced in the query.

The CREATE VIEW statement can optionally assign a name to each column in the newly created view. If a list of column names is specified, it must have the same number of items as the number of columns produced by the query. Note that only the column names are specified; the data type, length, and other characteristics of each column are derived from the definition of the columns in the source tables. If the list of column names is omitted from the CREATE VIEW statement, each column in the view takes the name of the corresponding column in the query. The list of column names must be specified if the query includes calculated columns or if it produces two columns with identical names.

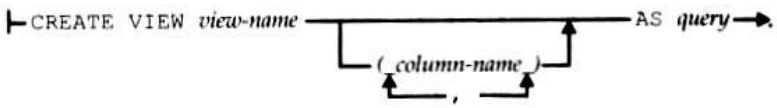


Fig. 2 : The CREATE VIEW statement syntax diagram

Although all views are created in the same way, in practice, different types of views are typically used for different purposes. The next few sections examine these types of views

Horizontal Views

A common use of views is to restrict a user's access to only selected rows of a table. For example, in the sample database, you may want to let a sales manager see only the SAI.ESREPS rows for salespeople in the manager's own region. To accomplish this, you can define two views, as follows:

Create a view doming Eastern region salespeople.

```
CREATE VIEW EASTREPS AS
        SELECT *
                FROM SALESREPS
                WHERE REP_OFFICE IN (11, 12, 13)
```

Create a Okw showing Western region salespeople.

```
CREATE VIEW WESTREPS AS
        SELECT *
                FROM SALESREPS
                WHERE REP_OFFICE IN (21, 22)
```

Now you can give each sales manager permission to access either the EASTREPS or the WESTREPS view, denying them permission to access the other view and the SALESREPS table itself. This effectively gives the sales manager a customized view of the SALESREPS table, showing only salespeople in the appropriate region.

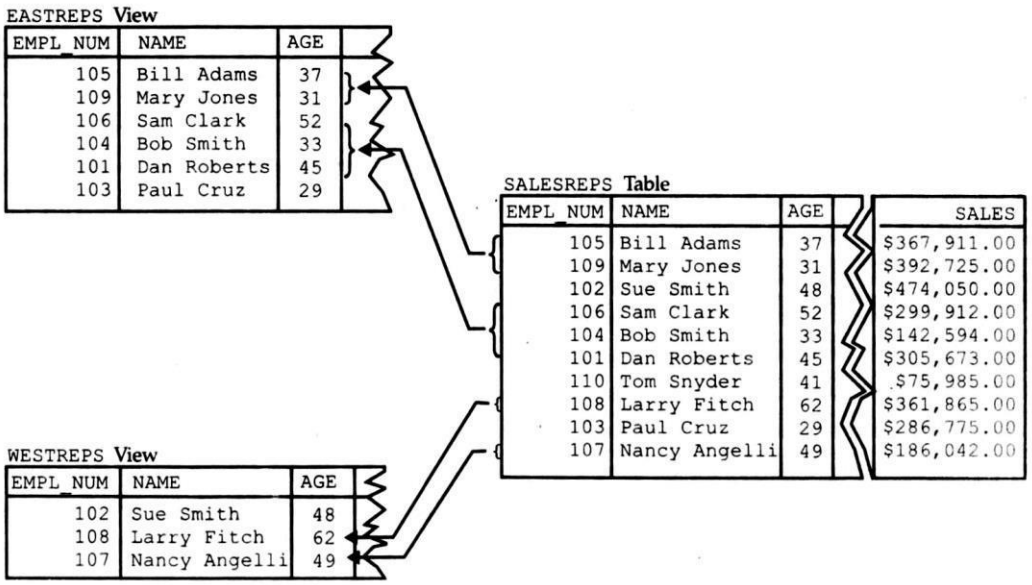


Fig. 3 : Two horizontal views of the SALESREPS table

A view like SASTREPS or WESTREPS is often called a horizontal view. As shown in Fig. 3, a horizontal view slices the source table horizontally to create the view. All of the columns of the source table participate in the view, but only some of its rows are visible through the view. Horizontal views are appropriate when the source table contains data that relates to various organizations or users. They provide a private table for each user, composed only of the rows needed by that user.

Here are some more examples of horizontal views:

Define a view containing only Eastern region offices.

```
CREATE VIEW EASTOFFICES AS
    SELECT *
        FROM OFFICES
        WHERE REGION = 'Eastern'
```

Define a view for Sue Smith (employee number 102) containing only orders placed by customers assigned to her.

```
CREATE VIEW SUEORDERS AS
  SELECT*
    FROM ORDERS
  WHERE CUST IN (SELECT CUST_NUM
                   FROM CUSTOMERS
                 WHERE CUST_REP = 102 )
```

Define a view showing only those customers who have more than $30,000 worth of orders currently on the books.

```
CREATE VEW BIGCUSTOMERS AS
  SELECT *
    FROM CUSTOMERS
  WHERE 30000.00 < (SELECT SUM (AMOUNT)
                                    FROM ORDERS
                                    WHERE        CUST=
CUST_NUM)
```

In each of these examples, the view is derived from a single source table. The view is defined by a SELECT * query and therefore has exactly the same columns as the source table. The WHERE clause determines which rows of the source table are visible in the view.

Vertical Views

Another common use of views is to restrict a user's access to only certain columns of a table. For example, in the sample database, the order-processing department may need access to the employee number, name, and office assignment of each salesperson, because this information may be needed to process an order correctly. However, there is no need for the order-processing staff to see the salesperson's year-to-date sales or quota. This selective view of the SALESREPS table can be constructed with the following view:

Create a view showing selected salesperson information.

```
CREATE VIEW REPINFO AS
    SELECT EMPL_NUM, NAME,
        REP_OFFICE FROM SALESREPS
```

By giving the order-processing staff access to this view and denying access to the SALESREPS table itself, access to sensitive sales and quota data is effectively restricted.

A view like the REPINFO view is often called a vertical view. As shown in figure 4, vertical view slices the source table vertically to create the view. Vertical views are only found where the data stored in a table is used by various users or groups of users. They provide a private table for each user, composed only of the columns needed by that user.

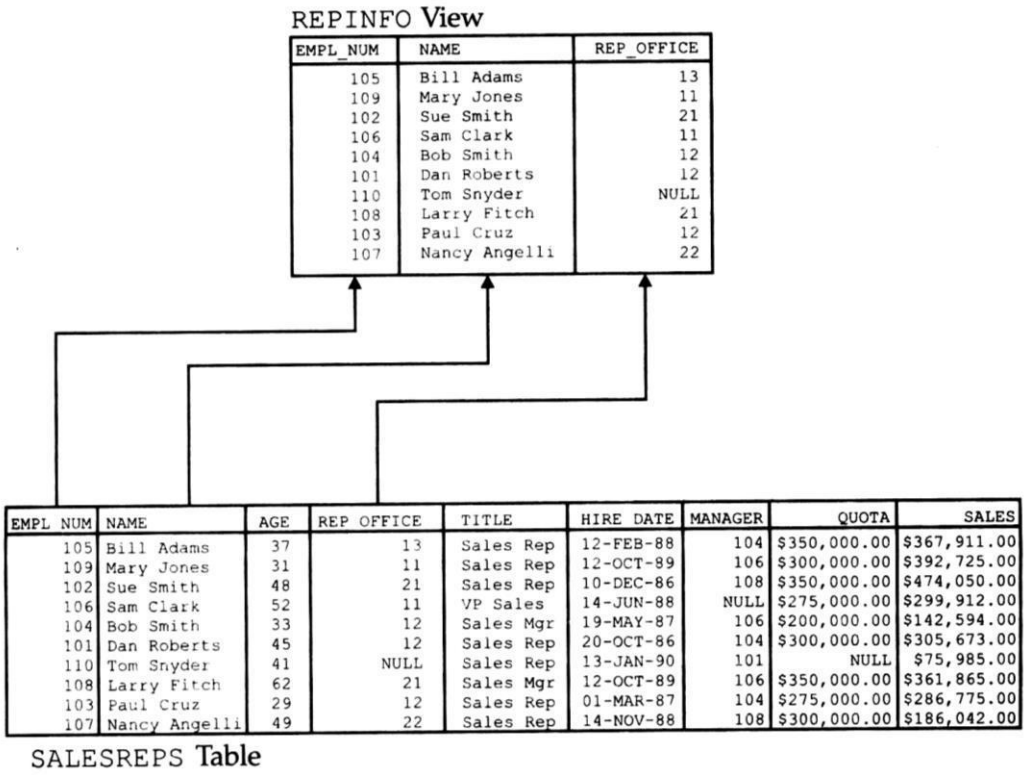
Here are some more examples of vertical views:

Define a view of the OFFICES table for the order-processing staff that includes the office's city, office number, and region.

```
CREATE VIEW OFFICEINFO AS
        SELECT OFFICE, CITY, REGION
            FROM OFFICES
```

Define a view of the CUSTOMERS table that includes only customer names and their assignment to salespeople.

```
CREATE VIEW CUSTINFO AS
        SELECT COMPANY, CUST_REP
            FROM CUSTOMERS
```



REPINFO View

EMPL_NUM	NAME	REP_OFFICE
105	Bill Adams	13
109	Mary Jones	11
102	Sue Smith	21
106	Sam Clark	11
104	Bob Smith	12
101	Dan Roberts	12
110	Tom Snyder	NULL
108	Larry Fitch	21
103	Paul Cruz	12
107	Nancy Angelli	22

SALESREPS Table

EMPL NUM	NAME	AGE	REP OFFICE	TITLE	HIRE DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	12-FEB-88	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	12-OCT-89	106	$300,000.00	$392,725.00
102	Sue Smith	48	21	Sales Rep	10-DEC-86	108	$350,000.00	$474,050.00
106	Sam Clark	52	11	VP Sales	14-JUN-88	NULL	$275,000.00	$299,912.00
104	Bob Smith	33	12	Sales Mgr	19-MAY-87	106	$200,000.00	$142,594.00
101	Dan Roberts	45	12	Sales Rep	20-OCT-86	104	$300,000.00	$305,673.00
110	Tom Snyder	41	NULL	Sales Rep	13-JAN-90	101	NULL	$75,985.00
108	Larry Fitch	62	21	Sales Mgr	12-OCT-89	106	$350,000.00	$361,865.00
103	Paul Cruz	29	12	Sales Rep	01-MAR-87	104	$275,000.00	$286,775.00
107	Nancy Angelli	49	22	Sales Rep	14-NOV-88	108	$300,000.00	$186,042.00

Fig. 4 : A vertical view of the SALEREPS table

In each of these examples, the view is derived from a single source table. The select list in the view definition determines which columns of the source table are visible in the view. Because these are vertical views, every row of the source table is represented in the view, and the view definition does not include a WHERE clause.

Row/Column Subset Views

When you define a view, SQL does not restrict you to purely horizontal or vertical slices of a table. In fact, the SQL language does not include the notion of horizontal and vertical views. These concepts merely help you to visualize how the view presents the information from the source table. It's quite common to define a view that slices a source table in both the horizontal and vertical dimensions, as in this example:

Define a view that contains the customer number, company name, and credit limit of all customers assigned to Bill Adams (employee number 105).

```
CREATE VIEW BILLCUST AS
        SELECT CUST_NUM, COMPANY,
                CREDIT_LIMIT FROM CUSTOMERS
        WHERE CUST_REP = 105
```

The data visible through this view is a row/column subset of the CUSTOMERS table. Only the columns explicitly named in the select list of the view and the rows that meet the search condition are visible through the view.

Grouped Views

The query specified in a view definition may include a GROUP BY clause. This type of view is called a grouped view, because the data visible through the view is the result of a grouped query Grouped views perform the same function as grouped queries; they group related rows of data and produce one row of query results for each group, summarizing the data in that group A grouped view makes these grouped query results into a virtual table, allowing you to perform further queries on them.

Here is an example of a grouped view :

Define a view that contains summary order data for each salesperson.

```
CREATE VIEW ORD_BY_REP (WHO, HOW_MANY, TOTAL, LOW, HIGH,
        AVERAGE) AS SELECT REP, COUNT (*), SUM (AMOUNT),
MIN(AMOUNT),
        MAX(AMOUNT),
        AVG (AMOUNT)
FROM ORDERS
GROUP BY REP
```

As this example shows, the definition of a grouped view always includes a column name list. The list assigns names to the columns in the grouped view, which are derived from column functions such as SUM () and MIN (). It may also specify a modified name for a grouping column. In this example, the REP column of the ORDERS table becomes the WHO column in the ORD_BY_REP view.

Once this grouped view is defined, it can be used to simplify queries. For example, this query generates a simple report that summarizes the orders for each salesperson :

Show the name, number of orders, total order amount, and average order size for each salesperson.

```
SELECT NAME, HOW_MANY, TOTAL, AVERAGE
        FROM SALESREPS, ORD_BY_REP
```

```
                    WHERE WHO =
EMPL_NUM ORDER BY TOTAL DESC
```

NAME	HOW_MANY	TOTAL	AVERAGE
Larry Fitch	7	$ 58,633.00	$ 8,376.14
Bill Adams	5	$ 39,327.00	$ 7,865.40
Nancy Angelli	3	$ 34,432.00	$ 11,477.33
Sam Clark	2	$ 32,958.00	$ 16,479.00
Dan Roberts	3	$ 26,628.00	$ 8,876.00
Tom Snyder	2	$ 23,132.00	$ 11,566.00
Sue Smith	4	$ 22,776.00	$ 5,694.00
Mary Jones	2	$ 7,105.00	$ 3,552.50
Paul Cruz	2	$ 2,700.00	$ 1,350.00

Unlike a horizontal or vertical view, the rows in a grouped view do not have a one-to-one correspondence with the rows in the source table. A grouped view is not just a filter on its source table that screens out certain rows and columns. It is a summary of the source tables; therefore, a substantial amount of DBMS processing is required to maintain the illusion of a virtual table for grouped views.

Grouped views can be used in queries just like other, simpler views. A grouped view cannot be updated, however. The reason should be obvious from the example. What would it mean to update the average order size for salesrep number 105? Because each row in the grouped view corresponds to a group of rows from the source table, and because the columns in the grouped view generally contain calculated data, there is no way to translate the update request into an update against the rows of the source table. Grouped views thus function as read-only views, which can participate in queries but not in updates.

Grouped views are also subject to the SQL restrictions on nested column functions. Recall from Chapter 8 that nested column functions, such as:

```
MIN (MIN(A))
```

are not legal in SQL expressions. Although the grouped view hides the column functions in its select list from the user, the DBMS still knows about them and enforces the restriction. Consider this example :

For each sates office, show the range of average order sizes for all salespeople who work in the office.

```
        SELECT REP_OFFICE, MIN(AVERAGE), MAX(AVERAGE)
FROM SALESREPS, ORD_BY_REP
WHERE EMPL_NUM = WHO
GROUP BY REP_OFFICE
```

Error : Nested column function reference

This query produces an error, even though it appears perfectly reasonable. It's a two-table query that groups the rows of the ORD_BY_REP view based on the office to which the salesperson is assigned. But the column functions MIN () and MAX () in the select list cause a problem. The argument to these column functions, the AVERAGE column, is itself the result of a column function. The "actual" query being requested from SQL is:

```
SELECT REP_OFFICE, MIN (AVG(AMOUNT)), MAX(AVG(AMOUNT))
        FROM SALESREPS, ORDERS
WHERE EMPL_NUM = REP
GROUP BY REP
GROUP BY REP_OFFICE
```

This query is illegal because of the double group by and the nested column functions. Unfortunately, as this example shows, a perfectly reasonable grouped SELECT statement may, in fact, cause an error if one of its source tables turns out to be a grouped view. There's no way to anticipate this situation; you must just understand the cause of the error when SQL reports it to you.

Joined Views

One of the most frequent reasons for using views is to simplify multitable queries. By specifying a two-table or three-table query in the view definition, you can create a joined view that draws its data from two or three different tables and presents the query results as a single virtual table. Once the view is defined, you can often use a simple single-table query against the view for requests that would otherwise each require a two-table or three-table join.

For example, suppose that Sam Clark, the vice president of sales, often runs queries against the ORDERS table in the sample database. However, Sam doesn't like to work with customer and employee numbers. Instead, he'd like to be able to use a version of the ORDERS table that has names instead of numbers. Here is a view that meets Sam's needs:

Create a view of the ORDERS table with names instead of numbers.

```
CREATE VIEW ORDER_INFO (ORDER_NUM, COMPANY, REP_NAME,
            AMOUNT) AS SELECT ORDER_NUM, COMPANY, NAME,
AMOUNT
            FROM ORDERS, CUSTOMERS,
    SALESREPS WHERE CUST = CUST_NUM AND REP
    = EMPL_NUM
```

This view is defined by a three-table join. As with a grouped view, the processing required to create the illusion of a virtual table for this view is considerable. Each row of the view is derived from a combination of one row from the ORDERS table, one row from the CUSTOMERS table, and one row from the SALESREPS table.

