SQL:Structured qery language

1)First download sql management studio to run sql queries in your laptop or device

2)Run management studio

3)Open management studio

4)Sql are 4 types generally to execute queries

a)DML: data manipulation language…………

1. SELECT - extracts data from a database
2. UPDATE - updates data in a database
3. DELETE - delectes data from a database
4. INSERT INTO - inserts new data into a database

b)DDL:data description language…………..

1. CREATE DATABASE - creates a new database
2. ALTER DATABASE - modifies a database
3. CREATE TABLE - creates a new data
4. ALTER TABLE - modifies a table
5. DROP TABLE - deletes a table
6. CREATE INDEX - create an index(search key)
7. DROP INDEX - delete an index

How to write SQL queries:

Table Name: Persons3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **E\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Street23 | Hyderabad |
| 2 | John | Tove | Street10 | Hyderabad |
| 3 | Pettersen | Kari | Street32 | Hyderabad |

Table Name: Orders1

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 1 |
| 4 | 24562 | 1 |
| 5 | 34764 | 15 |

SQL **CREATE DATABASE** Statement

The CREATE DATABASE statement is used to create a database.

**Syntax:**

CREATE DATABASE database\_name ;

Eg: create database nov12;

SQL **CREATE TABLE** Statement

The CREATE TABLE statement is used to create a table in a database

**Syntax:**

CREATE TABLE table\_name

(

column\_name1 data\_type,

column\_name2 data\_type,

column\_name3 data\_type,

....

)

Eg: Create table persons3(eid int,firstname varchar(20),lastname varchar(30), address varchar(40) ,city varchar(50));

Eg2: Create table orders1(oid int,orderno int,noofprdoucts int );

SQL **INSERT INTO** Statement

The INSERT INTO statement is used to insert a new row in a table

Syntax:

INSERT INTO table\_name VALUES ( value1, value2, value3,...)

Eg: insert into persons3 values (1,'ola','hansen','street1','hderabad');

insert into persons3 values (2,'tove','john','street2','hderabad');

insert into persons3 values (3,'kari','petterson','street3','hderabad');

Eg2: insert into orders1 values (1, 77895,3);

insert into orders1 values (2,44678,3);

insert into orders1 values (3,22456,1);

insert into orders1 values (4,24562,1);

insert into orders1 values (5,34764,15);

SQL **UPDATE** Statement

The UPDATE statement is used to update existing records in a table

Syntax:

UPDATE table\_name SET column1=value, column2=value2,...

WHERE some\_column=some\_value

Eg:

1)update persons3 set address='street' where lastname='kari';

SQL **DELETE** Statement

The DELETE statement is used to delete rows in a table

DELETE FROM table\_name WHERE some\_column=some\_value ;

Eg: DELETE FROM orders1 WHERE oid=5;

**Note: If you omit the WHERE clause, all records will be deleted**

SQL **ALTER TABLE** Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table

**Syntax: To Add a column**

ALTER TABLE table\_name

ADD column\_name datatype

**Syntax: To Delete a column**

ALTER TABLE table\_name

DROP COLUMN column\_name

**Syntax: To Change the Data Type**

ALTER TABLE table\_name

ALTER COLUMN column\_name datatype

**Example:**

ALTER TABLE Persons

ADD DateOfBirth date

Eg: alter table persons3 add salary int;

**Drop Column**

ALTER TABLE Persons

DROP COLUMN DateOfBirth

Eg:drop column salary;

**SQL SELECT Syntax**

SELECT column\_name(s)

FROM table\_name

or

SELECT \* FROM table\_name

Eg: select \* from persons1;

Eg2:select firstname,lastname,eid from persons3;

**SQL SELECT DISTINCT Statement**

In a table, some of the columns may contain duplicate values. This is not a problem, however, sometimes you will want to list only the different (distinct) values in a table

Syntax:

SELECT DISTINCT column\_name(s)

FROM table\_name

Eg:select distinct eid from persons3;

**SQL WHERE Clause**

The where clause is used to filter the records

Syntax:

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator value

**Exercise:** select only the persons living in the city "Hyderabad" from the table above

**Ans:** SELECT \* FROM Persons WHERE City='Hyderabad'

**Quotes around Text Fields**

SQL uses single quotes around text values (most database systems will also accept double quotes Although, numeric values should not be enclosed in quotes

