Chapter 7: RELATIONAL DATABASE LANGUAGE

Background
The SQL language may be considered one of the major reasons for the success of relational databases in the commercial world. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, network or hierarchical systems—to relational systems. The reason is that even if users became dissatisfied with the particular relational DBMS product they chose to use, converting to another relational DBMS product would not be expected to be too expensive and time-consuming, since both systems would follow the same language standards.

The name SQL is derived from Structured Query Language. Originally, SQL was called SEQUEL (for Structured English QUERY Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. A joint effort by ANSI (the American National Standards Institute) and ISO (the International Standards Organization) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1. A revised and much expanded standard called SQL2 (also referred to as SQL-92) was subsequently developed. The next version of the standard was originally called SQL3, but is now called SQL-99.

SQL is a comprehensive database language: It has statements for data definition, query, and update. Hence, it is both a DOL and a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java.

SQL DATA DEFINITION AND DATA TYPES
The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

Schema and Catalog Concepts in SQL
An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements definitions. Alternatively, the schema can be assigned a name and authorization identifier.

For example, the following statement creates a schema called COMPANY owned by the user with authorization identifier JSMITH:

Chapter 7: Relational Database Language
CREATE SCHEMA COMPANY AUTHORIZATION JSMITH;

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

The CREATE TABLE Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

Implicitly, the CREATE TABLE statements are executed in the SQL schema in which the relations are declared. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

CREATE TABLE COMPANY.EMPLOYEE...

rather than

CREATE TABLE EMPLOYEE . . .

The relations declared through CREATE TABLE statements are called base tables (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from virtual relations, created through the CREATE VIEW statement, which may or may not correspond to an actual physical file.
Chapter 7: Relational Database Language

Attribute Data Types and Domains in SQL

The basic data types available for attributes include numeric, character string, bit string, boolean, date, and time.

- Numeric data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using `DECIMAL(i,j)` or `DEC(i,j)` or `NUMERIC(i,j)`—where `i`, the precision, is the total number of decimal digits and `j`, the scale, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

- Character-string data types are either fixed length--`CHAR(n)` or `CHARACTER(n)`, where `n` is the number of characters—or varying length—`VARCHAR(n)` or `CHAR VARYING(n)`, where `n` is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is case sensitive (a distinction is made between uppercase and lowercase).

- A boolean data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a boolean data type is UNKNOWN.

- New data types for date and time were added in SQL. The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.

- A timestamp data type (TIMESTAMP) includes both the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type.
• This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.
• The format of DATE, TIME, and TIMESTAMP can be considered as a special type of string. Hence, they can generally be used in string comparisons by being cast (or coerced or converted) into the equivalent strings.

It is possible to specify the data type of each attribute directly, alternatively, a domain can be declared, and the domain name used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE AS CHAR(9);

SPECIFYING BASIC CONSTRAINTS IN SQL
The basic constraints that can be specified in SQL as part of table creation which include key and referential integrity constraints, as well as restrictions on attribute domains and NULLs, and constraints on individual tuples within a relation.

Specifying Attribute Constraints and Attribute Defaults
Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

It is also possible to define a default value for an attribute by appending the clause DEFAULT <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute. If no default clause is specified, the default default value is NULL for attributes that do not have the NOTNULL constraint.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition. For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of `DNUMBER` in the `DEPARTMENT` table to the following:

`DNUMBER INT NOT NULL CHECK (DNUMBER > 0 AND DNUMBER < 21);`

Specifying Key and Referential Integrity Constraints
The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly. For example, the primary key of `DEPARTMENT` can be specified as follows

`DNUMBER INT PRIMARY KEY;`
The **UNIQUE** clause specifies alternate (secondary) keys, as illustrated in the **DEPARTMENT** table declarations in the above Figure. Referential integrity is specified via the **FOREIGN KEY** clause. A referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. However, the schema designer can specify an alternative action to be taken if a referential integrity constraint is violated, by attaching a referential triggered action clause to any foreign key constraint. The options include **SET NULL**, **CASCADE**, and **SET DEFAULT**. An option must be qualified with either **ON DELETE** or **ON UPDATE**.

