MANIPAL INSTITUTE OF TECHNOLOGY

Manipal – 576 104

DEPARTMENT OF INFORMATION & COMMUNICATION TECHNOLOGY

Certificate

This is to certify that Ms./Mr. ……...…… Reg. No. …..…… Section: …… Roll No: ……... has satisfactorily completed the lab exercises prescribed for Database Systems Lab [ICT 2266] of Second Year B. Tech. (CCE) Degree at MIT, Manipal, in the academic year 2022-2023.

Date: …….....

Signature of the faculty

CONTENTS

Lab No.	Title	Page No.	Marks	Remarks	Sign
	Course Objectives, Outcomes and Evaluation Plan	i			
	Instructions to the Students	ii			
	Introduction to vc# and SQL plus	iv			
1	Basics of vc# programs – i	1			
2	Basics of vc# programs –ii	14			
3	Data definition and manipulation language	23			
4	Basic operations of SQL queries	39			
5	Nested subqueries	63			
6	Procedural language	76			
7	E-R model and user interface design	91			
8	Data access from vc#	97			
9	Mongo DB	108			
10	Relational database design & Database implementation and data	116			
11	Project implementation	121			
12	Testing and validation	126			
	References	130			

Course Objectives

- To get acquainted with *Front end* design using Visual C# and *Back end* database processing using SQL and PL/SQL constructs.
- To familiarize with database design concepts like E-R Model, Schema design and Normalization.
- To design and implement a database mini-project.

Course Outcomes

The student should be able to:

- Implement a graphical user interface using a front end software.
- Create database design using conceptual data model.
- Demonstrate the working of procedural and non-procedural language.
- Construct the database using representational data model.

Evaluation plan

Split up of 60 marks for Regular Lab Evaluation
Total of 6 regular evaluations which will be carried out in alternate weeks. Each evaluation is for 10 marks which will have the following split up: Record : 4 Marks Viva: 4 Marks Execution: 2 Marks Total = 10 Marks Total Internal Marks: 6 * 10 =60 Marks
End Semester Lab evaluation: 40 marks (Duration 2 hrs)
SQL and PL/SQL Execution: 20 Marks Project Demo: 20 Marks Total: 20+20 =40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationary to every lab session
2. Be on time and follow the institution dress code
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
  - Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
  - Statements within the program should be properly indented.
  - Use meaningful names for variables and functions.
  - Make use of constants and type definitions wherever needed.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.

- The exercises for each week are divided under three sets:
  - Solved exercise
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Students missing out lab on genuine reasons like conference, sport or activities assigned by the department or institute will have to take **prior permission** from the HOD to attend **additional lab**(in other batch) and complete it **before** the student goes on leave. The student could be awarded marks for the write up for that day provided he submits it during the **immediate** next lab.
- Students who fall sick should get permission from the HOD for evaluating the lab records. However, the attendance will not be given for that lab.
- Students will be evaluated only by the faculty with whom they are registered even though they carry out additional experiment in other batch.
- Presence of the student during the lab end semester exams is mandatory even if the student assumes he has scored enough to pass the examination
- Minimum attendance of 75% is mandatory to write the final exam.
- If the student loses his book, he/she will have to rewrite all the lab details in the lab record.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

INTRODUCTION TO VC# AND SQL PLUS EDITOR

The **Visual Studio** product family shares a single integrated development environment (IDE) that is composed of several elements: the Menu bar, Standard toolbar, various tool windows docked or auto-hidden on the left, bottom, and right sides, as well as the editor space. The tool windows, menus, and toolbars available depend on the type of project or file you are working in. Depending on the settings you have applied, and any subsequent customizations you have made, the placement of tool windows and other elements in the IDE differs. You can change settings by using the Import and Export Settings Wizard. By selecting the Reset all settings option, you can change your default programming language.

You use the **Solution Explorer** to manage solution or project items and browse through your code. To display the Solution Explorer choose View Solution Explorer, press CTRL + ALT L, or type Solution Explorer in the Quick Launch window. The Solution Explorer helps you to do the following:

- Add projects to a solution
- Add items to a project
- Copy or move items and projects
- Rename solutions, projects, and items
- Delete, remove, or unload projects

Some Solution Explorer commands work differently in different project types. For example, the Delete command deletes a file permanently in a **Visual C#** project, but in a Visual C++ project it removes the link between the file and the project, without deleting the physical file.

The **Toolbox** displays icons for controls and other items that you can add to Visual Studio projects. To open the Toolbox, click Toolbox on the View menu. You can dock the Toolbox, and you can pin it open or set it to Auto Hide. Every Toolbox icon can be dragged to a design view or pasted in a code editor in the Visual Studio integrated development environment (IDE). Either action adds the fundamental code to create an instance of the Toolbox item in the active project file.

The Toolbox only displays items appropriate to the type of file you are working in. You can search within the Toolbox to further filter the items that appear. If your project requires a control that is not supported by the Client Profile, you can set your project to target the entire framework by editing the project properties.