**Example:**

SELECT \* FROM Persons WHERE FirstName='Tove'

|  |  |
| --- | --- |
| **Operators** | **Description** |
| = | Equal |
| < > | Not equal |
| > | Greater than |
| < | Less than |
| > = | Greater than or equal |
| < = | Less than or equal |
| BETWEEN | Between an inclusive range |
| LIKE | Search for a pattern |
| IN | If you know the exact value you want to return for at least one of the columns |

**SQL AND & OR Operators**

The AND & OR operators are used to filter records based on more than one condition **Example:**

select only the persons with the first name equal to "Tove" AND the last name equal to "John":

SELECT \* FROM Persons WHERE FirstName='Tove' AND LastName='John'

Example for Combination of AND and OR

SELECT \* FROM Persons WHERE LastName='John' AND (FirstName='Tove' OR FirstName='Ola')

**SQL ORDER BY Keyword**

The ORDER BY keyword is used to sort the result-set by a specified column

The ORDER BY keyword sort the records in ascending order by default

Systax:

SELECT column\_name(s) FROM table\_name ORDER BY column\_name(s) ASC|DESC

Eg:select eid from persons3 order by desc

select eid from persons3 order by asc

**Delete All Rows**

DELETE FROM table\_name

or

DELETE \* FROM table\_name

Eg:

Delete \* from persons3;

Delete eid from persons3;

**Note:** Be very careful when deleting records. You cannot undo this statement!

**SQL TOP Clause**

The TOP clause is used to specify the number of records to return

Syntax:

SELECT TOP number|percent column\_name(s)

FROM table\_name

Eg: select top 2 \* from persons3;

select top 2 eid from persons3;

**SQL Wildcards**

SQL wildcards can substitute for one or more characters when searching for data in a database.

SQL wildcards must be used with the SQL LIKE operator.

With SQL, the following wildcards can be used:

|  |  |
| --- | --- |
| **wildcard** | **Description** |
| % | A substitute for zero or more characters |
| - [charlist] | A substitute for exactly one character |
| ^ [charlist] or ! [charlist] | Any single character not in charlist |

SELECT \* FROM Persons WHERE FirstName LIKE '\_la'

SELECT \* FROM Persons WHERE LastName LIKE '[bsp]%'

**SQL LIKE Operator**

**The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.**

SELECT column\_name(s)

FROM table\_name

WHERE column\_name LIKE pattern

SELECT \* FROM Persons

WHERE City LIKE 's%'

eg: select \* from orders1 where orderno like 77895;

select \* from persons3 where firstname like '%p%';

'

SELECT \* FROM Persons

WHERE City NOT LIKE '%tav%'

Eg:

select \* from persons3 where firstname not like '%j%';

**SQL IN Operator**

The **IN** operator allows you to specify multiple values in a **WHERE** clause

Syntax:

SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1,value2,...)

**Example:** SELECT \* FROM Persons WHERE LastName IN ('Hansen','Pettersen')

Eg:

SELECT \* FROM Persons3 WHERE LastName IN ('Hansen','Pettersen');

**SQL BETWEEN Operator**

The **BETWEEN** operator selects a range of data between two values. The values can be numbers, text, or dates

Syntax:

SELECT column\_name(s)

FROM table\_name

WHERE column\_name

BETWEEN value1 AND value2

Eg1:SELECT \* FROM Persons3

WHERE LastName

BETWEEN 'Hansen' AND 'Pettersen'

Output:

1 ola hansen street1 hderabad

2 tove john street2 hderabad

Eg2:SELECT \* FROM Persons3

WHERE LastName

NOT BETWEEN 'Hansen' AND 'Pettersen'

Output: 3 kari petterson street3 hderabad

**SQL Alias**

You can give a table or a column another name by using an alias. This can be a good thing to do if you have very long or complex table names or column names

An alias name could be anything, but usually it is short. **Syntax For Tables:**SELECT column\_name(s) FROM table\_name AS alias\_name

**Syntax For Columns:**

SELECT column\_name AS alias\_name

FROM table\_name

Example

SELECT po.OrderID, p.LastName, p.FirstName

FROM Persons AS p,

Product\_Orders AS po

WHERE p.LastName='Hansen' AND p.FirstName='Ola'

**SQL UNION Operator**

The SQL UNION Operator

The **UNION** operator is used to combine the result-set of two or more **SELECT** statements.