Although it has a relatively complex definition, this view can provide some real benefits. Here is a query against the view that generates a report of orders, grouped by salesperson:

Show the total current orders for each company for each salesperson.

```
        SELECT REP_NUM, COMPANY, SUM(AMOUNT)
            FROM ORDER_INFO
GROUP BY REP_NAME, COMPANY
```

REP_NAME	COMPANY	SUM(AMOUNT)

Bill Adams	Acme Mfg.	$ 35,582.00
Bill Adams	JCP Inc.	$ 3,745.00
Dan Roberts	First Corp.	$ 3,978.00
Dan Roberts	Holm & Landis	$ 150.00
Dan Roberts	Ian & Schmidt	$ 22,500.00
Larry Fitch	Midwest Systems	$ 3,608.00
Larry Fitch	Orion Corp.	$ 7,100.00
Larry Fitch	Zetacorp	$ 47,925.00

Note that this query is a single−table SELECT statement, which is considerably simpler than the equivalent three−table SELECT statement for the source tables :

```
SELECT NAME, COMPANY, SUM (AMOUNT)
        FROM SALESREPS, ORDERS,
CUSTOMERS WHERE REP = EMPL_NUM
        AND CUST = CUST_NUM
GROUP BY NAME, COMPANY
```

Similarly, it's easy to generate a report of the largest orders, showing who placed them and who received them, with this query against the view:

Show the largest current orders, sorted by amount.

```
SELECT COMPANY, AMOUNT, REP_NAME
        FROM ORDER_INFO
WHERE AMOUNT > 20000.00
        ORDER BY AMOUNT DESC
```

COMPANY	AMOUNT	REP_NAME
Zetacorp	$ 45,000.00	Larry Fitch
J. P. Sinclair	$ 31,500.00	Sam Clark
Chen associates	$ 31,350.00	Nancy Angelli
acme Mfg.	$ 27,500.00	Bill Adams
Ace International	$ 22,500.00	Tom Snyder
Ian & Schmidt	$ 22,500.00	Dan Roberts

The view makes it much easier to see what’s going on in the query than if it were expressed as the equivalent three−table join. Of course, the DBMS must work just as hard to generate the query results for the single−table query against the view as it would to generate the query results for the equivalent three−table query. In fact, the DBMS must perform slightly more work to handle the query against the view.
However, for the human user of the database, it’s much easier to write and understand the single−table query that references the view.

UPDATING A VIEW

What does it mean to insert a row of data into a view, delete a row from a view, or update a row of a view? For some views, these operations can obviously be translated into equivalent operations against the source table(s) of the view. For example, consider once again the EASTREPS view, defined earlier in this chapter :

Create a view showing Eastern region salespeople.

```
CREATE VIEW EASTREPS AS
        SELECT *
                FROM SALESREPS
                WHERE REP_OFFICE IN (11, 12, 13)
```

This is a straightforward horizontal view, derived from a single source table. As shown in Fig. 5, it makes sense to talk about inserting a row into this view; it means the new row should be inserted into the underlying SALESREPS table from which the view is derived. It also makes sense to delete a row from the EASTREPS view; this would delete the corresponding row from the SALESREPS table. Finally, updating a row of the EASTREPS view makes sense; this would update the corresponding row of the SALESREPS table. In each case, the action can be carried out against the corresponding row of the source table, preserving the integrity of both the source table and the view.

However, consider the ORD_BY_REP grouped view, as it was defined earlier in the section "Grouped Views":

Define a view that contains summary order data for each salesperson.

```
CREATE VIEW ORD_BY_REP (WHO, HOW_MANY, TOTAL, LOW, HIGH,
AVERAGE) AS SELECT REP, COUNT (*), SUM(AMOUNT),
MIN(AMOUNT), MAX(AMOUNT),
        AVG(AMOUNT)
        FROM ORDERS
        GROUP BY REP
```

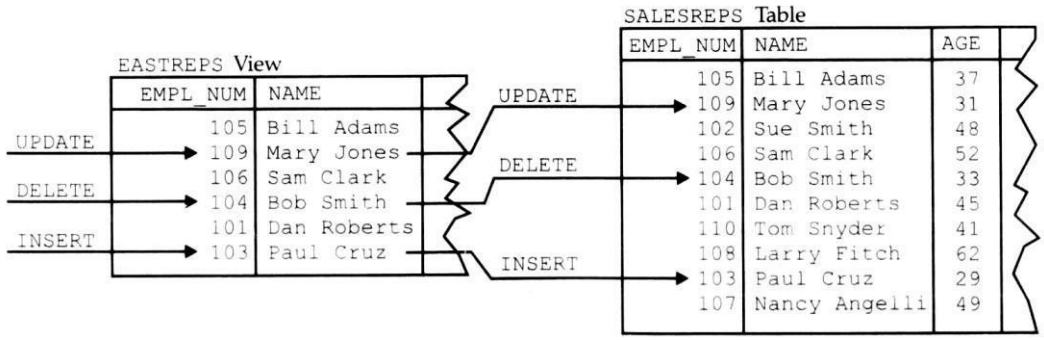


Fig. 5 : Updating data through a view

There is no one-to-one correspondence between the rows of this view and the rows of the underlying ORDERS table, so it makes no sense to talk about inserting, deleting, or updating rows of this view. The ORO_BY_REP view is not updateable; it is a read-only view.

The EASTREPS view and the ORD_BY_REP view are two extreme examples in term of the complexity of their definitions. There are views more complex than EASTREPS where it still makes sense to update the view, and there are views less complex than ORD_BY_REP where updates do not make sense. In fact, which views can be updated and which cannot has been an important relational database research problem over the years.

View Updates and the ANSI/ISO Standard

The ANSI/ISO SQL1 standard specifies the views that must be updateable in a database that claims conformance to the standard. Under the standard, a view can be updated if the query that defines the view meets all of these restrictions:

- DISTINCT must not be specified; that is, duplicate rows must not be eliminated from the query results.
- The FROM clause must specify only one updateable table; that is, the view must have a single source table for which the user has the required privileges. If the source table is itself a view, then that view must meet these criteria.
- Each select item must be a simple column reference; the select list cannot contain expressions, calculated columns, or column functions.
- The WHERE clause must not include a subquery; only simple row-by-row search conditions may appear.
- The query must not include a GROUP BY or a HAVING clause.

The basic concept behind the restrictions is easier to remember than the rules themselves. For a view to be updateable. the DBMS must be able to trace any row of the view back to its source row in the source table Similarly, the DBMS must be able to trace each individual column to be updated back to its source column in the source table.
If the view meets this test, then it's possible to define meaningful INSERT, DELETE, and UPDATE operations for the view in terms of the source table(s).

Checking View Updates (CHECK OPTION)

If a view is defined by a query that includes a WHERE clause, only rows that meet the search condition are visible in the view. Other rows may be present in the source table(s) from which the view is derived, but they are not visible through the view. For example, the EASTREPS view, described in the "Horizontal Views" section earlier in this chapter, contains only those rows of the SALESREPS table with specific values in the REP_OFFICE column:

Create a view showing Eastern region salespeople.

```
CREATE VIEW EASTREPS AS
        SELECT *
        FROM SALESREPS
        WHERE REP_OFFICE IN (11, 12, 13)
```

This is an updateable view for most commercial SQL implementations. You can add a new salesperson with this INSERT statement:

```
INSERT INTO EASTREPS (EMPL_NUM, NAME, REP_OFFICE, AGE, SALES)
VALUES (113, 'Jake Kimball', 11, 43, 0.00)
```

The DBMS will add the new row to the underlying SALESREPS table, and the row will be visible through the EASTREPS view. But consider what happens when you add a new salesperson with this INSERT statement:

```
INSERT INTO EASTREPS (EMPL_NUM, NAME, REP_OFFICE, AGE, SALES)
VALUES (114, 'Fred Roberts', 21, 47, 0.00)
```

This is a perfectly legal SQL statement, and the DBMS will insert a new row with the specified column values into the SALESREPS table. However, the newly inserted row doesn't meet the search condition for the view. Its REP_OFFICE value (21) specifies the

Los Angeles office, which is in the Western region. As a result, if you run this query immediately after the INSERT statement:

```
SELECT EMPL_NUM, NAME,
REP_OFFICE FROM EASTREPS
```

EMPL_NUM	NAME	REP_OFFICE
105	Bill Adams	13
109	Mary Jones	11
106	Sam Clark	11
104	Bob Smith	12
101	Dan Roberts	12
103	Paul Cruz	12

the newly added row doesn't show up in the view. The same thing happens if you change the office assignment for one of the salespeople currently in the view. This UPDATE statement:

```
UPDATE EASTREPS
        SET REP_OFFICE = 21
WHERE EMPL_NUM = 104
```

modifies one of the columns for Bob Smith's row and immediately causes it to disappear from the view. Of course, both of the vanishing rows show up in a query against the underlying table:

```
SELECT EMPL_NUM, NAME,
        REP_OFFICE FROM SALESREPS
```

EMPL_NUM	NAME	REP_OFFICE
105	Bill Adams	13
109	Mary Jones	11
102	Sue Smith	21
106	Sam Clark	11
104	Bob Smith	21
101	Dan Roberts	12
110	Tom Snyder	NULL
108	Larry Fitch	21
103	Paul Cruz	12
107	Nancy Angelli	22
114	Fred Roberts	21

The fact that the rows vanish from the view as a result of an INSERT or UPDATE statement is disconcerting, at best. You probably want the DBMS to detect and prevent this type of INSERT or UPDATE from taking place through the view. SQL allows you to specify this kind of integrity checking for views by creating the view with a check option. The check option is specified in the CREATE VIEW statement, as shown in this redefinition of the EASTRFPS view:

```
CREATE VIEW EASTREPS AS
        SELECT *
                FROM SALESREPS
        WHERE REP_OFFICE IN (11, 12, 13)
WITH CHECK OPTION
```

When the check option is requested for a view, SQL automatically checks each INSERT and each UPDATE operation for the view to make sure that the resulting row(s) meet the search criteria in the view definition. If an inserted or modified row would not meet the condition, the INSERT or UPDATE statement fails, and the operation is not carried out.

The SQL2 standard specifies one additional refinement to the check option: the choice of CASCADED or LOCAL application of the check option. This choice applies when a view is created, and its definition is based not on an underlying table, but on one or more other views. The definitions of these underlying views might, in turn, be based on still other views, and so on. Each of the underlying views might or might not have the check option specified.

If the new view is created WITH CASCADED CHECK OPTION, any attempt to update the view causes the DBMS to go down through the entire hierarchy of view definitions on which it is based, processing the check option for each view where it is specified. If the new view is created WITH LOCAL CHECK OPTION, then the DBMS checks only that view; the underlying views are not checked. The SQL2 standard specifies CASCADED as the default, if the WITH CHECK OPTION clause is used without specifying LOCAL or CASCADED.

Dropping a View (DROP VIEW)

Recall that the SQL1 standard treated the SQL Data Definition Language (DDL) as a static specification of the structure of a database, including its tables and views. For this reason, the SQL1 standard did not provide the ability to drop a view when it was no longer needed. However, all major DBMS brands have provided this capability for some time. Because views behave like tables and a view cannot have the same name as a table, some DBMS brands used the DROP TABLE statement to drop views as well. Other SQL implementations provided a separate DROP VIEW statement.

The SQL2 standard formalized support for dropping views through a DROP VIEW statement. It also provides for detailed control over what happens when a user attempts to drop a view when the definition of another view depends on it. For example, suppose two views on the SALESREPS table have been created by these two CREATE VIEW statements:

```
CREATE VIEW EASTREPS AS
        SELECT *
        FROM SALESREPS
        WHERE REP_OFFICE IN (11, 12, 13)
CREATE VIEW NYREPS AS
        SELECT *
        FROM EASTREPS
        WHERE REP_OFFICE = 11
```

For purposes of illustration, the NYREPS view is defined in terms of the EASTREPS view, although it could just as easily have been defined in terms of the underlying table. Under the SQL2 standard, the following DROP VIEW statement removes both of the views from the database:

```
        DROP VIEW EASTREPS CASCADE
```

The CASCADE option tells the DBMS to delete not only the named view, but also any views that depend on its definition. In contrast, this DROP VIEW statement:

```
        DROP VIEW EASTREPS RESTRICT
```

fails with an error, because the RESTRICT option tells the DBMS to remove the view only if no other views depend on it. This provides an added precaution against unintentional side-effects of a DROP VIEW statement. The SQL2 standard requires that either RESTRICT or CASCADE be specified. But many commercial SQL products support a version of the DROP VIEW statement without an explicitly specified option for backward compatibility with earlier versions of their products released before the publication of the SQL2 standard. The specific behavior of dependent views in this case depends on the particular DBMS brand.

Source: https://www.youtube.com/watch?v=5pmkuoJu2Qg

DATA INDEPENDENCE

A very important advantage of using a DBMS is that it offers data independence. That is, application programs are insulated from changes in the way the data is structured and stored. Data independence is achieved through use of the three levels of data abstraction; in particular, the conceptual schema and the external schema provide distinct benefits in this area.

Relations in the external schema (view relations) are in principle generated on demand from the relations corresponding to the conceptual schema.

If the underlying data is reorganized, that is, the conceptual schema is changed, the definition of a view relation can be modified so that the same relation is computed as before. For example, suppose that the Faculty relation in our university database is replaced by the following two relations:

Faculty public(d: string , fname: string , o ce: integer)
Faculty private(d: string , sal: real)

Intuitively, some confidential information about faculty has been placed in a separate relation and information about offices has been added. The Courseinfo view relation can be redefined interms of Faculty public and Faculty private, which together contain all the

information in Faculty, so that a user who queries Courseinfo will get the same answers as before.

Thus users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This property is called logical data independence.

In turn, the conceptual schema insulates users from changes in the physical storage of the data. This property is referred to as physical data independence. The conceptual schema hides details such as how the data is actually laid out on disk, the file structure, and the choice of indexes. As long as the conceptual schema remains the same, we can change these storage details without altering applications. (Of course, performance might be affected by such changes.)

In practice, they could be pre-computed and stored to speed up queries on view relations, but the computed view relations must be updated whenever the underlying relations are updated.

SECURITY

When you entrust your data to a database management system, the security of the stored data is a major concern. Security is especially important in an SQL-based DBMS because interactive SQL makes database access very easy. The security requirements of a typical production database are many and varied:

- The data in any given table should be accessible to some users, but access by other users should be prevented.
- Some users should be allowed to update data in a particular table; others should be allowed only to retrieve data.
- For some tables, access should be restricted on a column-by-column basis.
- Some users should be denied interactive SQL access to a table but should be allowed to use application programs that update the table.

The SQL security scheme described in this chapter provides these types of protection for data in a relational database.

SQL Security Concepts

Implementing a security scheme and enforcing security restrictions are the responsibility of the DBMS software. The SQL language defines an overall framework for database security, and SQL statements are used to specify security restrictions. The SQL security scheme is based on three central concepts:

- **Users.** The actors in the database. Each time the DBMS retrieves, inserts, deletes, or updates data, it does so on behalf of some user. The DBMS permits or prohibits the action depending on which user is making the request.

- **Database objects.** The items to which SQL security protection can be applied. Security is usually applied to tables and views, but other objects such as forms, application programs, and entire databases can also be protected. Most users will have permission to use certain database objects but will be prohibited from using others.
- **Privileges.** The actions that a user is permitted to carry out for a given database object. A user may have permission to SELECT and INSERT rows in a certain table, for

example, but may lack permission to DELETE or UPDATE rows of the table. A different user may have a different set of privileges.

Figure 6 shows how these security concepts might be used in a security scheme for the sample database.