The value of the affected referencing attributes is changed to **NULL** for **SET NULL**, and to the specified default value for **SET DEFAULT**. The action for **CASCADE ON DELETE** is to delete all the referencing tuples, whereas the action for **CASCADE ON UPDATE** is to change the value of the foreign key to the updated (new) primary key value for all referencing tuples.

**Giving Names to Constraints**

A constraint may be given a constraint name, following the keyword **CONSTRAINT**. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional.

**SCHEMA CHANGE STATEMENTS IN SQL**

**The DROP Command**

The **DROP** command can be used to drop **named** schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is not needed any more, the **DROP SCHEMA** command can be used. There are two **drop behavior** options: **CASCADE** and **RESTRICT**. For example, to remove the **COMPANY** database schema and all its tables, domains, and other elements, the **CASCADE** option is used as follows:

```
DROP SCHEMA COMPANY CASCADE;
```
If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed.

If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command. For example,

```
DROP TABLE DEPENDENT CASCADE;
```

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints (for example, by foreign key definitions in another relation). With the CASCADE option, all such constraints and views that reference the table are dropped automatically from the schema, along with the table itself.

The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relations in the COMPANY schema, we can use the command

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

BASIC QUERIES IN SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually.

The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
SELECT <attribute list>
FROM <table list>
WHERE <condition>;
```

where

- `<attribute list>` is a list of attribute names whose values are to be retrieved by the query.
- `<table list>` is a list of the relation names required to process the query.
- `<condition>` is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. Retrieve the birthdate and address of the employee(s) whose name is 'John B. Smith'.

```
SELECT BDATE, ADDRESS FROM EMPLOYEE WHERE FNAME='John' AND MINIT='B' AND LNAME='Smith';
```
Chapter 7: Relational Database Language

Ambiguous Attribute Names, Aliasing, and Tuple Variables

In SQL the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is the case, and a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period.

```sql
SELECT FNAME, EMPLOYEE.NAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DEPARTMENT.NAME='Research' AND
      DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result.

```sql
SELECT SSN
FROM EMPLOYEE;
```

If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected. For example,

```sql
SELECT SSN, DNAME
FROM EMPLOYEE, DEPARTMENT;
```

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes. For example,

```sql
SELECT *
FROM EMPLOYEE
WHERE DNO=5;
```

Tables as Sets in SQL

SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

If we do want to eliminate duplicate tuples from the result of an SQL query, we use the keyword DISTINCT in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates and a query with ALL does not eliminate duplicates.

```sql
SELECT ALL SALARY FROM EMPLOYEE;
```
SELECT DISTINCT SALARY FROM EMPLOYEE;

Substring Pattern Matching and Arithmetic Operators
This first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore _ replaces a single character. For example, consider the following query. Retrieve all employees whose address is in Houston, Texas.

SELECT FNAME, LNAME FROM EMPLOYEE WHERE ADDRESS LIKE '%Houston,TX%';
To retrieve all employees who were born during the 1950s, we can use.

SELECT FNAME, LNAME FROM EMPLOYEE WHERE BDATE LIKE '__ 5 '; Here, '5' must be the third character of the string (according to our format for date), so we use the value '__ 5 ', with each underscore serving as a placeholder for an arbitrary character. Also, we need a rule to specify apostrophes or single quotation marks ("') if they are to be included in a string, because they are used to begin and end strings. If an apostrophe (’) is needed, it is represented as two consecutive apostrophes ("’) so that it will not be interpreted as ending the string.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue query to see what their salaries would become.

SELECT FNAME, LNAME, 1.1*SALARY AS INCREASED_SAL FROM EMPLOYEE, WORKS_ON, PROJECT

For string data types, the concatenate operator II can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (-) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator that can be used for convenience is BETWEEN, which is illustrated in below query.