Notice that each **SELECT** statement within the **UNION** must have the same number of columns. The columns must

also have similar data types. Also, the columns in each SELECT statement must be in the same order.

**Syntax:**

SELECT column\_name(s) FROM table\_name1

UNION

SELECT column\_name(s) FROM table\_name2

EG: SELECT eid FROM persons3

UNION

SELECT oid FROM orders1

**Note:**The **UNION** operator selects only distinct values by default. To allow duplicate values, use **UNION ALL.**

SELECT column\_name(s) FROM table\_name1

UNION ALL

SELECT column\_name(s) FROM table\_name2

**PS:** The column names in the result-set of a UNION are always equal to the column names in the first SELECT

statement in the **UNION.**

**"Employees\_Norway":**

|  |  |
| --- | --- |
| **E\_ID** | **E\_Name** |
| 01 | Hansen, Ola |
| 02 | John, Tove |
| 03 | John, Stephen |
| 04 | Pettersen, Kari |

**"Employees\_USA":**

Scott, Stephen

|  |  |
| --- | --- |
| **E\_ID** | **E\_Name** |
| 01 | Turner, Sally |
| 02 | Kent, Clark |
| 03 | John, Stephen |
| 04 |  |

Now we want to list all the different employees in Norway and USA

We use the following **SELECT** statement:

SELECT E\_Name FROM Employees\_Norway

UNION

SELECT E\_Name FROM Employees\_USA

The result-set will look like this:

|  |
| --- |
| **E\_Name** |
| Hansen, Ola |
| John, Tove |
| John, Stephen |
| Pettersen, Kari |
| Turner, Sally |
| Kent, Clark |
| Scott, Stephen |

**Note:** This command cannot be used to list all employees in Norway and USA. In the example above we have two employees with equal names, and only one of them will be listed. The UNION command selects only distinct values.

SQL UNION ALL Example

Now we want to list all employees in Norway and USA:

SELECT E\_Name FROM Employees\_Norway

UNION ALL

SELECT E\_Name FROM Employees\_USA

**Result :**

|  |
| --- |
| **E\_Name** |
| Hansen, Ola |
| John, Tove |
| John, Stephen |
| Pettersen, Kari |
| Turner, Sally |
| Kent, Clark |
| John, Stephen |
| Scott, Stephen |

**SQL SELECT INTO Statement**

The **SELECT INTO** statement selects data from one table and inserts it into a different table

The **SELECT INTO** statement is most often used to create backup copies of tables

Syntax:

SELECT \*

INTO new\_table\_name [IN externaldatabase]

FROM old\_tablename

Or we can select only the columns we want into the new table:

SELECT column\_name(s)

INTO new\_table\_name [IN externaldatabase]

FROM old\_tablename

Eg:

SELECT \*

INTO person4

FROM persons3;

select \* from person4;

**SQL SELECT INTO Example**

Make a Backup Copy - Now we want to make an exact copy of the data in our "Persons" table.

We use the following SQL statement:

SELECT \*

INTO Persons\_Backup

FROM Persons

Eg: SELECT \*

INTO Persons3\_Backup

FROM Persons3;

We can also use the IN clause to copy the table into another database:

SELECT \*

INTO Persons\_Backup IN 'Backup.mdb'

FROM Persons

We can also copy only a few fields into the new table:

SELECT LastName,FirstName

INTO Persons\_Backup

FROM Persons

**SQL SELECT INTO**- With a WHERE Clause

We can also add a WHERE clause.

The following SQL statement creates a "Persons\_Backup" table with only the persons who lives in the city

"Hyderabad":

SELECT LastName,Firstname

INTO Persons\_Backup

FROM Persons

WHERE City='Hyderabad'

**SQL Constraints**

Constraints are used to limit the type of data that can go into a table.

Constraints can be specified when a table is created (with the CREATE TABLE statement) or after the table is

created (with the ALTER TABLE statement).

We will focus on the following constraints:

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY
4. FOREIGN KEY
5. CHECK DEFAULT
6. DEFAULT

**SQL NOT NULL Constraint**

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

NULL values.

The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new

record, or update a record without adding a value to this field

**Example**

CREATE TABLE Persons

(

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Address varchar(255),

City varchar(255) )

**SQL UNIQUE Constraint**

The UNIQUE constraint uniquely identifies each record in a database table.