To establish a security scheme for a database, you use the SQL GRANT statement to specify which users have which privileges on which database objects. For example, here is a GRANT statement that lets Sam Clark retrieve and insert data in the OFFICES table of the sample database:

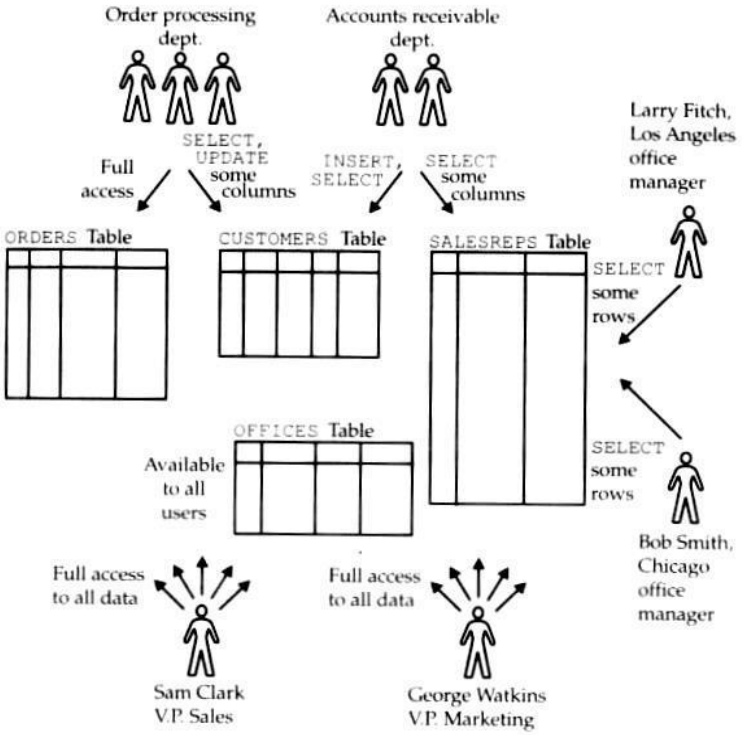


Fig. 6 : A security scheme for the sample database

Let Sam Clark retrieve and insert data in the OFFICES table.

```
GRANT SELECT, INSERT
      ON OFFICES
      TO SAM
```

The GRANT statement specifies a combination of a user−id (SAM), an object (the OFFICES table), and privileges (SELECT and INSERT). Once granted, the privileges can be rescinded later with this REVOKE statement:

Take away the privileges granted earlier to Sam Clark.

```
REVOKE SELECT, INSERT
       ON OFFICES
```

FROM SAM

The GRANT and REVOKE statements are described in detail later in this chapter, in the sections "Granting Privileges" and "Revoking Privileges."

User-Ids

Each user of a SQL-based database is typically assigned a user-id, a short name that identifies the user to the DBMS software. The user-id is at the heart of SQL security. Every SQL statement executed by the DBMS is carried out on behalf of a specific user-id. The user-id determines whether the statement will be permitted or prohibited by the DBMS. In a production database, user-ids are assigned by the database administrator. A personal computer database may have only a single user-id, identifying the user who created and who owns the database. In special-purpose databases (for example, those designed to be embedded within an application or a special-purpose system), there may be no need for the additional overhead associated with SQL security. These databases typically operate as if there were a single user-id.

In practice, the restrictions on the names that can be chosen as user-ids vary from implementation to implementation. The SQL1 standard permitted user-ids of up to 18 characters and required them to be valid SQL names. In some mainframe DBMS systems, user-ids may have no more than eight characters. In Sybase and SQL Server, user-ids may have up to 30 characters. If portability is a concern, it's best to limit user-ids to eight or fewer characters. Figure 7 shows various users who need access to the sample database and typical user-ids assigned to them. Note that all of the users in the order-processing department can be assigned the same user-id because they are to have identical privileges in the database.

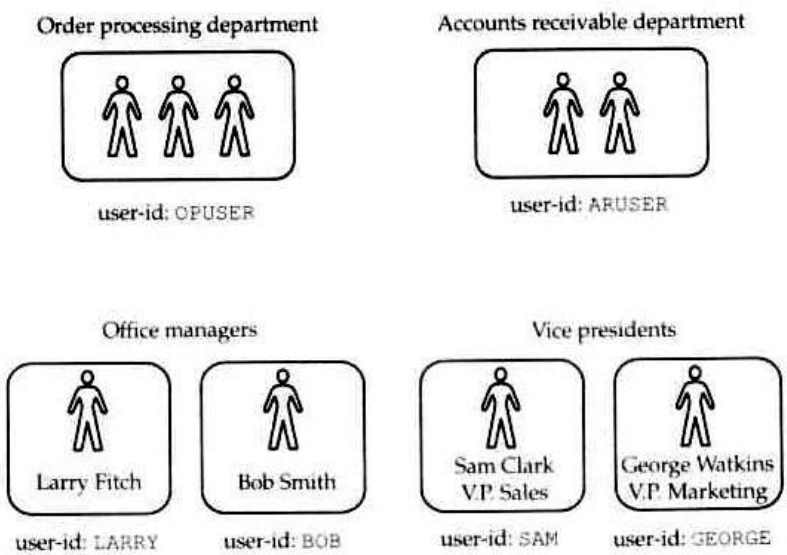


Fig. 7 : User-id assignments for the sample database

The ANSI/ISO SQL standard uses the term authorization-id instead of user-id, and you will occasionally find this term used in other SQL documentation. Technically, authorization-id is a more accurate term because the role of the ID is to determine authorization or privileges in the database. There are situations, as in Fig. 7, where it

makes sense to assign the same user-id to different users. In other situations, a single person may use two or three different user-ids. In a production database, authorization-ids may be associated with programs and groups of programs, rather than with human users. In each of these situations, authorization-id is a more precise and less confusing term than user-id. However, the most common practice is to assign a different user-id to each person, and most SQL-based DBMS use the term user-id in their documentation.

User Authentication

The SQL standard specifies that user-ids provide database security; however, the specific mechanism for associating a user-id with a SQL statement is outside the scope of the standard because a database can be accessed in many different ways. For example, when you type SQL statements into an interactive SQL utility, how does the DBMS determine which user-id is associated with the statements? If you use a forms-based data entry or query program, how does the DBMS determine your user-id? On a database server, a report-generating program might be scheduled to run at a preset time every evening; what is the user-id in this situation, where there is no human user? Finally, how are user-ids handled when you access a database across a network, where your user-id on the system where you are actively working might be different than the user-id established on the system where the database resides?

Most commercial SQL implementations establish a user-id for each database session. In interactive SQL, the session begins when you start the interactive SQL program, and it lasts until you exit the program. In an application program using programmatic SQL, the session begins when the application program connects to the DBMS, and it ends when the application program terminates. All of the SQL statements used during the session are associated with the user-id specified for the session.

Usually, you must supply both a user-id and an associated password at the beginning of a session. The DBMS checks the password to verify that you are, in fact, authorized to use the user-id that you supply. Although user-ids and passwords are common across most SQL products, the specific techniques used to specify the user-id and password vary from one product to another.

Some DBMS brands, especially those that are available on many different operating system platforms, implement their own user-id/password security. For example, when you use Oracle's interactive SQL program, called SQLPLUS, you specify a user name and associated password in the command that starts the program, like this:

```
SQLPLUS SCOTT / TIGER
```

The Sybase interactive SQL program, called ISOL, also accepts a user name and password, using this command format:

```
ISQL / USER=SCOTT / PASSWORD = TIGER
```

In each case, the DBMS validates the user-id (SCOTT) and the password (TIGER) before beginning the interactive SQL session.

Many other DBMS brands, including Ingres and Informix, use the user names of the host computer's operating system as database user-ids. For example, when you log in to a UNIX-based computer system, you must supply a valid UNIX user name and password to gain access. To start the Ingres interactive SQL utility, you simply give the command:

ISQL SALESDB

where SALESDB is the name of the Ingres database you want to use. Ingres automatically obtains your UNIX user name and makes it your Ingres user-id for the session. Thus, you don't have to specify a separate database user-id and password. DB2's interactive SQL, running under MVS/TSO, uses a similar technique. Your TSO login name automatically becomes your DB2 user-id for the interactive SQL session.

SQL security also applies to programmatic access to a database, so the DBMS must determine and authenticate the user-id for every application program that tries to access the database. Again, the techniques and rules for establishing the user-id vary from one brand of DBMS to another. For widely used utility programs, such as a data entry or an inquiry program, it is common for the program to ask the user for a user-id and password at the beginning of the session, via a screen dialog. For more specialized or custom-written programs, the appropriate user-id may be obvious from the application to be performed and hard-wired into the program.

Security Objects

SQL security protections apply to specific objects contained in a database The SQL1 standard specified two types of security objects—tables and views. Thus, each table and view can be individually protected. Access to a table or view can be permitted for certain user-ids and prohibited for other user-ids. The SQL2 standard expanded security protections to include other objects, including domains and user-defined character sets, and added a new type of protection for table or view access.

Most commercial SQL products support additional security objects. In a SQL Server database, for example, a stored procedure is an important database object. The SQL security scheme determines which users can create and drop stored procedures and which users are allowed to execute them. In IBM's DB2, the physical tablespaces where tables are stored are treated as security objects. The database administrator can give some user-ids permission to create new tables in a particular tablespace and deny that permission to other user-ids. Other SQL implementations support other security objects. However, the underlying SQL security scheme—of specific privileges applied to specific objects, granted or revoked through the same SQL statements—is almost universally applied.

Privileges

The set of actions that a user can carry out against a database object are called the privileges for the object. The SQL1 standard specifies four basic privileges for tables and views:

- The SELECT privilege allows you to retrieve data from a table or view. With this privilege, you can specify the table or view in the FROM clause of a SELECT statement or subquery.
- The INSERT privilege allows you to insert new rows into a table or view. With this privilege, you can specify the table or view in the t NTO clause of an INSERT statement.
- The DELETE privilege allows you to delete rows of data from a table or view. With this privilege, you can Specify the table or view in the FROM clause of a DELETE statement.
- The UPDATE privilege allows you to modify rows of data in a table or view. With this privilege, you can specify the table or view as the target table in an UPDATE statement. The UPDATE privilege can be restricted to specific columns of the table or view, allowing updates to these columns but disallowing updates to any other columns.

These four privileges are supported by virtually all commercial SQL products.

Views and SQL Security

In addition to the restrictions on table access provided by the SQL privileges, views also play a key role in SQL security. By carefully defining a view and giving a user permission to access the view but not its source tables, you can effectively restrict the user's access to only selected columns and rows. Views thus offer a way to exercise very precise control over what data is made visible to which users.

For example, suppose you wanted to enforce this security rule in the sample database: Accounts receivable personnel should be able to retrieve employee numbers, names, and office numbers from the SALESREPS table, but data about sales and quotas should not be available to them.

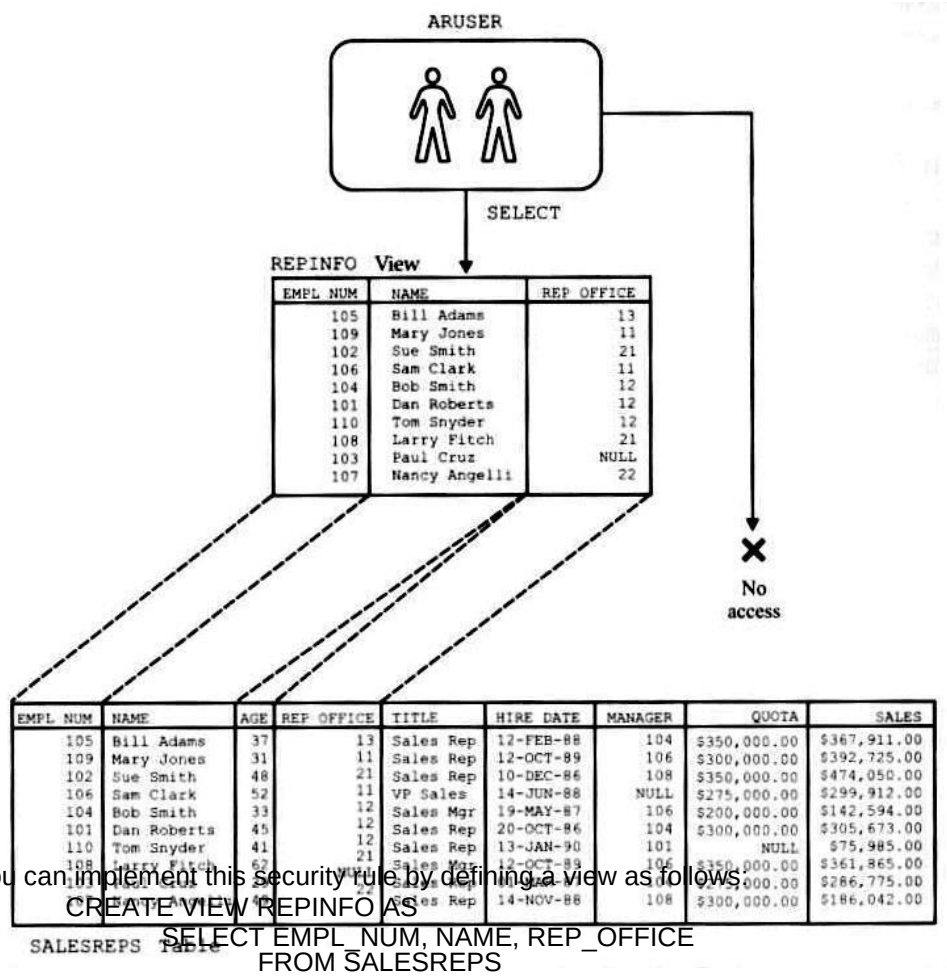


Fig. 8 : Using a view to restrict column access

You can implement this security rule by defining a view as follows:

```
CREATE VIEW REPINFO AS
    SELECT EMPL_NUM, NAME, REP_OFFICE
    FROM SALESREPS
```

and giving the SELECT privilege for the view to the ARUSER user-id, as shown in Fig.8. This example uses a vertical view to restrict access to specific columns.

Horizontal views are also effective for enforcing security rules such as this one:
The Sales managers in each region should have full access to SALESREPS data for the salespeople assigned to that region.

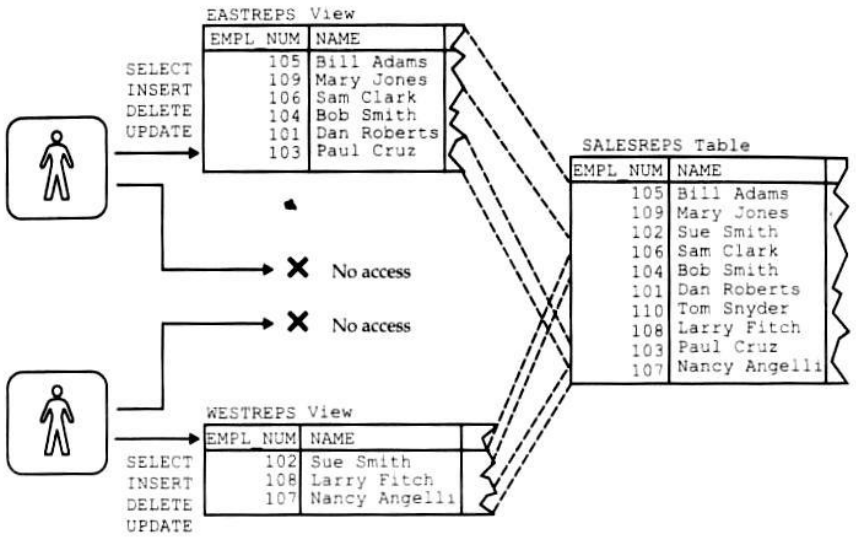


Fig. 9 : Using views to restrict row access

As shown in figure 9, you can define two views, EASTVIEWS and WESTVIEWS, containing SALESREPS data for each of the two regions, and then grant each office manager access to the appropriate view.

Of course, views can be much more complex than the simple row and column subsets of a single table shown in these examples. By defining a view with a grouped query, you can give a user access to summary data but not to the detailed rows in the underlying table. A view can also combine data from two or more tables, providing precisely the data needed by a particular user and denying access to all other data. The usefulness of views for implementing SQL security is limited by the two fundamental restrictions.

- **Update restrictions.** The SELECT privilege can be used with read-only views to limit data retrieval, but the INSERT, DELETE, and UPDATE privileges are meaningless for these views. If a user must update the data visible in a read-only view, the user must be given permission to update the underlying tables and must use INSERT, DELETE, and UPDATE statements that reference those tables.
- **Performance.** Because the DBMS translates every access to a view into a corresponding access to its source tables, views can add significant overhead to database operations. Views cannot be used indiscriminately to restrict database access without causing overall database performance to suffer.

Granting Privileges (GRANT)

The basic GRANT statement, shown in figure 10, is used to grant security privileges on database objects to specific users. Normally, the GRANT statement is used by the owner of a table or view to give other users access to the data. As shown in the figure, the

GRANT statement includes a specific list of the privileges to be granted, the name of the table to which the privileges apply, and the user-id to which the privileges are granted.

