Module 3 Notes

6.1 SQL Data Definition and Data Types

- **6.1.1 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.
- 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, and the elements can be defined later. For example, the following statement creates a schema called COMPANY owned by the user with authorization identifier 'Jsmith'.
  Note that each statement in SQL ends with a semicolon.
- **CREATE SCHEMA** COMPANY **AUTHORIZATION** 'Jsmith';
- 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

- **Attribute Data Types**

  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

- **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*) or CHARACTER VARYING(*n*), where *n* is the maximum number of characters.
- **Bit-string** data types are either of fixed length *n*—BIT(*n*)—or varying length—
  BIT VARYING(*n*), where *n* is the maximum number of bits.
- **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
- 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.

Specifying Constraints in SQL

What are Constraints?

Constraints enforce limits to the data or type of data that can be inserted/updated/deleted from a **table**.

Types of Constraints

- **NOT NULL -** Ensures that a column cannot have a NULL value
- **UNIQUE -** Ensures that all values in a column are different
- **PRIMARY KEY -** A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY -** Uniquely identifies a row/record in another table
- **CHECK -** Ensures that all values in a column satisfies a specific condition
- **DEFAULT -** Sets a default value for a column when no value is specified

Not Null Constraint

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column
- to NOT accept NULL values.

- CREATE TABLE employee (
- ID int NOT NULL,
- LastName varchar(255) NOT NULL,
- FirstName varchar(255) NOT NULL,
- Age int
-);

- To create a NOT NULL constraint on the "Age“ column when the “employee" table is already created, use the following SQL:

- ALTER TABLE Persons
- MODIFY Age int NOT NULL;

UNIQUE constraint

- The UNIQUE constraint ensures that all
- values in a column are different.

- CREATE TABLE employee (
- ID int NOT NULL UNIQUE,
- LastName varchar(255) NOT NULL,
- FirstName varchar(255),
- Age int
-);

PRIMARY KEY constraint

- The **PRIMARY KEY constraint** uniquely identifies each record in a table.

- **Primary keys** must contain UNIQUE values, and cannot contain NULL values.

- CREATE TABLE employee (
- ID int PRIMARY KEY,
- LastName varchar(255) NOT NULL,
- FirstName varchar(255),
- Age int
-);

Specifying PK using Constraint name

- CREATE TABLE employee (
- ID int NOT NULL,
- LastName varchar(255) NOT NULL,
- FirstName varchar(255),
- Age int,
- CONSTRAINT PK_Person PRIMARY KEY (ID)
-);

- Here PK_Person is a constraint name

FOREIGN KEY

- A FOREIGN KEY is a key used to link two tables together.

- A FOREIGN KEY is a field (or collection of
- fields) in one table that refers to the

- PRIMARY KEY in another table.
- The table containing the foreign key is called
- the child table,

Example for Foreign key

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

"Orders" table:

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

- Notice that the "PersonID" column in the "Orders" table points to the "PersonID“ column in the "Persons" table.

- The "PersonID" column in the "Persons" table
- is the PRIMARY KEY in the "Persons" table.
- The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

Foreign key example query

- CREATE TABLE Orders (
- OrderID int NOT NULL,
- OrderNumber int NOT NULL,
- PersonID int,
- PRIMARY KEY (OrderID),
- FOREIGN KEY (PersonID) REFERENCES Pers
- ons(PersonID)
-);

CHECK constraint

- The CHECK constraint is used to limit the value range that can be placed in a column. If you define a CHECK constraint on a single column it allows only certain values for this column.

- CREATE TABLE employee (
- ID int NOT NULL,
- LastName varchar(255) NOT NULL,
- FirstName varchar(255),
- Age int CHECK (Age>=18)
-);

DEFAULT constraint

- The DEFAULT constraint is used to provide a default value for a column. The default value will be added to all new
- records IF no other value is specified.

- CREATE TABLE employee (
- ID int NOT NULL,
- LastName varchar(255) NOT NULL,
- FirstName varchar(255),
- Age int,
- City varchar(255) DEFAULT ‘Bangalore'
-);

Basic Retrieval Queries in SQL

Q1: Retrieve the birth date and address of the employee(s) whose name is
'John B. Smith'.