The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of

columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

CREATE TABLE Persons

(

P\_Id int NOT NULL UNIQUE,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Address varchar(255),

City varchar(255)

)

**Constraint on multiple columns**

CREATE TABLE Persons

(

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Address varchar(255),

City varchar(255),

CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

)

**SQL UNIQUE Constraint on ALTER TABLE**

ALTER TABLE Persons

ADD UNIQUE (P\_Id)

ALTER TABLE Persons

ADD CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

**To DROP a UNIQUE Constraint**

ALTER TABLE Persons

DROP CONSTRAINT uc\_PersonID

**SQL PRIMARY KEY Constraint**

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

Primary keys must contain unique values.

A primary key column cannot contain NULL values

Each table should have a primary key, and each table can have only ONE primary key

CREATE TABLE Persons

(

P\_Id int NOT NULL PRIMARY KEY,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Address varchar(255),

City varchar(255)

)

SQL PRIMARY KEY Constraint on ALTER TABLE

ALTER TABLE Persons

ADD PRIMARY KEY (P\_Id)

ALTER TABLE Persons

ADD CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName)

To DROP a PRIMARY KEY Constraint

ALTER TABLE Persons

DROP PRIMARY KEY

ALTER TABLE Persons

DROP CONSTRAINT pk\_PersonID

**SQL FOREIGN KEY Constraint**

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Let's illustrate the foreign key with an example. Look at the following two tables:

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Street25 | Hyderabad |
| 2 | John | Tove | Street10 | Hyderabad |
| 3 | Pettersen | Kari | Street32 | Bangalore |

**The "Orders" table**

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Note that the "P\_Id" column in the "Orders" table points to the "P\_Id" column in the "Persons" table.

The "P\_Id" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "P\_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents that invalid data form being inserted into the foreign key column,

because it has to be one of the values contained in the table it points to.

**Example**

CREATE TABLE Orders

(

O\_Id int NOT NULL PRIMARY KEY,

OrderNo int NOT NULL,

P\_Id int FOREIGN KEY REFERENCES Persons(P\_Id)

)

CREATE TABLE Orders

(

O\_Id int NOT NULL,

OrderNo int NOT NULL,

P\_Id int,

PRIMARY KEY (O\_Id),

CONSTRAINT fk\_PerOrders FOREIGN KEY (P\_Id)

REFERENCES Persons(P\_Id)

)

SQL FOREIGN KEY Constraint on ALTER TABLE

ALTER TABLE Orders

ADD FOREIGN KEY (P\_Id)

REFERENCES Persons(P\_Id)

ALTER TABLE Orders

ADD CONSTRAINT fk\_PerOrders

FOREIGN KEY (P\_Id)

REFERENCES Persons(P\_Id)

To DROP a FOREIGN KEY Constraint ALTER TABLE Orders DROP CONSTRAINT fk\_PerOrders

**SQL 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.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other

columns in the row

**SQL CHECK Constraint on CREATE TABLE**

The following SQL creates a CHECK constraint on the "P\_Id" column when the "Persons" table is created. The

CHECK constraint specifies that the column "P\_Id" must only include integers greater than 0.

CREATE TABLE Persons

(

P\_Id int NOT NULL CHECK (P\_Id>0),

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Address varchar(255),

City varchar(255)

)

SQL CHECK Constraint on ALTER TABLE ALTER TABLE Persons ADD CHECK (P\_Id>0)

**SQL DEFAULT Constraint**

The DEFAULT constraint is used to insert a default value into a column.

The default value will be added to all new records, if no other value is specified

CREATE TABLE Persons

(

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Address varchar(255),

City varchar(255) DEFAULT 'Hyderabad' )

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

CREATE TABLE Orders

(

O\_Id int NOT NULL,

OrderNo int NOT NULL,

P\_Id int,

OrderDate date DEFAULT GETDATE()

)

**SQL AGGREGATE FUNCTIONS**

SQL Aggregate functions return a single value, using values in a table column. Sales

Sales TABLE:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **OrderDate** | **OrderPrice** | **OrderQuantity** | **CustomerName** |
| 1 | 12/22/2005 | 160 | 2 | Smith |
| 2 | 08/10/2005 | 190 | 2 | Johnson |
| 3 | 07/13/2005 | 500 | 2 | Baldwin |
| 4 | 07/15/2005 | 420 | 2 | Smith |
| 5 | 12/22/2005 | 1000 | 4 | Wood |
| 6 | 10/02/2005 | 820 | 4 | Smith |

