**ST.XAVIER’S COLLEGE**

MAITIGHAR, KATHMANDU



Database Management System

Assignment #8

Submitted By:

Aabhash Dhakal

013BSCCSIT001

2nd year/ 4th semester

Submitted to:

|  |  |
| --- | --- |
| Er. Sanjay Kumar Yadav  Lecturer  Department of Computer Science |  |

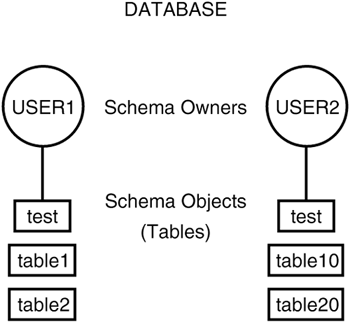
# Data Definition Language

A data definition language (DDL) is a computer language used to create and modify the structure of database objects in a database. These database objects include views, schemas, tables, indexes, etc.   
  
This term is also known as data description language in some contexts, as it describes the fields and records in a database table.

Commonly used DDL in SQL querying are:

* CREATE: This command builds a new table and has a predefined syntax. The CREATE statement syntax is CREATE TABLE [table name] ([column definitions]) [table parameters]. CREATE TABLE Employee (Employee Id INTEGER PRIMARY KEY, First name CHAR (50) NULL, Last name CHAR (75) NOT NULL).
* ALTER: An alter command modifies an existing database table. This command can add up additional column, drop existing columns and even change the data type of columns involved in a database table. An alter command syntax is ALTER object type object name parameters. ALTER TABLE Employee ADD DOB Date.
* DROP: A drop command deletes a table, index or view. Drop statement syntax is DROP object type object name. DROP TABLE Employee.
  + 1. **Domain Type, in SQL**
* *char(n)* : Fixed length character string, with user-specified length n.
* *varchar(n)* : Variable length character strings, with user-specified maximum length n.
* *int* : Integer (a finite subset of the integers that is machinedependent).
* *smallint* : Small integer (a machine-dependent subset of the integer domain type).
* *numeric(p,d)* : Fixed point number, with user-specified precision of p digits, with n digits to the right of decimal point.
* *real* :double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.
* *float(n)* : Floating point number, with user-specified precision of at least n digits.
  + 1. **Schema Definition in SQL**

Schema.create will connect to the database and create all tables, constraints, indexes, etc. While this is useful for development phases, when a system has been in production long enough it is often more convenient to generate the schema, and evolve it manually.



There are, two user accounts in the database that own tables: USER1 and USER2. Each user account has its own schema. Some examples for how the two users can access their own tables and tables owned by the other user follow:

| USER1 accesses own table1: | TABLE1 |
| --- | --- |
| USER1 accesses own test: | TEST |
| USER1 accesses USER2's table10: | USER2.TABLE10 |
| USER1 accesses USER2's test: | USER2.TEST |

In this example, both users have a table called TEST. Tables can have the same names in a database as long as they belong to different schemas. If you look at it this way, table names are always unique in a database because the schema owner is actually part of the table name. For instance, USER1.TEST is a different table than USER2.TEST. If you do not specify a schema with the table name when accessing tables in a database, the database server looks for a table that you own by default. That is, if USER1 tries to access TEST, the database server looks for a USER1-owned table named TEST before it looks for other objects owned by USER1, such as synonyms to tables in another schema.

* 1. **Data Manipulation Language**

The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from and modify tuples in the database.

* + 1. **The select Clause**

The SQL **SELECT** clause specifies the fields, constants, and expressions to display in the query results.

Syntax:

SELECT [ALL | DISTINCT] [TOP nExpr [PERCENT]] Select\_List\_Item [AS Column\_Name] [,...]

Example:

SELECT DISTINCT TAlias1.company, TAlias2.order\_date

* + 1. **The where Clause**

The **WHERE** clause specifies join and filter conditions that determine the rows that the query returns.

Syntax:

[WHERE JoinCondition | FilterCondition [AND | OR JoinCondition | FilterCondition] ...]