SELECT * FROM EMPLOYEE
WHERE (SALARY BETWEEN 30000 AND 40000) AND DNO = 5;
Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the \textit{ORDER BY} clause. This is illustrated by the following query.

\begin{verbatim}
SELECT DNAME, LNAME, FNAME, PNAME
FROM DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
WHERE DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER
ORDER BY DNAME, LNAME, FNAME;
\end{verbatim}

The default order is in ascending order of values. We can specify the keyword \texttt{DESC} if we want to see the result in a descending order of values. The keyword \texttt{ASC} can be used to specify ascending order explicitly.

MORE COMPLEX SQL QUERIES

Comparisons Involving \texttt{NULL} and Three-Valued Logic

In general, each \texttt{NULL} is considered to be different from every other \texttt{NULL} in the database. When a \texttt{NULL} is involved in a comparison operation, the result is considered to be \texttt{UNKNOWN} (it may be \texttt{TRUE} or it may be \texttt{FALSE}). Hence, SQL uses a three-valued logic with values \texttt{TRUE}, \texttt{FALSE}, and \texttt{UNKNOWN} instead of the standard two-valued logic with values \texttt{TRUE} or \texttt{FALSE}. It is therefore necessary to define the results of three-valued logical expressions when the logical connectives \texttt{AND}, \texttt{OR}, and \texttt{NOT}. In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression of the query to \texttt{TRUE} are selected. Tuple combinations that evaluate to \texttt{FALSE} or \texttt{UNKNOWN} are not selected.

\begin{verbatim}
\begin{tabular}{|c|c|c|c|}
\hline
\text{AND} & \text{TRUE} & \text{FALSE} & \text{UNKNOWN} \\
\hline
\text{TRUE} & \text{TRUE} & \text{FALSE} & \text{UNKNOWN} \\
\text{FALSE} & \text{FALSE} & \text{FALSE} & \text{FALSE} \\
\text{UNKNOWN} & \text{UNKNOWN} & \text{FALSE} & \text{UNKNOWN} \\
\hline
\text{OR} & \text{TRUE} & \text{FALSE} & \text{UNKNOWN} \\
\hline
\text{TRUE} & \text{TRUE} & \text{TRUE} & \text{TRUE} \\
\text{FALSE} & \text{TRUE} & \text{FALSE} & \text{UNKNOWN} \\
\text{UNKNOWN} & \text{TRUE} & \text{UNKNOWN} & \text{UNKNOWN} \\
\hline
\text{NOT} & & & \\
\hline
\text{TRUE} & \text{FALSE} & & \\
\text{FALSE} & \text{TRUE} & & \\
\text{UNKNOWN} & \text{UNKNOWN} & & \\
\hline
\end{tabular}
\end{verbatim}

SQL allows queries that check whether an attribute value is \texttt{NULL}. Rather than using \texttt{=} or \texttt{<>} to compare an attribute value to \texttt{NULL}, SQL uses \texttt{IS} or \texttt{IS NOT}. This is because SQL considers each \texttt{NULL} value as being distinct from every other \texttt{NULL} value, so equality comparison is not appropriate. For example, retrieve the names of all employees who do not have supervisors.
SELECT FNAME, LNAME
FROM EMPLOYEE
WHERE SUPERSSN IS NULL;

Nested Queries, Tuples, and Set/Multiset Comparisons
Nested queries are complete select-from-where blocks within the WHERE clause of another query. That other query is called the outer query.

For Example:

Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', as a manager of the department that controls the project.

SELECT DISTINCT PNUMBER
FROM PROJECT
WHERE PNUMBER IN ( SELECT PNUMBER
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUMBER=DNUMBER AND
MGRSSN=SSN AND
LNAME='Smith')

If a nested query returns a single attribute and a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a table (relation), which is a set or multiset of tuples.

SQL allows the use of tuples of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

SELECT DISTINCT ESSN
FROM WORKS_ON
WHERE (PNO, HOURS) IN (SELECT PNO, HOURS FROM WORKS_ON WHERE SSN='123456789');

This query will select the social security numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose SSN = '123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (PNO, HOURS) for each tuple in WORKS_ON with the set of union-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query). The = ANY (or SOME) operator returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN. The keywords ANY and SOME have the same meaning. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and < >. The keyword ALL can also be combined with each of these operators.