**The SQL COUNT Function:**

Returns the number of rows in a table satisfying the criteria specified in the WHERE clause

SELECT COUNT(\*) FROM SALES

WHERE CustomerName=’Smith’

Eg: SELECT COUNT(\*) FROM persons3 WHERE lastname='petterson';

Output is one

How can we get the number of unique customers that have ordered from our store? We need to use the DISTINCT

keyword along with the COUNT function to accomplish that:

SELECT COUNT(DISTINCT CustomerName) FROM Sales

Eg:select count (distinct firstname)from persons3;

Output:3

**The SQL SUM Function:**

Used to select the sum of values from numeric column.

SELECT SUM(OrderPrice) FROM Sales

Eg: select sum(eid)from persons3;

Output is 6

**The SQL AVG Function:**

Retrieves the average value for a numeric column.

SELECT AVG(OrderQuantity) FROM Sales

Eg:select avg(eid)from persons3;

Output is :2

You can use AVG function with the WHERE clause, thus restricting the data you operate on

SELECT AVG(OrderQuantity) FROM Sales WHERE OrderPrice > 200

**The SQL MIN Function:**

Selects the smallest number from a numeric column.

SELECT MIN(OrderPrice) FROM Sales

Eg: select min(firstname) from persons3;

Output is:kari

**The SQL MAX Function:**

Retrieves the maximum numeric value from a numeric column.

SELECT MAX(OrderPrice) FROM Sales

Eg: select max(firstname)from persons3;

Output is tove

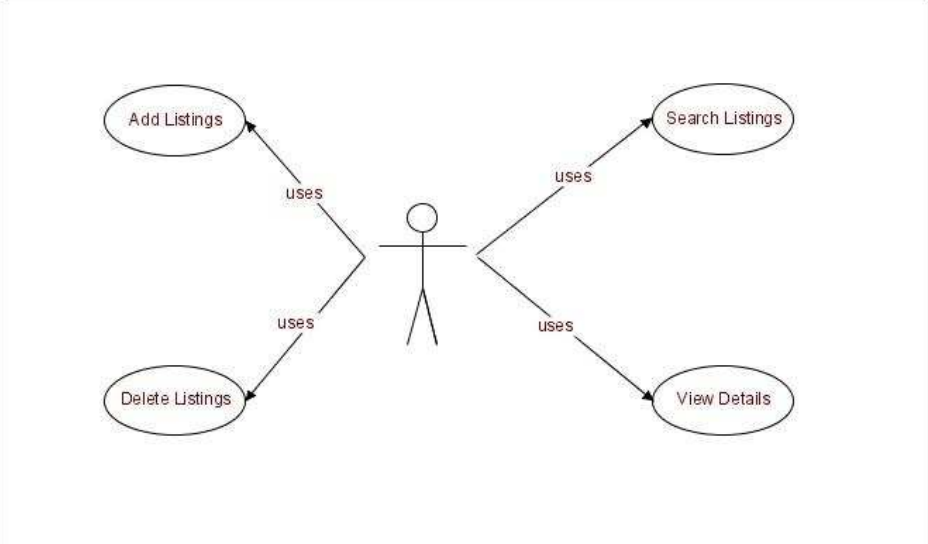
ASSignment :

designing the database schema for web application

<https://www.brickzrealtors.com/>

use case diagram:

Use Case Diagram for the Real Estate Web Application. The buyer can search the property listings, view property details, enter the selected listings in the buyer cart and can delete the listings from the buyer cart.