The GRANT statement shown in the syntax diagram conforms to the ANSI/ISO SQL standard. Many DBMS brands follow the DB2 GRANT statement syntax, which is more flexible. The DB2 syntax allows you to specify a list of user-ids and a list of tables, making it simpler to grant many privileges at once. Here are some examples of simple GRANT statements for the sample database:
Give order-processing users full access to the ORDERS table.

```
GRANT SELECT, INSERT, DELETE,
        UPDATE ON ORDERS
        TO OPUSER
```

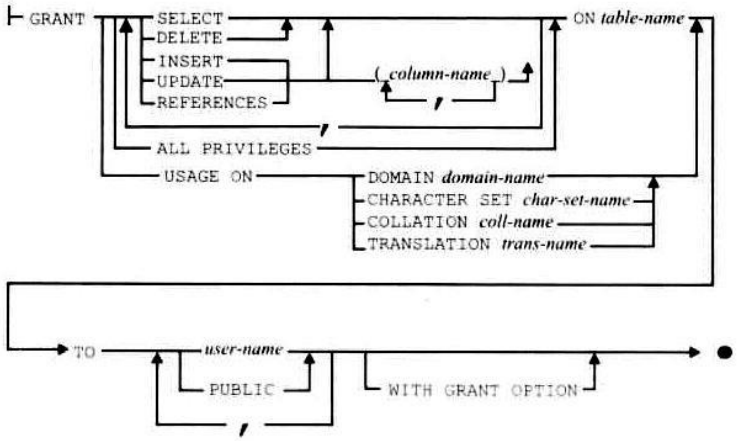


Fig. 10 : The GRANT statement syntax diagram

Let accounts receivable users retrieve customer data and add new customers to the CUSTOMERS table, but give order-processing users read-only access.

```
GRANT SELECT,  INSERT
    ON CUSTOMERS
    TO ARUSER

GRANT SELECT
        ON CUSTOMERS
        TO OPUSER
```

Allow Sam Clark to insert or delete an office.

```
GRANT INSERT, DELETE
        ON OFFICES
        TO SAM
```

For convenience, the GRANT statement provides two shortcuts that you can use when granting many privileges or when granting them to many users. Instead of specifically listing all of the privileges available for a particular object, you can use the keywords ALL PRIVILEGES. This GRANT statement gives Sam Clark, the vice president of sales, full access to the SALESREPS table:

Give all privileges on the SALESREPS table to Sam Clark.

```
GRANT ALL PRIVILEGES
        ON SALESREPS
        TO SAM
```

Instead of giving privileges to every user of the database one-by-one, you can use the keyword PUBLIC to grant a privilege to every authorized database user. This GRANT statement lets anyone retrieve data from the OFFICES table:

Give all users SELECT access to the OFFICES table.

```
GRANT SELECT
        ON OFFICES
        TO PUBLIC
```

Note that this GRANT statement grants access to all present and future authorized users, not just to the user-ids currently known to the DBMS. This eliminates the need for you to explicitly grant privileges to new users as they are authorized.

Column Privileges

The SQL1 standard allows you to grant the UPDATE privilege for individual columns of a table or view, and the SQL2 standard allows a column list for INSERT and REFERENCES privileges as well. The columns are listed after the UPDATE, INSERT, or REFERENCES keyword and enclosed in parentheses. Here is a GRANT statement that allows the order-processing department to update only the company name and assigned salesperson columns of the CUSTOMERS table:

Let order-processing users change company names and salesperson assignments.

```
GRANT UPDATE (COMPANY, CUST_REP)
        ON CUSTOMERS
        TO OPUSER
```

If the column list is omitted, the privilege applies to all columns of the table or view, as in this example:

Let accounts receivable users change any customer information.

```
GRANT UPDATE
    ON CUSTOMERS
    TO ARUSER
```

The ANSI/ISO standard does not permit a column list for the SELECT privilege; it requires that the SELECT privilege apply to all of the columns of a table or view. In practice, this isn't a serious restriction. To grant access to specific columns, you first define a view on the table that includes only those columns and then grant the SELECT privilege only for the view. However, views defined solely for security purposes can clog the structure of an otherwise simple database. For this reason, some DBMS brands allow a column list for the SELECT privilege. For example, the following GRANT statement is legal for the Sybase, SQL Server, and Informix DBMS brands:

Give accounts receivable users read-only access to the employee number, name, and sales office columns of the SALESREPS table.

```
GRANT SELECT (EMPL_NUM, NAME, REP_OFFICE)
        ON SALESREPS
```

```
        TO ARUSER
```

This GRANT statement eliminates the need for the REPINFO view defined in Fig. 8, and in practice, it can eliminate the need for many views in a production database However, the use of a column list for the SELECT privilege is unique to certain SQL dialects, and it is not permitted by the ANSI/ISO standard or by the IBM SQL products.

Passing Privileges (GRANT OPTION)

When you create a database object and become its owner, you are the only person who can grant privileges to use the object. When you grant privileges to other users, they are allowed to use the object, but they cannot pass those privileges on to other users. In this way, the owner of an object maintains very tight control both over who has permission to use the object and over which forms of access are allowed.

Occasionally, you may want to allow other users to grant privileges on an object that you own. For example, consider again the EASTREPS and WESTREPS views in the sample database. Sam Clark, the vice president of sales, created these views and owns them. He can give the Los Angeles office manager, Larry Fitch, permission to use the WESTREPS view with this GRANT statement:

```
GRANT SELECT
        ON WESTREPS
        TO LARRY
```

What happens if Larry wants to give Sue Smith (user-id SUE) permission to access the WESTREPS data because she is doing some sales forecasting for the Los Angeles office? With the preceding GRANT statement, he cannot give her the required privilege. Only Sam Clark can grant the privilege, because he owns the view.

If Sam wants to give Larry discretion over who may use the WESTREPS view, he can use this variation of the previous GRANT statement:

```
GRANT SELECT
        ON WESTREPS
        TO LARRY
WITH GRANT OPTION
```

Because of the WITH GRANT OPTION clause, this GRANT statement conveys, along with the specified privileges, the right to grant those privileges to other users.

Larry can now issue this GRANT statement:

```
GRANT SELECT
        ON WESTREPS
        TO SUE
```

which allows Sue Smith to retrieve data from the WESTREPS view. Figure 11 graphically illustrates the flow of privileges, first from Sam to Larry, and then from Larry to Sue. Because the GRANT statement issued by Larry did not include the WITH GRANT OPTION clause, the chain of permissions ends with Sue; she can retrieve the WESTREPS data but cannot grant access to another user. However, if Larry's grant of privileges to Sue had included the GRANT OPTION, the chain could continue to another level, allowing Sue to grant access to other users.

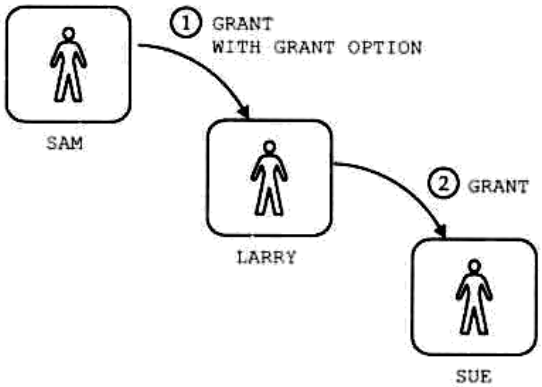


Fig. 11 : Using the GRANT OPTION

Alternatively, Larry might construct a view for Sue including only the salespeople in the Los Angeles office and give her access to that view:

```
CREATE VIEW LAREPS AS
        SELECT *
                FROM WESTREPS
                WHERE OFFICE = 21

GRANT ALL PRIVILEGES
        ON LAREPS
        TO SUE
```

Larry is the owner of the LAREPS view, but he does not own the WESTREPS view from which this new view is derived. To maintain effective security, the DBMS requires that Larry not only have the SELECT privilege on WESTREPS, but also requires that he have the GRANT OPTION for that privilege before allowing him to grant the SELECT privilege on LAREPS to Sue.

Once a user has been granted certain privileges with the GRANT OPTION, that user may grant those privileges and the GRANT OPTION to other users. Those other users can, in turn, continue to grant both the privileges and the GRANT OPTION. For this reason, you should use great care when giving other users the GRANT OPTION. Note that the GRANT OPTION applies only to the specific privileges named in the GRANT statement. If you want to grant certain privileges with the GRANT OPTION and grant other privileges without it, you must use two separate GRANT statements, as in this example :

Let Larry Fitch retrieve, insert, update, and delete data from the WESTREPS table, and let him grant retrieval permission to other users.

```
    GRANT SELECT
```

```
    ON WESTREPS
    TO LARRY
WITH GRANT OPTION
GRANT INSERT, DELETE, UPDATE
    ON WESTREPS
    TO LARRY
```

Revoking Privileges (REVOKE)

In most SQL-based databases, the privileges that you have granted with the GRANT statement can be taken away with the REVOKE statement, shown in figure 12. The REVOKE statement has a structure that closely parallels the GRANT statement, specifying a specific set of privileges to be taken away, for a specific database object, from one or more user−ids.

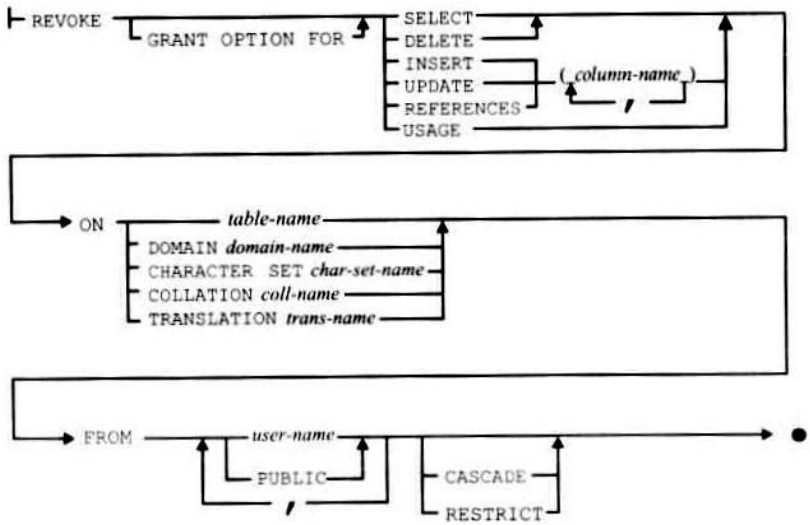


Fig. 12 : The REVOKE statement syntax diagram

A REVOKE statement may take away all or some of the privileges that you previously granted to a user−id. For example, consider this statement sequence :

Grant an then revoke some SALESREPS table privileges.

```
    GRANT SELECT, INSERT, UPDATE
    ON SALESREPS
    TO ARUSER, OPUSER

    REVOKE INSERT, UPDATE
        ON SALESREPS
        FROM OPUSER
```

The INSERT and UPDATE privileges on the SALESREPS table are first given to the two users and then revoked from one of them. However, the SELECT privilege remains for both user-ids. Here are some other examples of the REVOKE statement : Take away all privileges granted earlier on the OFFICES table.

```
    REVOKE ALL PRIVILEGES
```

```
        ON OFFICES
        FROM ARUSER
```

Take away UPDATE and DELETE privileges for two user−ids.

```
    REVOKE UPDATE, DELETE
        ON OFFICES
    FROM ARUSER, OPUSER
```

Take away all privileges on the OFFICES that were formerly granted to all users.

```
    REVOKE ALL PRIVILEGES
                ON OFFICES
        FROM PUBLIC
```

When you issue a REVOKE statement, you can take away only those privileges that you previously granted to another user. That user may also have privileges that were granted by other users; those privileges are not affected by your REVOKE statement. Note specifically that if two different users grant the same privilege on the same object to a user and one of them later revokes the privilege, the second user's grant will still allow the user to access the object. This handling of overlapping grants of privileges is illustrated in the following example sequence.

The sales vice president, gives Larry Fitch SELECT privileges for the SALESREPS table and SELECT and UPDATE privileges for the ORDERS table, using the following statements :

```
    GRANT SELECT
        ON SALESREPS
        TO LARRY

    GRANT SELECT, UPDATE
        ON ORDERS
        TO LARRY
```

A few days later George Watkins, the marketing vice president, gives Larry the SELECT and DELETE privileges for the ORDERS table and the SELECT privilege for the CUSTOMERS table, using these statements :

```
    GRANT SELECT, DELETE
        ON ORDERS
        TO LARRY

    GRANT SELECT
        ON CUSTOMERS
        TO LARRY
```

Note that Larry has received privileges on the ORDERS table from two different sources. In fact, the SELECT privilege on the ORDERS table has been granted by both sources. A few days later, Sam revokes the privileges he previously granted to Larry for the ORDERS table :

```
    REVOKE SELECT, UPDATE
        ON ORDERS
        FROM LARRY
```

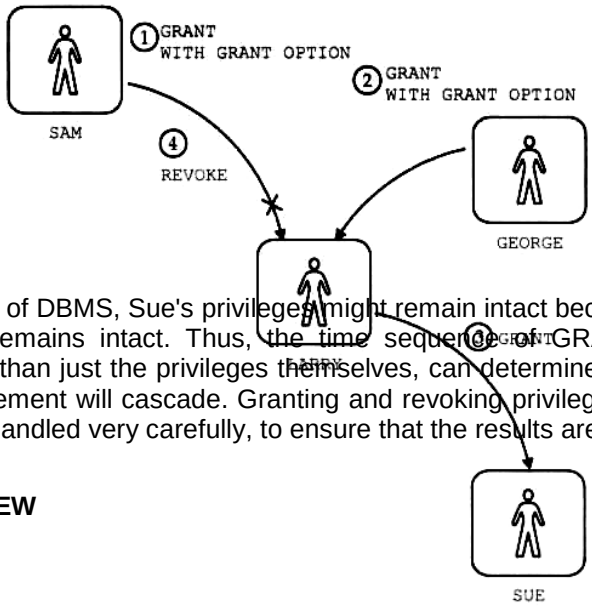
After the DBMS processes the REVOKE statement, Larry still retains the SELECT privilege on the SALESREPS table, the SELECT and DELECT privileges on the ORDERS table, and the SELECT privilege on the CUSTOMERS table, but he has lost the update privilege on the ORDERS table.

REVOKE and the GRANT OPTION

When you grant privileges with the GRANT OPTION and later revoke these privileges, most DBMS brands will automatically revoke all privileges derived from the original grant. Consider again the chain of privileges in figure 11, from Sam Clark, the sales vice president, to Larry Fitch, the Los Angeles office manager, and then to Sue Smith. If Sam now revokes Larry's privileges for the WESTREPS view, Sue's privilege is automatically revoked as well.

The situation gets more complicated if two or more users have granted privileges and one of them later revokes the privileges. Consider figure 13, a slight variation on the last example. Here, Larry receives the SELECT privilege with the GRANT OPTION from both Sam (the sales vice president) and George (the marketing vice president) and then grants privileges to Sue. This time when Sam revokes Larry's privileges, the grant of privileges from George remains. Furthermore, Sue's privileges also remain because they can be derived from George's grant.

However, consider another variation on the chain of privileges, with the events slightly rearranged, as shown in figure 14. Here, Larry receives the privilege with the GRANT OPTION from Sam, grants the privilege to Sue, and then receives the grant, with the GRANT OPTION, from George. This time when Sam revokes Larry's privileges, the results an- slightly different, and they may vary from one DBMS to another. As in figure 13, Larry retains the SELECT privilege on the WESTREPS view because the grant from George is still intact. But in a DB2 or SQL/DS database, Sue automatically loses her SELECT privilege on the table. Why? Because the grant from Larry to Sue was clearly derived from the grant from Sam to Larry, which has just been revoked. It could not have been derived from George's grant to Larry because that grant had not yet taken place when the grant from Larry to Sue was made.



In a different brand of DBMS, Sue's privileges might remain intact because the grant from George to Larry remains intact. Thus, the time sequence of GRANT and REVOKE statements, rather than just the privileges themselves, can determine how far the effects of a REVOKE statement will cascade. Granting and revoking privileges with the GRANT OPTION must be handled very carefully, to ensure that the results are those you intend.

UPDATING ON VIEW

Fig. 13 : Revoking privileges granted any two users

What does it mean to insert a row of data into a view, delete a row from a view, or update a row of a view? For some views, these operations can obviously be translated into equivalent operations against the source table(s) of the view. For example, consider once again the EASTREPS view, defined earlier in this chapter : Create a view showing Eastern region salespeople.

```
CREATE VIEW EASTREPS AS
        SELECT *
                FROM SALESREPS
                WHERE REP_OFFICE IN (11, 12, 13)
```

This is a straightforward horizontal view, derived from a single source table. As shown in Fig. 15, it makes sense to talk about inserting a row into this view; it means the new row should be inserted into the underlying SALESREPS table from which the view is derived. It also makes sense to delete a row from the EASTREPS view; this would delete the corresponding row from the SALESREPS table. Finally, updating a row of the EASTREPS view makes sense; this would update the corresponding row of the SALESREPS table. In each case, the action can be carried out against the corresponding row of the source table, preserving the integrity of both the source table and the view.