Example:

SELECT TAlias1.company, TAlias2.order\_date, TAlias2.shipped\_on;

FROM customer TAlias1, orders TAlias2;

WHERE TAlias1.cust\_id = TAlias2.cust\_id;

* + 1. **The from clause**

The **FROM** clause specifies one or more tables containing the data that the query retrieves from.

Syntax:

FROM [FORCE] Table\_List\_Item [, ...]

[[JoinType] JOIN DatabaseName!]Table [[AS] Local\_Alias]

[ON JoinCondition [AND | OR [JoinCondition | FilterCondition] ...]

Example:

SELECT customer.company;

FROM customer

* + 1. **The rename clause**

SQL provides a way of renaming the attributes of a result relation. It uses the **as** clause, taking the form:

*old-name* **as** *new-name*

The ‘**as’** clause is particularly useful when renaming relations. One reason to rename a relation is to replace a long relation name with a shortened version that is more convenient to use elsewhere in the query. To illustrate, we rewrite the query “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.”

**select** *T*.*name*, *S*.*course id*

**from** *instructor* **as** *T*, *teaches* **as** *S*

**where** *T*.*ID*= *S*.*ID*;

Another reason to rename a relation is a case where we wish to compare tuples in the same relation. We then need to take the Cartesian product of a relation with itself and, without renaming, it becomes impossible to distinguish one tuple from the other. Suppose that we want to write the query “Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.”We can write the SQL expression:

**Select distinct** *T*.*name*

**from** *instructor* **as** *T*, *instructor* **as** *S*

**where** *T.salary > S.salary* **and** *S.dept name* = ’Biology’;

* + 1. **Tuple Variable**

Tuple variables can be used in SQL, and are defined in the **from** clause:

**select distinct** *cname, T.loan#*

**from** *borrower* ***as*** *S, loan* ***as*** *T*

**where** *S.loan# = T.loan#*

These variables can then be used throughout the expression. Think of it as being something like the rename operator.

Finds the names of all branches that have assets greater than at least one branch located in Burnaby.

**select distinct** *T.bname*

**from** *branch S, branch T*

**where** *S.bcity=``Burnaby''* **and** *T.assets > S.assets*

* + 1. **String Operations**

SQL specifies strings by enclosing them in single quotes, for example, ’Computer’. A single quote character that is part of a string can be specified by using two single quote characters; for example, the string “It’s right” can be specified by “It”s right”.

The SQL standard specifies that the equality operation on strings is case sensitive; as a result the expression “’comp. sci.’ = ’Comp. Sci.’” evaluates to false.

However, some database systems, such as MySQL and SQL Server, do not distinguish uppercase from lowercase when matching strings; as a result “’comp. sci.’ = ’Comp. Sci.’” would evaluate to true on these databases. This default behavior can, however, be changed, either at the database level or at the level of specific attributes.

SQL also permits a variety of functions on character strings, such as concatenating (using “\_”), extracting substrings, finding the length of strings, converting strings to uppercase (using the function **upper**(*s*) where *s* is a string) and lowercase (using the function **lower**(*s*)), removing spaces at the end of the string (using **trim**(*s*)) and so on.

Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:

* Percent (%): The % character matches any substring.
* Underscore ( ): The character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

* ’Intro%’ matches any string beginning with “Intro”.
* ’%Comp%’ matches any string containing “Comp” as a substring, for example, ’Intro. to Computer Science’, and ’Computational Biology’.
* ’ ’ matches any string of exactly three characters.
* ’ %’ matches any string of at least three characters.
  + 1. **Ordering the display of Tuples**

SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

**select** *name*

**from** *instructor*

**where** *dept name* = ’Physics’

**order by** *name*;

By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order.

Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire *instructor* relation in descending order of *salary*. If severalinstructors have the same salary, we order them in ascending order by name.We express this query in SQL as follows:

**select** \*

**from** *instructor*

**order by** *salary* **desc**, *name* **asc**;

* + 1. **Duplicate Tuples**