Q2. Retrieve the name and address of all employees who work for the
'Research' department.

Q3. Retrieve all employees whose address is in Houston, Texas.

Q4. To retrieve all employees who were born during the 1970s.

Q5: Show the resulting salaries if every employee working on the 'ProductX' project is given a 10% raise.

Q6: Retrieve all employees in department 5 whose salary is between
$30,000 and $40,000

- **The INSERT Command**
  In its simplest **First form**, INSERT is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple.

- INSERT INTO EMPLOYEE
- VALUES (‘Richard’, ‘K’, ‘Marini’, ‘653298653’, ‘1962-12-30’, ‘98
- Oak Forest, Katy, TX’, ‘M’, 37000, ‘653298653’, 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.

- INSERT INTO EMPLOYEE
- VALUES (‘Richard’, ‘K’, ‘Marini’, ‘653298653’, ‘1962-12-30’, ‘98
- Oak Forest, Katy, TX’, ‘M’, 37000, ‘653298653’, 4);

- **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.

- DELETE FROM EMPLOYEE
- WHERE Lname = ‘Brown’;

- DELETE FROM EMPLOYEE
- WHERE Ssn = ‘123456789’;

- DELETE FROM EMPLOYEE
- WHERE Dno = 5;

- DELETE FROM EMPLOYEE; // This deletes all tuples from employee table

Options are available if a deletion operation causes a violation.

- **Restrict:** The first option, called **restrict, is to *reject the deletion*.**

- ***The second option, called cascade, is*** to *attempt to cascade (or propagate) the deletion by deleting tuples that reference the* tuple that is being deleted.

- **The UPDATE Command**
  The **UPDATE** command is used to modify attribute values of one or more selected tuples.

- UPDATE PROJECT
- SET Plocation = ‘Bellaire’, Dnum = 5
- WHERE Pnumber = 10;

- UPDATE EMPLOYEE
- SET Salary = Salary * 1.1
- WHERE Dno = 5;

Additional Features of SQL

- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules)

- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We called these commands a *storage definition language* (*SDL*)

- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.

- SQL has language constructs for specifying the *granting and revoking of privileges* to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement.

- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates.

- SQL and relational databases can interact with new technologies such as XML

More Complex SQL Retrieval Queries
Nested Queries

- **Nested queries**: It is the complete select-from-where blocks written within another SQL query. That other query is called the **outer query**.

- OR

- An SQL Query written inside another SQL Query is called as nested query.

Example 1

- **Select * from emp – outer Query/Main Query**
- **where sal > (select avg(sal) from emp);-- SubQuery/Inner Query**

- **The Subquery can be executed independently as it is not having any dependency. Therefore subquery is always executed first.**

- **The subquery is executed once.**

- **The result of the subquery will be used by the outer query to filter its results and give the required results to the user.**

Example 2

- **SELECT DISTINCT Essn**
- **FROM WORKS_ON**
- **WHERE (Pno, Hours) IN (SELECT Pno, Hours**
- **FROM WORKS_ON**
- **WHERE Essn = '123456789');**

- This query will select the Essns 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) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query

Correlated Nested Queries

- A Subquery which is related to the outer query.
- Whenever a condition in the 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**.
- Processing of your subquery depends on the values returned from the outer Query.
- 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.*

Example 1

- **Find the employees in each department who earn more than the average salary in that department.**
- Select * from emp e1
- Where sal > (select avg(sal)
- from emp e2
- where e2.dname=e1.dname);

- The inner query is executed once for each and every record from the outer query.

- Retrieve the name of each employee who has a dependent with the
- same first name and is the same Gender as the employee.
- Q16: SELECT E.Fname, E.Lname
- FROM EMPLOYEE AS E
- WHERE E.Ssn IN (SELECT D.Essn
- FROM DEPENDENT AS D
- WHERE E.Fname = D.Dependent_name
- AND E.gender = D.gender);

The EXISTS and UNIQUE Functions in SQL

- EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE;
- Hence, they can be used in a WHERE clause condition.
- The EXISTS function in SQL is used to check whether the result of a nested query is *empty* (contains no tuples) or not.
- The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples.