However, consider the ORD_BY_REP grouped view, as it was defined earlier in the section "Grouped Views":

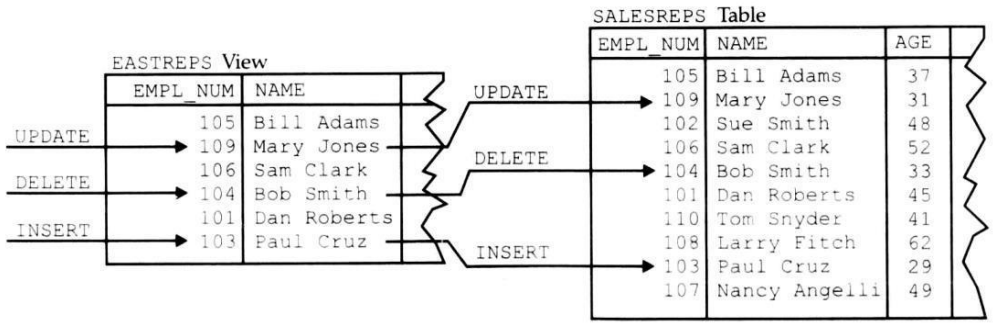


Fig. 15 : Updating data through a view

Define a view that contains summary order data for each salesperson.

```
CREATE VIEW ORD_BY_REP (WHO, HOW_MANY, TOTAL, LOW, HIGH,
AVERAGE) AS SELECT REP, COUNT (*), SUM(AMOUNT),
MIN(AMOUNT), MAX(AMOUNT),
        AVG(AMOUNT)
        FROM ORDERS
        GROUP BY REP
```

There is no one-to-one correspondence between the rows of this view and the rows of the underlying ORDERS table, so it makes no sense to talk about inserting, deleting, or updating rows of this view. The ORO_BY_REP view is not updateable; it is a read-only view.

The EASTREPS view and the ORD_BY_REP view are two extreme examples in term of the complexity of their definitions. There are views more complex than EASTREPS where it still makes sense to update the view, and there are views less complex than ORD_BY_REP

where updates do not make sense. In fact, which views can be updated and which cannot has been an important relational database research problem over the years.

View Updates and the ANSI/ISO Standard

The ANSI/ISO SQL1 standard specifies the views that must be updateable in a database that claims conformance to the standard. Under the standard, a view can be updated if the query that defines the view meets all of these restrictions:

- DISTINCT must not be specified; that is, duplicate rows must not be eliminated from the query results.
- The FROM clause must specify only one updateable table; that is, the view must have a single source table for which the user has the required privileges. If the source table is itself a view, then that view must meet these criteria.
- Each select item must be a simple column reference; the select list cannot contain expressions, calculated columns, or column functions.
- The WHERE clause must not include a subquery; only simple row-by-row search conditions may appear.
- The query must not include a GROUP BY or a HAVING clause.

The basic concept behind the restrictions is easier to remember than the rules themselves. **The CREATE TABLE Command**

The create Table command defines each column of the table uniquely. Each column has a minimum of three attributes, a name, datatype and size (i.e. column width). Each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the SQL statement is terminated with a semi colon.

Rules for Creating Tables

1) A name can have maximum upto 30 characters
2) Alphabets from A−Z, a−z and numbers from 0−9 are allowed
3) A name should begin with an alphabet
4) The use of the special character like_is allowed and also recommended. (Special characters like $, # are allowed only in Oracle).
5) SQL reserved words not allowed. For Example: create, select, and so on.

Syntax :

```
CREATE TABLE <TableName>
        (<ColumnName 1> <Data Type>(<size>), <Column Name 2>

        <Data Type>(<Size>)):
```

Note : Each column must have a datatype. The column should either be defined as null or not null and if this value is left blank, the database assumes “null” as the default.

A brief Checklist When Creating Tables

The following provides a small checklist for the issues that need to be considered before creating a table:

- What are the attributes of the rows to be stored?
- What are the data types of the attributes?
- Should varchar2 be used instead of char?

- Which columns should be used to build the primary key?
- Which columns do (not) allow null values? Which columns do / do not, allow duplicates?
- Are there defaults values for certain columns that also allow null values?

Example 1 : Create the BRANCH_MSTR table as shown in the Chapter 6 along with the structure for other table belonging to the Bank System.

```
CREATE TABLE "DBA_BANKSYS"."BRANCH_MSTR"( "BRANCH_NO"
        VARCHAR2(10),
        "NAME" VARCHAR2(25);
```

Output:
Table created.
Note : All table column belong to a single record. Therefore all the table column definitions are enclosed within parenthesis.

Inserting Data into Tables
Once a table is created, the most natural thing to do is load this with table with data to be manipulated later.

When inserting a single row of date into the table, the insert operation:
- Creates a new row (empty) in the database table
- Loads the values passed (by the SQL insert) into the columns specified

Syntax:

```
INSERT INTO <tablename> (<Columname 1>, <Columnname 2>)
        VALUES (<expression>, <expression2>):
```

Example 2 : Insert the values into the BRANCH_MSTR table (For values refer to 6th chapter under Test Records)

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('b1', 'Vile Parle (HO)'); INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('b2', 'Andheri'); INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('b3', 'Churchgate'); INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('b4', 'Sion'); INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('b5', 'Borivali'); INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('b6', 'Matunga');
```

Output for each of the above INSERT INTO statements:
1 row created.

Tip : Character expressions placed within the INSERT INTO statement must be enclosed in single quotes (').

In the INSERT INTO SQL sentence, table columns and values have a one to one relationship, (i.e. the first value described is inserted into the first columns, and the second value described is inserted into the second column and so on).

Hence, in an INSERT INTO SQL sentence if there are exactly the same numbers of values as there are columns and the values are sequences in exactly in accordance with the data type of the table columns, there is no need to indicate the column names.

However, if there are less values being described than there are columns in the table then it is mandatory to indicate both the table column name and its corresponding value in the INSERT INTO SQL sentence.

In the absence of mapping a table column name to a value in the INSERT INTO SQL sentence, the Oracle engine will not know which columns to insert the data into. This will generally cause a loss of data integrity. Then the data held within the table will be largely useless.

Viewing Data in the Tables

Once data has been inserted into a table, the next most logical operation would be to view what has been inserted. The SELECT SQL verb is used to achieve this. The SELECT command is used to retrieve rows selected from one or more tables.

- **All Rows and All Columns**

  In order to view global table data the syntax is :

  Select <ColumnName 1> TO <ColumnName N> FROM TableName;

  Here, ColumnName1 to ColumnName N represents table column names.

  Syntax :

  SELECT * FROM <TableName>;

Example 3 : Show all employee numbers, first name, middle name and last name who work in the bank.

```
SELECT EMP_NO, FNAME, MNAME, LNAME FROM EMP_MSTR;
```

Output :

EMP_NO	FNAME	MNAME	LNAME
E1	Ivan	Nelson	Bayross
E2	Amit		Desai
E3	Maya	Mahima	Joshi
E4	Peter	Iyer	Joseph
E5	Mandhar	Dilip	Dalvi
E6	Sonal	Abdul	Khan
E7	Anl	Ashutosh	Kambli
E8	Seema	P.	Apte
E9	Vikram	Vilas	Randive
E10	Anjali	Sameer	Pathak

10 riws sekected,

Example 4 : Show all the details related to the Fixed deposit Slab.

```
SELECT * FROM FDSLAB_MSTR;
```

FDSLAB_NO	MINPERIOD	MAXPERIOD	INTRATE
1	1	30	5
2	31	92	5.5
3	93	183	6
4	184	365	6.5
5	366	731	7.5
6	732	1097	8.5
7	1098	1829	10

7 rows selected.

Delete Operations
The DELETE command deletes rows from the table that satisfies the condition provided by its where clause, and returns the number of records deleted.

Caution : If a DELETE statement without a where clause is issued then all rows are deleted.

The verb DELETE in SQL is used to remove either:
- All the rows from a table

OR
- A set of rows from a table

Removal of All Rows
Syntax:
DELETE FROM <TableName>;

Example 5 : Empty the ACCT_DTLS table

DELETE FROM ACCT_DTLS

Output :
16 rows deleted.

Removal Of Specific Row(s)
Syntax:
DELETE FROM <Tablename> WHERE <Condition>;

Example 6 : Remove only the savings bank accounts details from the ACCT_DTLS table.

DELETE FROM ACCT_DTLS WHERE ACCT_NO LIKE ‘SB%;

Output:
6 rows deleted:

Removal of Specific Row(s) Based on the Data Held by other Tables
Sometimes it is desired to delete records in one table based on values in another table. Since it is not possible to list more than one table in the FROM clause while performing a delete, the EXISTS clause can be used.

Example 7 : Remove the address details of the customer named Ivan.

```
DELETE FROM ADDR_DTLS WHERE EXISTS(SELECT
        FNAME FROM CUST_MSTR
    WHERE CUST_MSTR.CUST_NO = ADDR−DTLS.CODE_NO
    AND CUST_MSTR.FNAME = 'Ivan');
```

Output:
1 row updated.

Modifying the Structure of Tables

The structure of a table can be modified by using the ALTER TABLE command. ALTER TABLE allows changing the structure of an existing table. With ALTER TABLE it is possible to add or delete columns, create or destroy indexes, change the data type of existing columns, or rename columns or the table itself.

ALTER TABLE work by making a temporary copy of the original table. The alteration is performed on the copy, then the orginal table is deleted and the new one is renamed. While ALTER TABLE is executing, the original table is still readable by users of Oracle.

UPDATES and writes to the table are stalled until the new table is ready, and then are automatically redirected to the new table without any failed updates.

Note : To use ALTER TABLE, the ALTER INSERT, and CREATE privileges for the table are required.

- **Adding New Columns** Syntax:

```
ALTER TABLE <TableName> ADD(<NewColumnName>
    <Datatype> (<Size>),
        <NewColumnName> <Datatype> (<Size>)….);
```

Example 8 : Enter a new field called City in the table BRANCH_MSTR. ALTER

```
TABLE BRANCH_MSTR ADD (CITY VARCHAR2(25));
```

Output:
Table altered.

- **Dropping a Column from a table** Syntax :

```
ALTER TABLE <TableName> DROP COLUMN <ColumnName>;
```

Example 9 : Drop the column city from the BRANCH_MSTR table.

```
ALTER TABLE BRANCH_MSTR DROP COLUMN CITY;
```

Output:
Table altered.

- **Modifying Existing Columns** Syntax:

```
Alter Table <TableName>
```

MODIFY (<ColumnName> <NewDatatype>(<NewSize>));

Example 10 : Alter the BRANCH_MSTR table to allow the NAME field to hold maximum of 30 characters.

```
ALTER TABLE BRANCH_MSTR MODIFY (NAME varchar2(30));
```

Output:
Table altered.

Restrictions on the **ALTER TABLE**
THE following tasks cannot be performed when using the ALTER TABLE clause:

- Change the name of the table
- Change the name of the column
- Decrease the size of a column if table data exists

Renaming Tables
Oracle allows renaming of tables. The rename operation is done atomically, which means that no other thread can access any of the tables while the rename process is running.

Note : To rename a table the ALTER and DROP privileges on the original table, and the CREATE and INSERT privileges on the new table are required.

To rename s table, the syntax is
Syntax :

```
RENAME <TableName> TO <NewTableName>
```

Example 11 : Change the name of branches table to branch table

```
RENAME BRANCH_MSTR TO BRANCHES;
```

Output:
Table renamed.

Truncating Tables
TRUNCATE TABLE empties a table completely. Logically, this is equivalent to a DELETE statement the deletes all rows, but there are practical difference under some circumstances.

TRUNCATE TABLE differs from DELETE in the following ways:

- Truncate operations drop and re−create the table, which is much faster than deleting rows one by one
- Truncate operations are not transaction−safe (i.e. an error will occur if ab active transaction or an active table lock exists)
- The number of deleted rows are not returned

Syntax:

```
TRUNCATE TABLE <TableName>
```

Example 12 : Truncate the table BRANCH_MSTR

TRUNCATE TABLE BRANCH_MSTR;

Output:
Table truncated.

Destroying Tables
Sometimes tables within a particular database become obsolete and need to be discarded. In such situation using the DROP TABLE statement with the table name can destroy a specific table.

Syntax :
DROP TABLE <TableName>

Caution : If a table is dropped all records held within it are lost and cannot be recovered.

Example 13 : Remove the table BRANCH_MSTR along with the data held.

DROP TABLE BRANCH_MSTR;

Output:
Table dropped.

SQL

DATA DEFINITION, AGG FUNCTIONS, NULL
VALUES THE DATA DEFINITION LANGUAGE

The SELECT, INSERT, DELETE, UPDATE, COMMIT, and ROLLBACK statements described in Parts II and III of this book are all concerned with manipulating the data in a database. These statements collectively are called the SQL Data Manipulation Language, or DML. The DML statements can modify the data stored in a database, but they cannot change its structure. None of these statements creates or deletes tables or columns, for example.

Changes to the structure of a database are handled by a different set of SQL statements, usually called the SQL Data Definition Language, or DDL. Using DDL statements, you can :

- Define and create a few table
- Remove a table that's no longer needed
- Change the definition of an existing table
- Define a virtual table (or view) of data
- Establish security controls for a database
- Build an index to make table access faster
- Control the physical storage of data by the DBMS

For the most part, the DPL statements insulate you from the low-level details of how data is physically stored in the database. They manipulate abstract database objects, such as tables and columns. However, the DDL cannot avoid physical storage issues entirely, and by necessity, the DDL statements and clauses that control physical storage vary from one DBMS to another.

The core of the Data Definition Language is based on three SQL verbs:

- CREATE. Defines and creates a database object
- DROP. Removes an existing database object
- ALTER. Changes the definition of a database object

In all major SQL-based DBMS products, these three DDL verbs can be used while the DBMS is running. The database structure is thus dynamic. The DBMS can be creating, dropping, or changing the definition of the tables in the database, for example, while it is simultaneously providing access to the database for its users. This is a major advantage of SQL and relational databases over earlier systems, where the DBMS had to be stopped before you could change the structure of the database. It means that a relational database can grow and change easily over time. Production use of a database can continue while new tables and applications are added.

Although the DDL and DML are two distinct parts of the SQL language, in most SQL-based DBMS products, the split is only a conceptual one. Usually, the DDL and DML statements are submitted to the DBMS in exactly the same way, and they can be freely intermixed in both interactive SQL sessions and programmatic SQL applications. If a program or user needs a table to store its temporary results, it can create the table, populate it, manipulate the data, and then delete the table. Again, this is a major advantage over earlier data models, in which the structure of the database was fixed when the database was created.

AGG FUNCTIONS

SQL lets your summarize data from the database through a set of column functions. A SQL column function takes an entire column of data as its argument and produces a single data item that summarizes the column. For example, the AVG ( ) column function takes a column of data and computes its average.

What re the average quota and average sales of our salespeople?

```
SALECT AVE (QUOTA), AVG (SALES)
    FROM SALESREPS
```

The argument to a column function can be a simple column name, as in the previous example, or it can be a SQL expression, as shown here :

What is the average quota performance of our salespeople?

```
SELECT AVE (10 * (SALES/QUOTA))
    FROM SALESREPS
```

```
AVG (100 * (SALES/QUOTA))
----------------------
                102.60
```

Computing a Column Total (Sum)

The SUM( ) column function computes the sum of a column of data values. The data in the column must have a numeric type (integer, decimal, floating point, or money). The result of the SUM( ) function has the same basic data type as the data in the column, but the result may have a higher precision.

What are the total quotas and sales for all salespeople?

```
SELECT SUM(QUOTA), SUM (SALES)
    FROM SALESREPS
```

Computing a Column Average (AVG)

The AVG ( ) column function computes the average of a column of data values. As with the SUM( ) function, the data in the column must have a numeric type.

Calculate the average price of products from manufacturer ACI.

```
SELECT AVG (PRICE)
    FROM PRODUCTS
  WHERE MFR_ID = 'ACI'
```

Finding Extreme Values (MIN and MAX)

The MIN ( ) and MAXIMUM ( ) column functions find the smallest and largest values in a column, respectively, The data in the column can contain numeric, string, or data/time information. The result of the MIN ( ) or MAX ( ) function has exactly the same data type as the data in the column.

What are the smallest and largest assigned quotas?

```
SELECT MIN (QUOTA) , MAX (QUOTA)
    FROM SALESREPS
```

What is the earliest order date in the database?