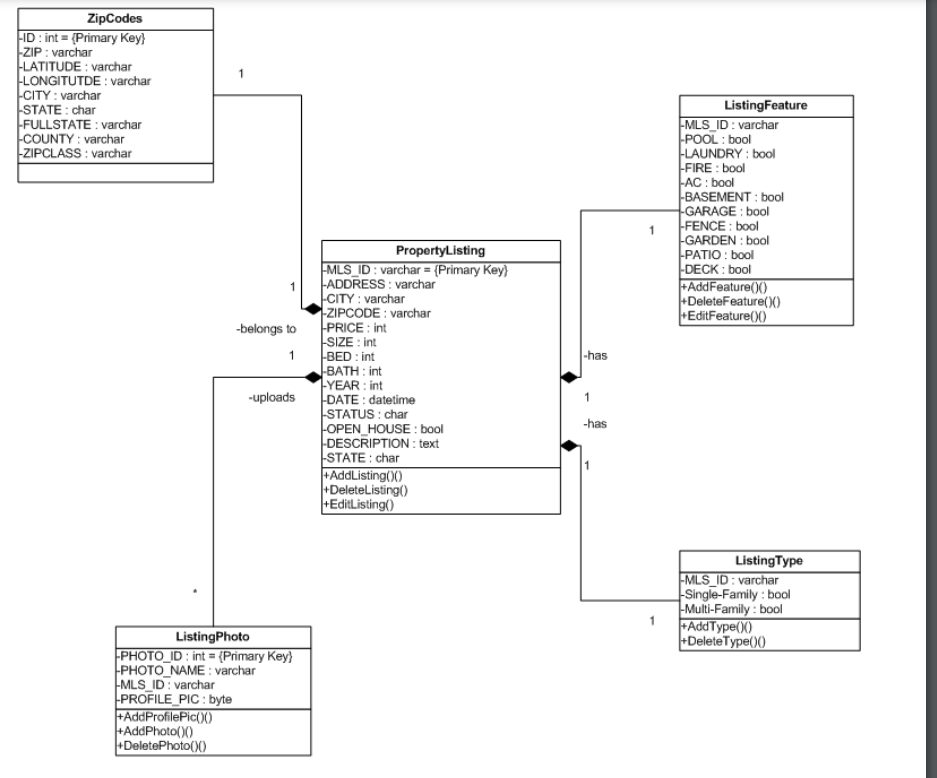
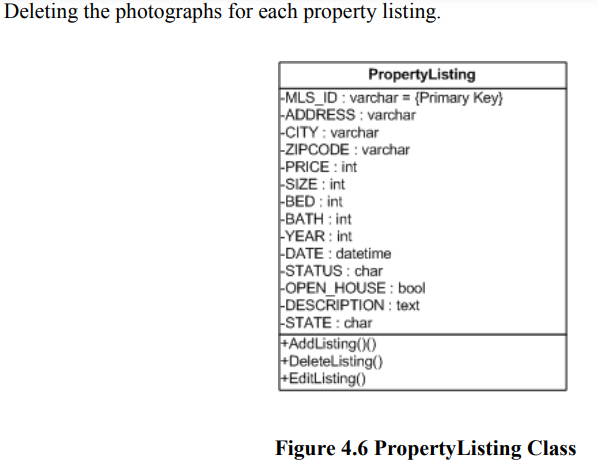


Fig: shows the Class Diagram representing the relations between the different classes.

Property Listing Class:

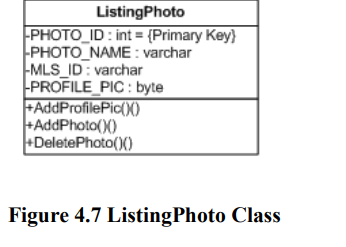
The PropertyListing class is the basic class for this application having attributes and description for each property listing which help the buyer to select the property desired for his need. This class includes the basic attributes to describe the listing like MLS#, listing\_name, listing\_address, listing\_description, etc. The methods included in this class are: Adding a new property listing to the database. Editing the existing property listing. Uploading photographs for each property listing. Deleting the photographs for each property listing.



ListingPhoto Class:

The ListingPhoto class associates itself with the PropertyListing. It is used to define the attributes of the photo that belongs to each Property Listing. The Real Estate Company can upload any number of photos but can have only one picture that acts as its profile picture. This 19 profile picture of the listing is viewed whenever that particular listing pops up on the data grid by entering a particular search scenario. The rest of the pictures corresponding to the listing can be viewed in the photo album. The methods corresponding to this class are AddPhoto and DeletePhoto.

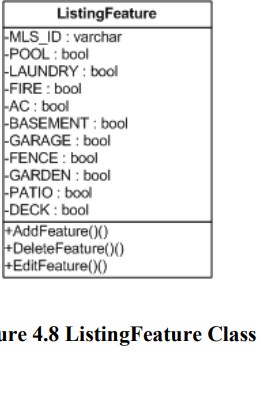
The Class Diagram for ListingPhoto is shown below:



ListingFeature Class:

The ListingFeature Class corresponds to the super class PropertyListing. This class contains the attributes which describe the extra features of the properties present in the PropertyListing Class. The methods AddFeatures and DeleteFeatures add information corresponding to the property present in the PropertyListing Class. Thus, all the attributes combined from the PropertyListing and PropertyPhoto Classes completely describe the property which a buyer is looking for.