- Ex:
- SELECT E.Fname, E.Lname
- FROM EMPLOYEE AS E
- WHERE EXISTS (SELECT * FROM DEPENDENT AS D
- WHERE E.Ssn = D.Essn AND E.gender = D.gender
- AND E.Fname = D.Dependent_name);

- For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, gender, and Dependent_name as the EMPLOYEE tuple; if atleast one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and returns **FALSE** otherwise. On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and returns **FALSE** otherwise. Next, we illustrate the use of NOT EXISTS.

- Retrieve the names of employees who have no dependents.
- SELECT Fname, Lname
- FROM EMPLOYEE
- WHERE NOT EXISTS (SELECT *
- FROM DEPENDENT
- WHERE Ssn = Essn);
- The correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected because the **WHERE**-clause condition will evaluate to **TRUE** in this case.

Explicit Sets and Renaming in SQL

- It is also possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.
- **Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.**
- SELECT DISTINCT Essn
- FROM WORKS_ON
- WHERE Pno IN (1, 2, 3);

- In SQL, it is possible to **rename** any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names in general, and it can be used in appropriate parts of a query.

- SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name
- FROM EMPLOYEE AS E, EMPLOYEE AS S
- WHERE E.Super_ssn = S.Ssn;

Specifying Constraints as Assertions and Actions as Triggers

- **Assertions**

- In SQL, users can specify general constraints via declarative assertions, using the CREATE ASSERTION statement.
- 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:

- 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.Mgr_ssn = M.Ssn));

- The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to disable the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated.

Triggers

- In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
- For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation.
- A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database
- The CREATE TRIGGER statement is used to implement such actions in SQL.

- **CREATE TRIGGER** SALARY_VIOLATION
  BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
  ON EMPLOYEE
  FOR EACH ROW
  WHEN (**NEW**.SALARY > (**SELECT** SALARY **FROM** EMPLOYEE
  WHERE SSN = **NEW**.SUPERVISOR_SSN))
  INFORM_SUPERVISOR(**NEW**.Supervisor_ssn,
  NEW.Ssn);

- The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later
- A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:
  The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for.
- These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed.
- An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed

- The **condition** that determines whether the rule action should be executed:
- Once the triggering event has occurred, an *optional* condition may be evaluated.
- If *no condition* is specified, the action will be executed once the event occurs.
- If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger
- The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

- **Triggers can be used in various applications,** such as
- maintaining database consistency,
- monitoring database updates,
- and updating derived data automatically.

Views (Virtual Tables) in SQL

- A **view** in SQL terminology is a single table that is derived from other tables.6 These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically 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.

- For example, referring to the COMPANY database, we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE, WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables.

- **CREATE VIEW WORKS_ON1**
- **AS SELECT E.Fname, E.Lname, P.Pname,W.Hours**
- **FROM EMPLOYEE E, PROJECT P, WORKS_ON W**
- **WHERE Ssn = Essn AND Pno = Pnumber;**

Schema Change Statements in SQL

- **Schema Change Statements in SQL** are:
- **The DROP Command**
- **The ALTER Command**

- The DROP command can be used to drop *named* schema elements, such as
  - tables,
  - Constraints
  - schema (using the DROP SCHEMA command).
- 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. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself

- If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, we can get rid of the DEPENDENT relation by issuing the following command:

  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) or views or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

- Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the **DELETE command should be used instead of DROP TABLE.**

The ALTER Command

- The definition of a base table or of other named schema elements can be changed by using the ALTER command.
- For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema we can use the command

- **ALTER TABLE** COMPANY.EMPLOYEE **ADD COLUMN** Job VARCHAR(12);
- **ALTER TABLE** COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;
- **ALTER TABLE** COMPANY.EMPLOYEE
  DROP CONSTRAINT EMPSUPERFK **CASCADE**;