```
SELECT MIN (ORDER_DATE)
    FROM ORDERS
```

What is the best sales performance of any salesperson?

```
SELECT MAXIMUM (100 * (SALES/QUOTA))
    FROM SALESREPS
```

Counting Data Values (COUNT)

The COUNT ( ) column function counts the number of data values in a column. The data in the column can be of any type. The COUNT ( ) function always returns an integer, regardless of the data type of the column.

How many customers are there?

```
SELECT COUNT (CUST_NUM)
    FROM CUSTOMERS
```

How many salespeople are over quota?

```
SELECT COUNT (NAME)
    FROM SALESREPS
  WHERE SALES> QUOTA
```

SQL supports a special COUNT (*) column function, which counts rows rather than data values.

```
WHERE AMOUNT > 25000.00
```

Source: https://www.youtube.com/watch?v=xzoprhGAwxo

NULL VALUES

Because a database is usually a model of a real-world situation, certain pieces of data - are inevitably missing, unknown, or don't apply. In the sample database, for example, the QUOTA column in the SALESREPS table contains the sales goal for each salesperson. However, the newest salesperson has not yet been assigned a quota; this data is missing for that row of the table. You might be tempted to put a zero in the column for this salesperson, but that would not be an accurate reflection of the situation. The salesperson does not have a zero quota; the quota is just "not yet known."

Similarly, the MANAGER column in the SALESREPS fable contains the employee number of each salesperson's manager. But Sam Clark, the Vice President of Sales, has no manager in the sales organization. This column does not apply to Sam. Again, you might think about entering a zero, or a 9999 in the column, but neither of these values would really be the employee number of Sam's boss. No data value is applicable to this row.

SQL supports missing, unknown, or inapplicable data explicitly, through the concept of a null value. A null value is an indicator that tells SQL (and the user) that the data is missing or not applicable. As a convenience, a missing piece of data is often said to have the value NULL But the NULL value is not a real data value like 0,473.83, or "Sam Clark." Instead, it's a signal, or a reminder, that the data value is missing or unknown. Figure 16 shows the contents of the SALESREPS table. Note that the QUOTA and REP_OFFICE values for Tom Snyder's row and the MANAGER value for Sam Clark's row of the table all contain NULL values.
In many situations, NULL values require special handling by the DBMS. For example, if the user requests the sum of the QUOTA column, how should the DBMS handle the missing data when computing the sum? The answer is given by a set of special rules that govern NULL value handling in various SQL statements and clauses. Because of these

rules, some leading database authorities feel strongly that NULL values should not be used. Others, including Dr. Codd, have advocated the use of multiple NULL values, with distinct indicators for "unknown" and "not applicable" data.

Regardless of the academic debates, NULL values are a well-entrenched part of the ANSI/ISO SQL standard and are supported in virtually all commercial SQL products. They also play an important, practical role in production SQL databases. The special rules that apply to NULL values (and the cases where NULL values are handled inconsistently by various SQL products) are pointed out throughout this book.

SALESREPS Table

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	12-FEB-88	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	12-OCT-89	106	$300,000.00	$392,725.00
102	Sue Smith	48	21	Sales Rep	10-DEC-86	108	$350,000.00	$474,050.00
106	Sam Clark	52	11	VP Sales	14-JUN-88	NULL	$275,000.00	$299,912.00
104	Bob Smith	33	12	Sales Mgr	19-MAY-87	106	$200,000.00	$142,594.00
101	Dan Roberts	45	12	Sales Rep	20-OCT-86	104	$300,000.00	$305,673.00
110	Tom Snyder	41	NULL	Sales Rep	13-JAN-90	101	NULL	$75,985.00
108	Larry Fitch	62	21	Sales Mgr	12-OCT-89	106	$350,000.00	$361,865.00
103	Paul Cruz	29	12	Sales Rep	14-NOV-88	108	$275,000.00	$286,775.00
107	Nancy Angelli	49	22	Sales Rep	14-NOV-88	108	$300,000.00	$186,042.00

Value unknown

Value not applicable

Value unknown

Fig. 14: NULL values in the SALESREPS table

NESTED SUBQUERIES

The SQL subquery feature lets you use the results of one query as part of another query. The ability to use a query within a query was the original reason for the word "structured" in the name Structured Query Language. The subquery feature is less well known than SQL's join feature, but it plays an important role in SQL for three reasons :

- A SQL statement with a subquery is often the most natural way to express a query, because it most closely parallels the English-language description of the query.
- Subqueries make it easier to write SELECT statements, because they let you break a query down into pieces (the query and its subqueries) and then put the pieces back together.
- Some queries cannot be expressed in the SQL language without using a subquery.

The first several sections of this chapter describe subqueries and show how they are used in the WHERE and HAVING clauses of a SQL statement. The later sections of this chapter describe the advanced query expression capabilities that have been added to the SQL2 standard, which substantially expands the power of SQL to perform even the most complex of database operations.

Using Subqueries

A subquery is a query within a query. The results of the subquery arc used by the DBMS to determine the results of the higher-level query that contains the subquery. In the simplest forms of a subquery, the subquery appears within the WHERE or HAVING clause of another SQL statement. Subqueries provide an efficient, natural way to handle query requests that are themselves expressed in terms of the results of other queries. Here is an example of such a request :

List the offices where the sales target for the office exceeds the sum of the individual salespeople's quotas.

The request asks for a list of offices from the OFFICES table, where the value of the TARGET column meets some condition. It seems reasonable that the SELECT statement that expresses the query should look something like this :

```
SELECT CITY FROM OFFICES WHERE TARGET > ???
```

The value "???" needs to be filled in and should be equal to the sum of the quotas of the salespeople assigned to the office in question. How can you specify that value in the query ? You know that the sum f the quotas for a specific office (say, office number 21) can be obtained with this query :

```
SELECT SUM (QUOTA)
        FROM SALESREPS
WHERE REP_OFFICE = 21
```

But it would be inefficient to have to type in this query, write down the results, and then type in the previous query with the correct amount. How can you put the results of this query into the earlier query in place of the question marks? It would seem reasonable to start with the first query and replace the " ? ? ?" with the second query, as follows:

```
SELECT CITY
          FROM OFFICES
       WHERE TARGET > (SELECT SUM (QUOTA)
                          FROM SALESREPS
                          WHERE REP_OFFICE = OFFICE)
```

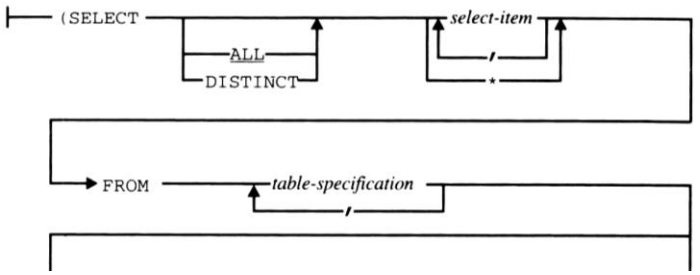
In fact, this is a correctly formed SQL query. For each office, the inner query (the subquery) calculates the sum of the quotas for the salespeople working in that office. The outer query (the main query) compares the office's target to the calculated total and decides whether to add the office to the main query results. Working together, the main query and the subquery express the original request and retrieve the requested data from the database.

SQL subqueries typically appear as part of the WHERE clause or the HAVING clause. In the WHERE clause, they help to select the individual rows that appear in tin results. In the HAVING clause, they help to select the row groups that appear in the query results.

What is a Subquery ?

Figure 17 shows the form of a SQL subquery. The subquery is enclosed in parentheses, but otherwise it has the familiar form of a SELECT statement, with a FROM clause and optional WHERE, GROUP BY, and HAVING clauses. The form of these clauses in a subquery is identical to that in a SELECT statement, and they perform their normal functions when used within a subquery. There are, however, a few differences between a subquery and an actual SELECT statement :

- In the most common uses, a subquery must produce a single column of data as its query results. This means that a subquery almost always has a single select item in its SELECT clause.



- The ORDER BY clause cannot be specified in a subquery. The subquery results are used internally by the main query and are never visible to the user, so it makes little sense to sort them anyway.
- Column names appearing in a subquery may refer to columns of tables in the main query. These outer references are described in detail later in the "Outer References" section.
- In most implementations, a subquery cannot be the UNION of several different SELECT statements; only a single SELECT is allowed. (The SQL2 standard allows much more powerful query expressions and relaxes this restriction, as described later in the section "Advanced Queries in SQL2.")

Subqueries in the WHERE Clause

Subqueries are most frequently used in the WHERE clause of a SQL statement. When a subquery appears in the WHERE clause, it works as part of the row selection process. The very simplest subqueries appear within a search condition and produce a value that is used to test the search condition. Here is an example of a simple subquery:

List the salespeople whose quota is less than 10 percent of the companywide sales target.

```
SALECT NAME
        FROM SALESREPS
WHERE QUOTA < ( .1 * (SELECT SUM (TARGET) FROM OFFICES ))

NAME
----------
Bob Smith
```

In this case, the subquery calculates the sum of the sales targets for all of the offices to determine the companywide target, which is multiplied by 10 percent to determine the cutoff sales quota for the query. That value is then used in the search condition to check each row of the SALESREPS table and find the requested names. In this simple case, the subquery produces the same value for every row of the SALESREPS table; the QUOTA value for each salesperson is compared to the same companywide number. In fact, you could carry out this query by first performing the subquery, to calculate the cutoff

quota amount ($275,000 in the sample database), and then carry out the main query using this number in a simple WHERE clause:

```
WHERE QUOTA < 275000
```

It's more convenient to use the subquery, but it's not essential. Usually, subqueries are not this simple. For example, consider once again the query from the previous section:

List the offices where the sales target for the office exceeds the sum of the individual salespeople's quotas.

```
SELECT CITY
    FROM OFFICES
    WHERE TARGET > (SELECT SUM (QUOTA)
                                                    FROM
SALESREPS
                                                    WHERE
REP_OFFICE = OFFICE)
CITY
----------
Chicago
Los Angeles
```

In this (more typical) case, the subquery cannot be calculated once for the entire query. The subquery produces a different value for each office, based on the quotas of the salespeople in that particular office. Figure 18 shows conceptually how SQL carries out the query. The main query draws its data from the OFFICES table, and the WHERE clause selects which offices will be included in the query results. SQL goes through the rows of the OFFICES table one by one, applying the test stated in the WHERE clause.

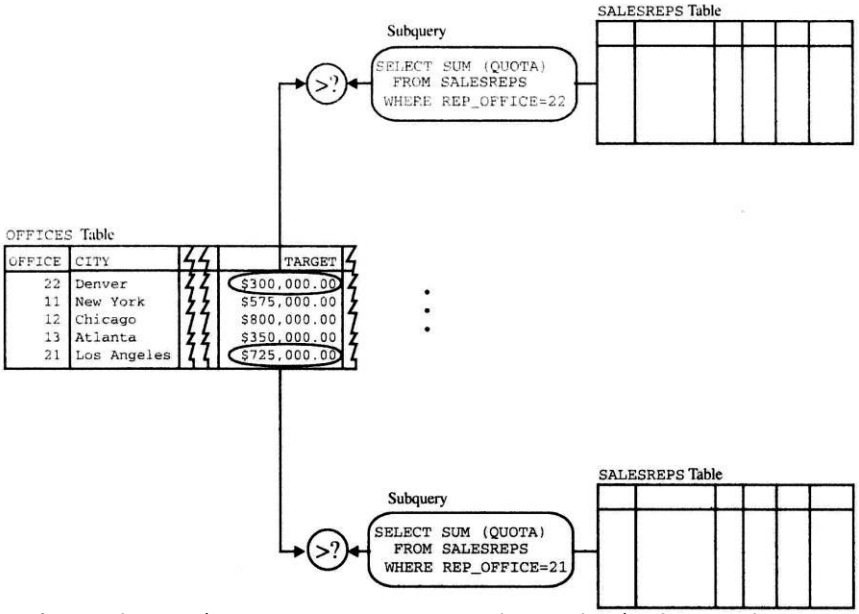


Fig.18 : Subquery comparesoperationthe invalueth ofWHEREtheTARGETclause column in the current row to the value produced by the subquery. To test the TARGET value, SQL carries out the subquery, finding the sum of the quotas for salespeople in the current office. The subquery produces a single number, and the WHERE clause compares the number to the TARGET value, selecting or rejecting the current office based on the comparison. As

the figure shows, SQL carries out the subquery repeatedly, once for each row tested by the WHERE clause of the main query.

Outer References

Within the body of a subquery, it's often necessary to refer to the value of a column in the current row of the main query. Consider once again the query from the previous sections:

List the offices where the sales target for the office exceeds the sum of the individual salespeople's quotas.

```
SELECT CITY
        FROM OFFICES
        WHERE TARGET > (SELECT SUM (QUOTA)
                                FROM SALESREPS
                                WHERE REP_OFFICE = OFFICE)
```

The role of the subquery in this SELECT statement is to calculate the total quota for those salespeople who work in a particular office—specifically, the office currently being tested by the WHERE clause of the main query. The subquery Joes this by scanning the SALESREPS table. But notice that the OFFICE column in the WHERE clause of the subquery doesn't refer to a column of the SALESREPS table; it refers to a column of the OFFICES table, which is a part of the main query. As SQL moves through each row of the OFFICES table, it uses the OFFICE value from the current row when it carries out the subquery.

The OFFICE column in this subquery is an example of an outer reference, which is a column name that does not refer to any of the tables named in the FROM clause of the subquery in which the column name appears. Instead, the column name refers to a column of a table specified in the FROM clause of the main query. As the previous example shows, when the DBMS examines the search condition in the subquery, the value of the column in an outer reference is taken from the row currently being tested by the main query.

Subquery Search Conditions

A subquery usually appears as part of a search condition in the WHERE or HAVING clause. The simple search conditions that can be used in these clauses. In addition, most SQL products offer these subquery search conditions:

- Subquery comparison test. Compares the value of an expression to a single value produced by a subquery. This test resembles the simple comparison test.
- Subquery set membership test. Checks whether the value of an expression matches one of the set of values produced by a subquery. This test resembles the simple set membership test.
- Existence test. Tests whether a subquery produces any rows of query results
- Quantified comparison test Compares the value of an expression to each of the set of values produced by a subquery.

The Subquery Comparison Test (=, <>, <, <=, >, >=)

The subquery comparison test is a modified form of the simple comparison test, as shown in figure 19. It compares the value of an expression to the value produced by a subquery and returns a TRUE result if the comparison is true. You use this test to a value from the row being tested to a single value produced by a subquery, as in this example :

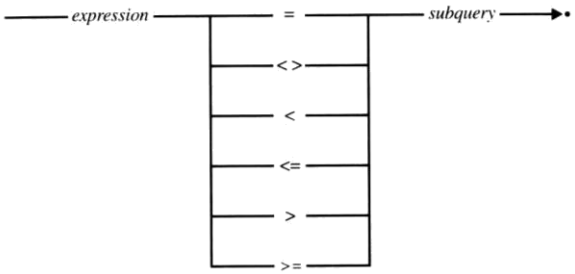


Fig. 19 : Subquery comparison test syntax diagram

List the salespeople whose quotas are equal to or higher than the target of the Atlanta sales office.

```
SELECT NAME
        FROM SALESREPS
WHERE QUOTA >= (SELECT TARGET
                        FROM OFFICES
                        WHERE CITY = 'Atlanta')

NAME
----------
Bill Adams
Sue Smith
Larry Fitch
```

The subquery in the example retrieves the sales target of the Atlanta office. The value is then used to select the salespeople whose quotas are higher than the retrieved target.

The subquery comparison test offers the same six comparison operators (=, <>, <, <=, >, >=) available with the simple comparison test. The subquery specified in this test must produce a single value of the appropriate data type—that is, it must produce a single row of query results containing exactly one column. If the subquery produces multiple rows or multiple columns, the comparison does not make sense, and SQL reports an error condition. If the subquery produces no rows or produces a NULL value, the comparison test returns NULL (unknown).

Here are some additional examples of subquery comparison tests:
List all customers served by Bill Adams.

```
SELECT COMPANY
        FROM CUSTOMERS
        WHERE CUST_REP = (SELECT EMPL_NUM
                                FROM SALESREPS
                                WHERE NAME = 'Bill Adams')
COMPANY
----------
Acme Mfg.
Three-Way Lines
```

List all products from manufacturer ACI where the quantity on hand is above the quantity on hand of product ACI-41004.