The Class Diagram for ListingFeature is shown below:



PropertyType Class: PropertyType also corresponds to the super class PropertyListing. This class contains the attributes which describe the type of the property the buyer is looking for. The type of the property can be single-family or multi-family. The operations on the class named AddType and DeleteType determine the type of the property listing.

Class Diagram for PropertyType is shown below:

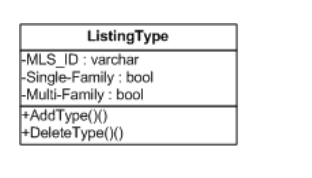


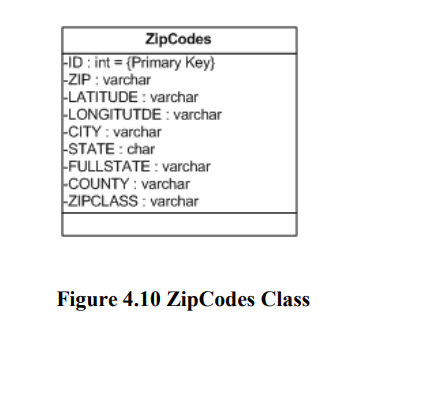
Figure 4.9 ListingType Class

ZipCodes Class:

The ZipCode class supports the PropertyListing class. One of the attribute of the PropertyListing class is zip code which identifies the place where the property is located. ZipCode class contains all the US zip codes and supports the PropertyListing class by locating all the zip codes in the vicinity of a particular zip code.

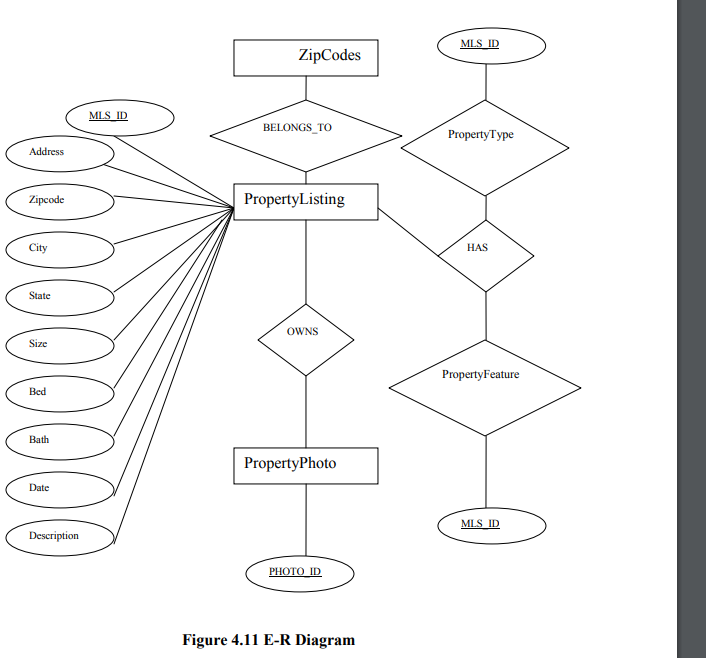
Class Diagram for ZipCode is shown below:

Figure 4.10 ZipCodes



Database Tier The Real Estate Web Application is supported by the SQL Server 2005 and its database. SQL Server provides a good response time of the data being stored making the search effective, convenient way for storing the photographs of the properties and storing the entire description and features of the Property Listings. The database schema for this application consists of five tables out of which the Listing is the main table to store the primary details of the property. Listing\_type and Listing\_feature reference the MLS# in the Listing table to describe more features of the listings. Lisitng\_photo stores the photographs of the properties in the form of binary data. The MLS\_ID column name is the attribute in the Listing table which is referenced by all the other tables. 4.2.3.1 ER Diagram

The Entity-Relationship model of the database being displayed is shown below:



4.2.3.2 Database Schema

Following is the database schema diagram for the Real Estate Web Application, the associations between various database classes can be easily seen. The associations are also consistent with the class diagram’s associations presented above. Member and Owns relationships are changed to strong entities in implemented.