For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than all the values in the set (or multiset) V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:
SELECT LNAME, FNAME 
FROM EMPLOYEE 
WHERE SALARY > ALL (SELECT SALARY FROM EMPLOYEE WHERE DNO=5);

Correlated Nested Queries
Whenever a condition in the \textit{WHERE} clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated. We can understand a correlated query better by considering that the nested query is evaluated once for each tuple (or combination of tuples) in the outer query. For example, for each EMPLOYEE tuple, evaluate the nested query, which retrieves the ESSN values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the SSN value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

SELECT E.FNAME, E.LNAME 
FROM EMPLOYEE E, DEPENDENT D 
WHERE E.SSN = D.ESSN AND E.SEX = D.SEX AND E.FNAME = D.DEPENDENT_NAME;

The \textit{EXISTS} and \textit{UNIQUE} Functions in SQL
The \textit{EXISTS} function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. We illustrate the use of \textit{EXISTS} and \textit{NOT EXISTS} with some examples.

SELECT E.FNAME, E.LNAME 
FROM EMPLOYEE AS E 
WHERE EXISTS (SELECT * FROM DEPENDENT WHERE E.SSN = ESSN AND E.SEX = SEX AND E.FNAME = DEPENDENT_NAME);

Retrieve the names of employees who have no dependents.

SELECT FNAME, LNAME 
FROM EMPLOYEE 
WHERE NOT EXISTS (SELECT * FROM DEPENDENT WHERE SSN = ESSN);

\textbf{INSERT, DELETE, AND UPDATE STATEMENTS IN SQL}

\textbf{The \textit{INSERT} Command}
In its simplest form, \textit{INSERT} is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in \textit{the same order} in which the corresponding attributes were specified in the \textit{CREATE TABLE} command. For example, to add a new tuple to the \textit{EMPLOYEE} relation.

\texttt{INSERT INTO EMPLOYEE 
VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98Oak Forest,Katy,TX', 'M', 37000, '987654321', 4);}
A second form of the `INSERT` statement allows the user to specify explicit attribute names that correspond to the values provided in the `INSERT` command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. Attributes with `NULL` allowed or `DEFAULT` values are the ones that can be left out. For example, to enter a tuple for a new `EMPLOYEE` for whom we know only the `FNAME`, `LNAME`, `DNO`, and `SSN` attributes, we can use

```
INSERT INTO EMPLOYEE (FNAME, LNAME, DNO, SSN)
VALUES ('Richard', 'Marini', 4, '653298653');
```

Attributes not specified are set to their DEFAULT or to `NULL`, and the values are listed in the same order as the attributes are listed in the `INSERT` command itself. It is also possible to insert into a relation multiple tuples separated by commas in a single `INSERT` command. The attribute values forming each tuple are enclosed in parentheses.

```
INSERT INTO DEPTS_INFO (DEPT_NAME, MGR_SSN)
SELECT DNAME, MGR_SSN
FROM DEPARTMENT;
```

**The DELETE Command**

The `DELETE` command removes tuples from a relation. It includes a `WHERE` clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL statements. Depending on the number of tuples selected by the condition in the `WHERE` clause, zero, one, or several tuples can be deleted by a single `DELETE` command. A missing `WHERE` clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.

```
DELETE FROM EMPLOYEE WHERE LNAME='Brown';
```

```
DELETE FROM EMPLOYEE
```

**The UPDATE Command**

The `UPDATE` command is used to modify attribute values of one or more selected tuples. As in the `DELETE` command, a `WHERE` clause in the `UPDATE` command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL. An additional `SET` clause in the `UPDATE` command specifies the attributes to be modified and their new values. For example, to change the location and controlling department number of project number 10 to 'TEXAS' and 5.

```
UPDATE PROJECT
SET PLOCATION = 'TEXAS', DNUM = 5
WHERE PNUMBER=10;
```

Several tuples can be modified with a single `UPDATE` command.

```
UPDATE EMPLOYEE SET SALARY=SALARY+ 1.1*SALARY AS INCREASED_SAL WHERE DNO=5;
```
SPECIFYING GENERAL CONSTRAINTS AS ASSERTIONS