```
SELECT DESCRIPTION, QTY_ON_HAND
        FROM PRODUCTS
WHERE MFR_ID = 'ACI'
```

```
AND QTY_ON_HAND > (SELECT QTY_ON_HAND
                   FROM PRODUCTS
                  WHERE MFR_ID = 'ACI'
                    AND PRODUCT_ID = '41001')
```

DESCRIPTION	QTY_ON_HAND
Size 3 Widget	207
Size 1 Widget	277
Size 2 Widget	167

The subquery comparison test specified by the SQL1 standard and supported y all of the leading DBMS products allows a subquery only on the right side of the comparison operator. This comparison :

A < (subquery)

is allowed, but this comparison :

(subquery) > A

is not permitted. This doesn't limit the power of the comparison test, because the operator in any unequal comparison can always be turned around so that the subquery is put on the right side of the inequality. However, it does mean that you must sometimes turn around the logic of an English-language request to get a form of the request that corresponds to a legal SQL statement.

The SQL2 standard eliminated this restriction and allows the subquery to appear on either side of the comparison operator. In fact, the SQL2 standard goes considerably further and allows a comparison test to be applied to an entire row of values instead of a single value. This and other more advanced query expression features of the SQL2 standard are described in the latter sections of this chapter. However, they are not uniformly supported by the current versions of the major SQL products. For portability, it's best to write subqueries that conform to the SQL1 restrictions, as described previously.

The Set Membership Test (IN)

The subquery set membership test (IN) is a modified form of the simple set membership test, as shown in Fig. 20. It compares a single data value to a column of data values produced by a subquery and returns a TRUE result if the data value matches one of the values in the column. You use this test when you need to compare a value from the row being tested to a set of values produced by a subquery. Here is a simple example:

List the salespeople who work in offices that are over target.

```
SELECT NAME
    FROM SALESREPS
    WHERE REP_OFFICE IN (SELECT OFFICE
                         FROM OFFICES
                         WHERE SALES > TARGET)

NAME
---------
Mary Jones
Sam Clark
Bill Adams
```

Sue Smith
Larry Fitch

The subquery produces a set of office numbers where the sales are above target. (In the sample database, there are three such offices, numbered 11, 13, and 21.) The main query then checks each row of the SALESREPS table to determine whether that particular salesperson works in an office with one of these numbers. Here are some other examples of subqueries that test set membership:

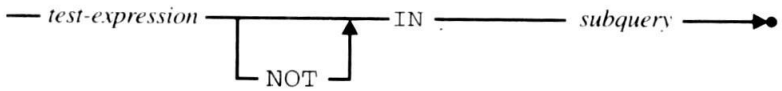


Fig. 20 : Subquery set membership test (IN) syntax diagram

List the salespeople who do not work in offices managed by Larry Fitch (employee 108).

```
SELECT NAME
        FROM SALESREPS
WHERE REP_OFICE NOT IN (SELECT OFFICE
                                                    FROM
OFFICES
                                                WHERE MGR =
108)
NAME
----------
Bill Adams
Mary Jones
Sam Clark
Bob Smith
Dan Roberts
Paul Cruz
```

List all of he customers who have placed orders for ACI Widgets (manufacturer ACI, product numbers starting with 4100) between January and June 1990.

```
SELECT COMPANY
        FROM CUSTOEMRS
WHERE CUST_NUM IN (SELECT DISTINCT CUST
                                                FROM
ORDERS
                                            WHERE MFR = 'ACI'
                                                AND
PRODUCT LIKE '4100%'
                                                AND
ORDER_DATE BETWEEN '01-JAN-90'

                AND '30-JUN-90')

COMPANY
------------
Acme Mfg.
Ace International
Holm & Landis
```

JCP Inc.

In each of these examples, the subquery produces a column of data values, and the WHERE clause of the main query checks to see whether a value from a row of the main query matches one of the values in the column. The subquery form of the IN test thus works exactly like the simple IN test, except that the set of values is produced by a subquery instead of being explicitly listed in the statement.

The Existence Test (EXISTS)

The existence test (EXISTS) checks whether a subquery produces any rows of query results, as shown in figure 21. There is no simple comparison test that resembles the existence test; it is used only with subqueries.

Here is an example of a request that can be expressed naturally using an existence test :

List the products for which an order of $25,000 or more has been received.

The request could easily be rephrased as:

List the products for which there exists at least one order in the ORDERS table (a) that is for the product in question and (b) that has an amount of at least $25,000.

The SELECT statement used to retrieve the requested list of products closely resembles the rephrased request :

```
SELECT DISTINCT DESCRIPTION
        FROM PRODUCTS
WHERE EXISTS ( SELECT ORDER_NUM
                                FROM ORDERS
                        WHERE PRODUCT = PRODUCT_ID
                                AND MFR = MFR_ID
                                AND AMOUNT >= 25000.00 )
```

```
DESCRIPTION
−−−−−−−−−−−−−−
500−1B Brace
Left Hinge
Right Hinge
Widget Remover
```

Conceptually, SQL processes this query by going through the PRODUCTS table and performing the subquery for each product. the subquery produces a column containing the order numbers of any orders for the “current” product that are over $25,000. If there are any such orders (that is, if the column is not empty), the EXISTS test is TRUE. If the subquery produces no rows, the EXISTS test is FALSE. The EXISTS test cannot produce a NULL value.

Fig. 21 : Existence test (EXISTS) syntax diagram

You can reverse the logic of the EXISTS test using the NOT EXISTS form. In this case, the test is TRUE if the subquery produces no rows, and FALSE otherwise.

Notice that the EXISTS search condition doesn't really use the results of the subquery at all. It merely tests to see whether the subquery produces any results. For this reason, SQL relaxes the rule that "subqueries must return a single column of data" and allows you to use the SELECT * form in the subquery of an EXISTS test. The previous subquery could thus have been written :

List the products for which an order of $25,000 or more has been received.

```
SELECT DESCRIPTION
        FROM PRODUCTS
        WHERE EXISTS (SELECT *
                                                FROM ORDERS
                                                WHERE  PRODUCT  =
PRODUCT_ID
                                                AND MFR = MFR_ID
                                                AND    AMOUNT   >=
25000.00)
```

In practice, the subquery in an EXISTS test is always written using the SELECT * notation. Here are some additional examples of queries that use EXISTS :

List any customers assigned to Sue Smith who have not placed an order for over $3000.

```
SELECT COMPANY
        FROM CUSTOMERS
WHERE CUST_REP = ( SELECT EMPL_NUM
                                                FROM SALESREPS
                                                WHERE NAME = 'Sue
Smith')
        AND NOT EXISTS (SELECT *
                                                FROM ORDERS
                                                WHERE    CUST    =
CUST_NUM
                                                AND    AMOUNT    >
3000.00 )
        COMPANY
----------------
Carter & Sons
Fred Lewis Corp.
```

List the offices where there is a salesperson whose quota represents more than 55 percent of the office's target.

```
SELECT CITY
        FROM OFFICES
WHERE EXISTS (SELECT *
                                        FROM SALESREPS
                                        WHERE REP_OFFICE = OFFICE
                                                AND QUOTA > (.55 * TARGET
))
CITY
-----------
```

Denver
Atlanta

Note that in each of these examples, the subquery includes an outer reference to a column of the table in the main query. In practice, the subquery in an EXISTS test will always contain an outer reference that links the subquery to the row currently being tested by the main query.

Quantified Tests (ANY and ALL) *

The subquery version of the IN test checks whether a data value is equal to some value in a column of subquery results. SQL provides two quantified tests, ANY and ALL, that extend this notion to other comparison operators, such as greater than (>) and less than (<). Both of these tests compare a data value to the column of data values produced by a subquery, as shown in figure 22.

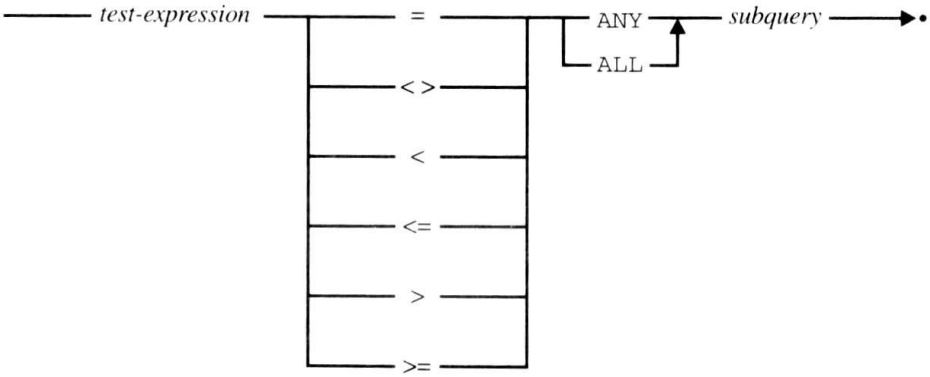


Fig. 22: Quantified comparison tests (ANY and ALL) syntax diagrams

The ANY Test *

The ANY test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value to a column of data values produced by a subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column, one at a time. If any of the individual comparisons yield a TRUE result, the ANY test returns a TRUE result. Here is an example of a request that can be handled with the ANY test:

List the salespeople who have taken an order that represents more than 10 percent of their quota.

```
SELECT NAME
        FROM SALESREPS
        WHERE ( .1 * QUOTA) < ANY ( SELECT AMOUNT

        FROM ORDERS

        WHERE REP = EMPL_NUM )
NAME
------------
Sam clark
```

Larry Fitch
Nancy Angelli

Conceptually, the main query tests each row of the SALESREPS table, one by one. The subquery finds all of the orders taken by the current salesperson and returns a column containing the order amounts for those orders. The WHERE clause of the main query then computes 10 percent of the current salesperson's quota and uses it as a test value, comparing it to every order amount produced by the subquery. If any order amount exceeds the calculated test value, the ANY test returns TRUE, and the salesperson is included in the query results. If not, the salesperson is not included in the query results. The keyword SOME is an alternative for ANY specified by the ANSI/ISO SQL standard. Either keyword can generally be used, but some DBMS brands do not support SOME.

The ANY test can sometimes be difficult to understand because it involves an entire set of comparisons, not just one. It helps if you read the test in a slightly different way than it appears in the statement. If this ANY test appears:

WHERE X < ANY (SELECT Y …)

instead of reading the test line this :

"where X is less than any select

Y…" try reading it like this :

"where, for some Y, X is less than Y"

When you use this trick, the preceding query becomes :

Select the salespeople where, for some order taken by the salesperson, 10 percent of the salesperson's quota is less than the order amount.

If the subquery in an ANY test produces no rows of query results, or if the query results include NULL values, the operation of the ANY test may vary from one DBMS to another. The ANSI/ISO SQL standard specifies these detailed rules describing the results of the ANY test when the test value is compared to the column of subquery results:

- If the subquery produces an empty column of query results, the ANY test returns FALSE—there is no value produced by the subquery for which the comparison test holds.
- If the comparison test is TRUE for at least one of the data values in the column, then the ANY search condition returns TRUE—there is indeed some value produced by the subquery for which the comparison test holds.
- If the comparison test is FALSE for every data value in the column, then the ANY search condition returns FALSE. In this case, you can conclusively state that there is no value produced by the subquery for which the comparison test holds.
- If the comparison test is not TRUE for any data value in the column, but it is NULL (unknown) for one or more of the data values, then the ANY search condition returns NULL. In this situation, you cannot conclusively state whether there is a value produced by the subquery for which the comparison test holds; there may or may not be, depending on the "actual" (but currently unknown) values for the NULL data.

The ANY comparison operator can be very tricky to use in practice, especially in conjunction with the inequality (<>) comparison operator. Here is an example that shows the problem:

List the names and ages of all the people in the sales force who do not manage an office. It's tempting to express this query as shown in this example:

```
SELECT NAME, AGE
    FROM SALESREPS
    WHERE EMPL_NUM <> ANY  (SELECT MGR
                                                    FROM
    OFFICES)
```

The subquery :

```
SELECT MGR
        FROM OFFICES
```

obviously produces the employee numbers of the managers, and therefore the query seems to be saying :

Find each salesperson who is not the manager of any office.
but that's not what the query says ! What it does say is this :
Find each salesperson who, for some office, is not the manager of that office.

Of course for any given salesperson, it's possible to find some office where that salesperson is not the manager. The query results would include all the salespeople and therefore fail to answer the question that was posed! The correct query is :

```
SELECT NAME, AGE
    FROM SALESREPS
WHERE NOT (EMPL_NUM = ANY (SELECT MGR FROM OFFICES))
```

NAME	**AGE**
Mary Jones	31
Sue Smith	48
Dan Roberts	45
Tom Snyder	41
Paul Cruz	29
Nancy Angelli	49

You can always turn a query with an ANY test into a query with an EXISTS test by moving the comparison inside the search condition of the subquery. This is usually a very good idea because it eliminates errors like the one just described. Here is an alternative form of the query, using the EXISTS test :

```
SELECT NAME, AGE
FROM SALESREPS
WHERE NOT EXISTS (SELECT * FROM OFFICES WHERE EMPL_NUM = MGR)
```

NAME	AGE
Mary Jones	31
Sue Smith	48
Dan Roberts	45
Tom Snyder	41
Paul Cruz	29
Nancy Angelli	49

The ALL Test *

Like the ANY test, the ALL test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value to a column of data values produced by a subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column, one at a time. If all of the individual comparisons yield a TRUE result, the ALL test returns a TRUE result.

Here is an example of a request that can be handled with the ALL test:
List the offices and their targets where all of the salespeople have sales that exceed 50 percent of the office's target.

```
SELECT CITY , TARGET
        FROM OFFICES
WHERE ( .50 * TARGET) < ALL ( SELECT SALES

        FROM SALESREPS

        WHERE REP_OFFICE = OFFICE )
```

CITY	TARGET
Denver	$300,000.00
New York	$575,000.00
Atlanta	$350,000.00

Conceptually, the main query tests each row of the OFFICES table, one by one. The subquery finds all of the salespeople who work in the current office and returns a column containing the sales for each salesperson. The WHERE clause of the main query then computes 50 percent of the office's target and uses it as a test value, comparing it to every sales value produced by the subquery. If all of the sales values exceed the calculated test value, the ALL test returns TRUE, and the office is included in the query results. If not, the office is not included in the query results.

Like the ANY test, the ALL test can be difficult to understand because it involves an entire set of comparisons, not just one. Again, it helps if you read the test in a slightly different way than it appears in the statement. If this ALL test appears:

```
WHERE X < ALL (SELECT Y …)
```

instead of reading it like this :

"where X is less than all select Y…"

try reading the test like this :

"where, for all Y, X is less than Y"

When you use this trick, the preceding query becomes :
Select the offices where, for all salespeople who work in the office, 50 percent of the office's target is less than the salesperson's sales.

If the subquery in an ALL test produces no rows of query results, or if the query results include NULL values, the operation of the ALL test may vary from one DBMS to another. The ANSI/ISO SQL standard specifies these detailed rules describing the results of the ALL test when the test value is compared to the column of subquery results:

- If the subquery produces an empty column of query results, the ALL test returns TRUE. The comparison test does hold for every value produced by the subquery; there just aren't any values.
- If the comparison test is TRUE for every data value in the column, then the ALL search condition returns TRUE. Again, the comparison test holds true for every value produced by the subquery.
- If the comparison test is FALSE for any data value in the column, then the ALL search condition returns FALSE. In this case, you can conclusively state that the comparison test does not hold true for every data value produced by the query.
- If the comparison test is not FALSE for any data value in the column, but it is NULL for one or more of the data values, then the ALL search condition returns NULL. In this situation, you cannot conclusively state whether there is a value produced by the subquery for which the comparison test does not hold true-there may or may not be, depending on the "actual" (but currently unknown) values for the NULL data.

The subtle errors that can occur when the ANY test is combined with the inequality (<>) comparison operator also occur with the ALL test. As with the ANY test, the ALL test can always be converted into an equivalent EXISTS test by moving the comparison inside the subquery.

Subqueries and Joins

You may have noticed as you read through this chapter that many of the queries that were written using subpueries could also have been written as multi table queries, or joins. This is often the case, and SQL allows you to write the query either way. This example illustrates the point :

List the names and ages of salespeople who work in offices in the Western region.

```
SELECT NAME, AGE
        FROM SALESREPS
    WHERE REP_OFFICE IN (SELECT OFFICE
                                FROM OFFICES
                                WHERE REGION = 'Western')
```

NAME	AGE
Sue Smith	48
Larry Fitch	62
Nancy Angelli	49

This form of the query closely parallels the stated request. The subquery yields a list of offices in the Western region, and the main query finds the salespeople who work in one of the offices in the list. Here is an alternative form of the query, using a two-table join :

List the names and ages of salespeople who work in offices in the Western region.

```
SELECT NAME, AGE
        FROM SALESREPS, OFFICES
        WHERE REP_OFFICE = OFFICE
            AND REGION = 'Western'
```

NAME	AGE
Sue Smith	48

Larry Fitch	62
Nancy Angelli	49

This form of the query joins the SALESREPS table to the OFFICES table to find the region where each salesperson works, and then eliminates those who do not work in the Western region.

Either of the two queries will find the correct salespeople, and neither one is right or wrong. Many people will find the first form (with the subquery) more natural, because the English request doesn't ask for any information about offices, and because it seems a little strange to join the SALESREPS and OFFICES tables to answer the request. Of course if the request is changed to ask for some information from the OFFICES table:

List the names and ages of the salespeople who work in offices in the Western region and the cities where they work.
The subquery form will no longer work, and the two-table query must be used. Conversely, many queries with subqueries cannot be translated into an equivalent join. Here is a simple example:

List the names and ages of salespeople who have above average quotas.

```
SELECT NAME, AGE
  FROM SALESREPS
  WHERE QUOTA > ( SELECT AVG (QUOTA)
                                    FROM SALESREPS )
```

NAME	AGE
Bill Adams	37
Sue Smith	48
Larry Fitch	62

In this case, the inner query is a summary query and the outer query is not, so there is no way the two queries can be combined into a single join.

Nested Subqueries

All of the queries described thus far in this chapter have been two-level queries, involving a main query and a subquery. Just as you can use a subquery inside a main query, you can use a subquery inside another subquery. Here is an example of a request that is naturally represented as a three-level query, with a main query, a subquery, and a sub-subquery:

List the customers whose salespeople are assigned to offices in the Eastern sales region.

```
SALECT COMPANY
        FROM CUSTOMERS
WHERE CUST_REP IN (SELECT EMPL_NUM
                                    FROM SALESREPS
                                    WHERE  REP_OFFICE
IN (SELECT OFFICE
                                FROM OFFICES
```

```
                    WHERE REGION = ‘eastern’))
    COMPANY
    −−−−−−−−−−−−−−−−
    First corp.
    Smithson Corp.
    AAA  Investments
    JCP Inc.
    Chen Associates
    QMA Assoc.
    Ian & Schmidt
    Acme Mfg.
```

In this example, the innermost subquery :

```
    SELECT OFFICE
        FROM OFFICES
      WHERE REGION = ‘Eastern’
```

produces a column containing the office numbers of the offices in the Eastern region. the next subquery:

```
    SELECT EMPL_NUM
        FROM SALESREPS
    WHERE REP_OFFICE IN (subquery)
```

produces a column containing the employee numbers of the salespeople who work in one of the selected offices. Finally, the outermost query :

```
    SELECT COMPANY
        FROM CUSTOMERS
    WHERE CUST_REP IN (subquery)
```

finds the customers whose salespeople have one of the selected employee numbers.

The same technique used in this three−level query can be used to build queries with four or more levels. The ANSI/ISO SQL standard does not specify a maximum number of nesting levels, but in practice, a query becomes much more time−consuming as the number of levels increases. The query also becomes more difficult to read , understand, and maintain when it involves more than one or two levels of subqueries. Many SQL implementations restrict the number of subquery levels to a relatively small number.

Correlated Subqueries *

In concept, SQL performs a subquery over and over again once for each row of the main query. For many subqueries, however, the subquery produces the same results for every row or row group. Here is an example ;

List the sales offices whose sales are below the average target.

```
    SELECT CITY
        FROM OFFICES
        WHERE SALES < (SELECT AVE (TARGET)
                                        FROM OFFICES )
    CITY
    −−−−−−−−−−
    Denver
```

Atlanta

In this query, it would be silly to perform the subquery five times (once for each office). The average target doesn't change with each office; it's completely independent of the office currently being tested. As a result, SQL can handle the query by first performing the subquery, yielding the average target ($550,000), and then converting the main query into :

```
SELECT CITY
        FROM OFFICES
WHERE SALES < 550000.00
```

Commercial SQL implementations automatically detect this situation and use this shortcut whenever possible to reduce the amount of processing required by a subquery. However, the shortcut cannot be used if the subquery contains an outer reference, as in this example :

List all of the offices whose targets exceed the sum of the quotas of the salespeople who work in them:

```
        SELECT CITY
                FROM OFFICES
                WHERE TARGET > ( SELECT SUM (QUOTA)
                                                        FROM
SALESREPS
                                                        WHERE
REP_OFFICE = OFFICE)
CITY
----------------
Chicago
Los Angeles
```

For each row of the OFFICES table to be tested by the WHERE clause of the main query, the OFFICE column (which appears in the subquery as an outer reference) has a different value. Thus, SQL has no choice but to carry out this subquery five times—once for each row in the OFFICES table. A subquery containing an outer reference is called a correlated subquery because its results are correlated with each individual row of the main query. For the same reason, an outer reference is sometimes called a correlated reference.

List the salespeople who are over 40 and who manage a salesperson over quota.

```
SELECT NAME
        FROM SALESREPS
WHERE AGE > 40
        AND EMPL_NUM IN ( SELECT MANAGER
                                                        FROM
SALESREPS
                                                WHERE   SALES   >
QUOTA )
NAME
----------
Sam Clark
Larry Fitch
```

The MANAGER. QUOTA, and SALES columns in the subquery are references to the SALESREPS table in the subquery's own FROM clause; SQL does not interpret them as outer references, and the subquery is not a correlated subquery. SQL can perform the subquery first in this case, finding the salespeople who are over quota and generating a list of the employee numbers of their managers. SQL can then turn its attention to the main query, selecting managers whose employee numbers appear in the generated list.

If you want to use an outer reference within a subquery like the one in the previous example, you must use a table alias to force the outer reference. This request, which adds one more qualifying condition to the previous one, shows how:

List the managers who are over 40 and who manage a salesperson who is over quota and who does not work in the same sales office as the manager.

```
SELECT NAME
        FROM SALESREPS MGRS
WHERE AGE > 40
        AND MGRS. EMPL_NUM IN (SELECT MANAGER

        FROM SALESREPS EMPS

        WHERE EMPS.QUOTA > EMPS.SALES
                                                AND
EMPS.REP_OFFICE <> MGRS.REP_OFFICE)
NAME
----------
Sam Clark
Larry Fitch
```

The copy of the SALESREPS table used in the main query now has the tag MGRS, and the copy in the subquery has the tag EMPS. The subquery contains one additional search condition, requiring that the employee's office number does not match that of the manager. The qualified column name MGRS. OFFICE in the subquery is an outer reference, and this subquery is a correlated subquery.

What Is a Trigger?

The concept of a trigger is relatively straightforward. For any event that causes a change in the contents of a table, a user can specify an associated action that the DBMS should carry out. The three events that can trigger an action triggered by an event is specified by a sequence of SQL statements.

To understand how a trigger works, let’s examine a concrete example. When a new order is added to the ORDERS table, these two changes to the database should also take place :

- The SALES column for the salesperson who took the order should be increased by the amount of the order.
- The QTY_ON_HAND amount for the product being ordered should be decreased by the quantity ordered.

This Transact-SQL statement defines a SQL Server trigger, named NEWORDER, that causes these database updates to happen automatically :

```
CREATE TRIGGER NEWORDER
```

```
        ON ORDERS
        FOR INSERT
AS UPDATE SALESREPS
        SET SALES = SALES + INSERTED.AMOUNT
        FROM SALESREPS, INSERTED
    WHERE SALESREPS.EMPL_NUM = INSERTED.REP
    UPDATE PRODUCTS
    SET QTY_ON_HAND = QTY_ON_HAND − INSERTED.QTY
    FROM PRODUCTS, INSERTED
    WHERE PRODUCTS.MFR_ID = INSERTED.MFR
    AND PRODUCTS.PRODUCT_ID = INSERTED.PRODUCT
```

The first part of the trigger definition tells SQL server that the trigger is to be invoked whenever an INSERT statement is attempted on the ORDERS table. The remainder of the definition (after the keyword AS) defines the action of the trigger. In this case, the action is a sequence of two UPDATE statements, one for the SALESREPS table and one for the PRODUCTS table. The row being inserted is referred to using the pseudo-table name inserted within the UPDATE statements. As the example shows, SQL Server extends the SQL language substantially to support triggers. Other extensions not shown here include IF/THEN/ELSE tests, looping, procedure calls, and even PRINT statements that display user messages.

The trigger capability, while popular in many DBMS products, is not a part of the ANSI/ISO SQL2 standard. As with other SQL features whose popularity has preceded standardization, this has let to a considerable divergence in trigger support across various DBMS brands. Some of the differences between brands are merely differences is syntax. Others reflect real differences in the underlying capability.

DB2's trigger support provides an instructive example of the differences. Here is the same trigger definition shown previously for SQL Server, this time using the DB2 syntax :

```
CREATE TRIGGER NEWORDER
              AFTER  INSERT ON ORDERS
    REFERENCING  NEW AS NEW_ORD
        FOR EACH  ROW MODE DB2SQL
            BEGIN  ATOMIC
          UPDATE  SALESREPS
                SET  SALES = SALES + NEW_ORD.AMOUNT
           WHERE SALESREPS. EMPL_NUM = NEW_ORD.REP;
          UPDATE PRODUCTS
                SET QTY_ON_HAND = QTY_ON_HAND − NEW_ORD.QTY
           WHERE PRODUCTS.MFR_ID = NEW_ORD.MFR
              AND PRODUCTS.PRODUCT_ID = NEW_ORD.PRODUCT;
              END
```

The beginning of the trigger definition includes the same elements as the SQL server definition, but rearranges them. It explicitly tells DB2 that the trigger is to be invoked after a new order is inserted into the database. DB2 also allows you to specify that the trigger is to be carried out before a triggering action is applied to the database contents. This doesn't make sense in this example, because the triggering event is an INSERT operation, but it does make sense for UPDATE or DELETE operations.

The DB2 REFERENCING clause specifies a table alias (NEW_ORD) that will be used to refer to the row being inserted throughout the remainder of the trigger definition. It serves the same function as the INSERTED keyword in the SQL Server trigger. The statement references the new values in the inserted row because this is an INSERT operation trigger. For a DELETE operation trigger, the old values would be reference For an UPDATE operation trigger, DB2 gives you the ability to refer to both the old (pre-UPDATE) values and new (post-UPDATE) values.

BEGIN ATOMIC and END serve as brackets around the sequence of SQL statements that define the triggered action. The two searched UPDATE statements in the body of the trigger definition are straightforward modifications of their SQL Server counterparts. They follow the standard SQL syntax for searched UPDATE statements, using the table alias specified by the REFERENCING clause to identify the particular row of the SALESREPS table and the PRODUCTS table to be updated. The row being inserted is referred to using the pseudo-table name inserted within the UPDATE statements.
Here is another example of a trigger definition, this time using Informix Universal Server:

```
CREATE TRIGGER NEWORDER
                INSERT ON ORDERS
                AFTER (EXECUTE PROCEDURE NEW_ORDER)
```

This trigger again specifies an action that is to take place after a new order is inserted. In this case, the multiple SQL statements that form the triggered action can't be specified directly in the trigger definition. Instead, the triggered statements are placed into an Informix stored procedure, named NEW_ORDER, and the trigger causes the stored procedure to be executed. As this and the preceding examples show, although the core concepts of a trigger mechanism are very consistent across databases, the specifics vary a great deal. Triggers are certainly among the least portable aspects of SQL databases today.

Triggers and Referential Integrity

Triggers provide an alternative way to implement the referential integrity constraints provided by foreign keys and primary keys. In fact, advocates of the trigger feature point out that the trigger mechanism is more flexible than the strict referential integrity provided by the ANSI/ISO standard. For example, here is a trigger that enforces referential integrity for the OFFICES/SALESREPS relationship and displays a message when an attempted update fails:

```
CREATE TRIGGER REP_UPDATE
        ON SALESREPS
FOR INSERT, UPDATE
AS IF (SELECT COUNT(*)
                FROM OFFICES, INSERTED
                WHERE OFFICES.OFFICE = INSERTED.REP_OFFICE) = 0)
            BEGIN
                PRINT “INVALID OFFICE NUMBER
            SPECIFIED.” ROLLBACK TRANSACTION
                END
```

Triggers can also be used to provide extended forms of referential integrity. For example, DB2 initially provided cascaded deletes through its CASCADE delete rule but did not

support cascaded updates if a primary key value is changed. This limitation need not apply to triggers, however. The following SQL Server trigger cascades any update of the OFFICE column in the OFFICES table down into the REP_OFFICE column of the SALESREPS table:

```
CREATE TRIGGER CHANGE_REP_OFFICE
        ON OFFICES
        FOR UPDATE
        AS IF UPDATE (OFFICE)
    BEGIN
                UPDATE SALESREPS
                        SET SALESREPS.REP_OFFICE =
            INSERTED.OFFICE FROM SALESREPS, INSERTED, DELETED
            WHERE SALESREPS.REP_OFFICE = DELETED.OFFICE
    END
```

As in the previous SQL Server example, the references DELETED. OFFICE and INSERTED. OFFICE in the trigger refer, respectively, to the values of the OFFICE column before and after the UPDATE statement. The trigger definition must be able to differentiate between these before and after values to perform the appropriate search and update actions specified by the trigger.

Trigger Advantages and Disadvantages

Over the last several years, the trigger mechanisms in many commercial DBMS products have expanded significantly. In many commercial implementations, the distinctions between triggers and stored procedures (described in Chapter 20) have blurred, so the action triggered by a single database change may be defined by hundreds of lines of stored procedure programming. The role of triggers has thus evolved beyond the enforcement of data integrity into a programming capability within the database.

A complete discussion of triggers is beyond the scope of this book, but even these simple examples show the power of the trigger mechanism. The major advantage of triggers is that business rules can be stored in the database and enforced consistently with each update to the database. This can dramatically reduce the complexity of application programs that access the database. Triggers also have some disadvantages, including these:

- **Database complexity.**
  When the rules are moved into the database, setting up the database becomes a more complex task. Users who could reasonably be expected to create small ad hoc applications with SQL will find that the programming logic of triggers makes the task much more difficult.
- **Hidden rules.**
  With the rules hidden away inside the database, programs that appear to perform straightforward database updates may, in fact, generate an enormous amount of database activity. The programmer no longer has total control over what happens to the database. Instead, a program-initiated database action may cause other, hidden actions.
- **Hidden performance implications.**
  With triggers stored inside the database, the consequences of executing a SQL statement are no longer completely visible to the programmer. In particular, an apparently simple SQL statement could, in concept, trigger a process that involves a

sequential scan of a very large database table, which would take a long time to complete. These performance implications of any given SQL statement are invisible to the programmer.

Source: https://www.youtube.com/watch?v=R3fvX_xf5P4

Questions:

1. What is Constraint
2. What are the different type of constraints. Also explain integrity constraints.
3. What is view. How DBMS handles view and what are its advantages.
4. Explain with example how to create different type of views.
5. How to do various updation on view.
6. What is the difference between table & view.
7. Explain DDL & DML.
8. What are various aggregation function.
9. Explain Triggers with the help of example
10. Explain Different types of Trigger.

Multiple Choice Questions

1 . Which of the following is not a class of constraint in SQL Server ?
a) NOT NULL
b) CHECK
c) **NULL**
d) UNIQUE

2. Constraints can be applied on :
a) Column
b) Table
c) Field
d**) All of the mentioned**

3. Which SQL function is used to count the number of rows in a SQL query
? a) COUNT()
b) NUMBER()
c) SUM()
d) COUNT(*)

4. Which SQL keyword is used to retrieve a maximum value ?
a) MOST
b) TOP
c) **MAX**
d) UPPER

5. Which of the following are TCL commands ?
a) UPDATE and TRUNCATE
b) SELECTand INSERT
c) GRANT and REVOKE
d) ROLLBACK and SAVEPOINT

6. With SQL, how can you return all the records from a table named “Persons” sorted descending by “FirstName” ?
a) SELECT * FROM Persons SORT BY ‘FirstName’ DESC b)
SELECT * FROM Persons ORDER FirstName DESC c)
SELECT * FROM Persons SORT ‘FirstName’ DESC
d) SELECT * FROM Persons ORDER BY FirstName DESC

7. You can delete a view with _____ command.
a**) DROP VIEW**
b) DELETE VIEW
c) REMOVE VIEW
d) TRUNCATE VIEW

8. 4. Which of the following is not a aggregate function?
a) Avg
b) Sum
c) With
d) Min

9. What is a view?
a) A view is a special stored procedure executed when certain event occurs
b) **A view is a virtual table which results of executing a pre-compiled query**
c) A view is a database diagram
d) None of the Mentioned

10. Which of the following blocks are used for error handling in SQL Server ? **a) TRY…CATCH**
b) TRY…FINAL
c) TRY…END
d) CATCH…TRY