In SQL, users can specify general constraint via declarative assertions, using the `CREATE ASSERTION` statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the `WHERE` clause of an SQL query. For example, to specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for" in SQL, we can write the following assertion:

```sql
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT * FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
WHERE E.SALARY>M.SALARY AND E.DNO=D.DNUMBER AND D.MGRSSN=M.SSN));
```

VIEWS (VIRTUAL TABLES) IN SQL

A view in SQL terminology is a single table that is derived from other tables. A view in SQL terminology is a single table that is derived from other tables physical form; it is considered a virtual table, in contrast to base tables, whose tuples are actually stored in the database. We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

Specification of Views in SQL

In SQL, the command to specify a view is `CREATE VIEW`. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure:

V1:
```sql
CREATE VIEW WORKS_ON1
AS SELECT FNAME, LNAME, PNAME, HOURS
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN=ESSN AND PNO=PNUMBER;
```

V2:
```sql
CREATE VIEW DEPTINFO(DEPT_NAME,NO_OF_EMPS,TOTAL_SAL)
AS SELECT DNAME, COUNT(*), SUM(SALARY)
FROM DEPARTMENT, EMPLOYEE
WHERE DNUMBER=DNO
GROUP BY DNAME;
```

WORKS_ON1

| FNAME | LNAME | PNAME | HOURS |

DEPT_INFO

| DEPT_NAME | NO_OF_EMPS | TOTAL_SAL |
In VI, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON. View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on 'ProjectX', we can utilize the WORKS_ON1 view and specify the query below:

```
SELECT FNAME, LNAME FROM WORKS_ON1 WHERE PNAME='ProjectX';
```

If we do not need a view any more, we can use the DROP VIEW command to dispose of it. For example, to get rid of the view VI, we can use the SQL statement

```
DROP VIEW WORKS_ON1;
```

**View Update**

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown below:

```
UPDATE WORKS_ON1
SET PNAME = 'ProductY'
WHERE LNAME='Smith' AND FNAME='John' AND PNAME='ProductX';
```

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

**EMBEDDED SQL**

Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program (LIKE C, ADA, C++, PASCAL, BASIC, JAVA). An embedded SQL

Prof. Sushant S. Sundikar
Database Management System
statement is distinguished from programming language statements by prefixing it with the keywords
EXEC SQL so that a preprocessor (or precompiler) can separate embedded SQL statements from the host
language code. The SQL statements can be terminated by a semicolon (;) or a matching END-EXEC.

To illustrate the concepts of embedded SQL, we will use C as the host programming language. Within an
embedded SQL command, we may refer to specially declared C program variables. These are called
shared variables because they are used in both the C program and the embedded SQL statements. Shared
variables are prefixed by a colon (:) when they appear in an SQL statement. This distinguishes program
variable names from the names of database schema constructs such as attributes and relations. It also
allows program variables to have the same names as attribute names, since they are distinguishable by
the ":;" prefix in the SQL statement.

Names of database schema constructs—such as attributes and relations—can only be used within the SQL
commands, but shared program variables can be used elsewhere in the C program without the ":;" prefix.

Suppose that we want to write C programs to process the COMPANY database. We need to declare program
variables to match the types of the database attributes that the program will process. The programmer
can choose the names of the program variables; they may or may not have names that are identical to
their corresponding attributes. We will use the C program variables declared in Figure 1. Shared variables
are declared within a declare section in the program. A few of the common bindings of C types to SQL
types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types long,
short, float, and double, respectively. Fixed-length and varying-length strings (CHAR[i], VARCHAR[i]) in SQL can
be mapped to arrays of characters (char [i+1], varchar [i+1]) in C.

0) int loop;
1) EXEC SQL BEGIN DECLARE SECTION
2) varchar dname [16], fname [16], lname [16], address [31]
3) char ssn [10], bdate [11], sex [2], minit [2];
4) float salary, raise;
5) int dno, dnumber;
6) int SQLCODE; char SQLSTATE [6];
7) EXEC SQL END DECLARE SECTION;

Figure 1 C program variables used in the embedded SQL