Use the **Properties window** to view and change the design-time properties and events of selected objects that are located in editors and designers. You can also use the Properties window to edit and view file, project, and solution properties. You can find Properties Window on the View menu. You can also open it by pressing F4 or by typing Properties in the Quick Launch window.

The Properties window displays different types of editing fields, depending on the needs of a particular property. These edit fields include edit boxes, drop-down lists, and links to custom editor dialog boxes. Properties shown in grey are read-only.

SQL Plus is the primary interface to the Oracle Database server, provides a powerful yet easy-to-use environment for querying, defining, and controlling data. SQL Plus delivers a full implementation of Oracle SQL and PL/SQL, along with a rich set of extensions. The exceptional scalability of the Oracle Database, coupled with the object-relational technology of SQL Plus, allows you to develop your complex data types and objects using Oracle's integrated systems solution.

SQL Plus is a command prompt editor which is used to work on the database queries. To do so, you need to open the command prompt which looks like the usual command prompt. Give a command **connect** and press enter which then will ask for the username and password. Once you enter the username and password, you can work on the database queries.

SQL Plus understands five categories of text:

1. SQL statements
2. PL/SQL blocks
3. SQL*Plus internal commands, for example:
   - environment control commands such as SET
   - environment monitoring commands such as SHOW
4. Comments
5. External commands prefixed by the ‘!’ char

Scripts can include all these components.

LAB NO.: 1 **Date:**

BASIC VC# PROGRAMS - I

Objectives:

- To use different tools using visual studio and code them in C#.

Introduction:

Description of the tools used:

Text Box: A Text Box control is used to display, or accept as input, a single line of text. This control has additional functionality that is not found in the standard Windows text box control, including multiline editing and password character masking. A text box object is used to display text on a form or to get user input while a C# program is running. In a text box, a user can type data or paste it into the control from the clipboard. For displaying a text in a Text Box control, you can code like this

textBox1.Text = “Hello Manipal!”;

You can also collect the input value from a Text Box control to a variable like this way

string var;

var = textBox1.Text;

Label: Labels are one of the most frequently used C# control. We can use the Label control to display text in a set location on the page. Label controls can also be used to add descriptive text to a Form to provide the user with helpful information. The Label class is defined in the System. Windows.Forms namespace.

Add a Label control to the form - Click Label in the Toolbox and drag it over the forms Designer and drop it in the desired location. If you want to change the display text of the Label, you have to set a new text to the Text property of Label.

label1.Text = "This is my first Label";

In addition to displaying text, the Label control can also display an image using the Image property, or a combination of the Image Index and Image List properties.

label1.Image = Image.FromFile("C:\\testimage.jpg");

Button: A button is a control, which is an interactive component that enables users to communicate with an application. The Button class inherits directly from the Button Base class. A Button can be clicked by using the mouse, ENTER key, or SPACEBAR if

the button has focus. When you want to change display text of the Button, you can change the Text property of the button.

button1.Text = "Click Here";

Similarly if you want to load an Image to a Button control, you can code like this

button1.Image = Image.FromFile("C:\\testimage.jpg");

Radio Button: A radio button or option button enables the user to select a single option from a group of choices when paired with other Radio Button controls. When a user clicks on a radio button, it becomes checked, and all other radio buttons with same group become unchecked. The Radio Button control can display text, an image, or both. Use the Checked property to get or set the state of a Radio Button.

radioButton1.Checked = true;

The radio button and the check box are used for different functions. Use a radio button when you want the user to choose only one option. When you want the user to choose all appropriate options, use a check box. Like check boxes, radio buttons support a Checked property that indicates whether the radio button is selected.

Check Box: Check Boxes allow the user to make multiple selections from a number of options. Check Box is used, to give the user, an option, such as true/false or yes/no. You can click a check box to select it and click it again to deselect it. The Check Box control can display an image or text or both. Usually Check Box comes with a caption, which you can set in the Text property.

checkBox1.Text = "Net-informations.com";

You can use the Check Box control Three State property to direct the control to return the Checked, Unchecked, and Indeterminate values. You need to set the check box's Three State property to True to indicate that you want it to support three states.

checkBox1.ThreeState = true;

The radio button and the check box are used for different functions. Use a radio button when you want the user to choose only one option. When you want the user to choose all appropriate options, use a check box.

List Box: The List Box control enables you to display a list of items to the user that the user can select by clicking. In addition to display and selection functionality, the List Box also provides features that enable you to efficiently add items to the List Box and to find text within the items of the list. You can use the Add or Insert method to add items to a list box. The Add method adds new items at the end of an unsorted list box.

listBox1.Items.Add("Sunday");

If you want to retrieve a single selected item to a variable, you can code like this:

string var; var = listBox1.Text;

The Selection Mode property determines how many items in the list can be selected at a time. A List Box control can provide single or multiple selections using the Selection Mode property. If you change the selection mode property to multiple select, then you will retrieve a collection of items from ListBox1.SelectedItems property.

listBox1.SelectionMode = SelectionMode.MultiSimple;

Combo Box: A Combo Box displays a text box combined with a List Box, which enables the user to select items from the list or enter a new value. The user can type a value in the text field or click the button to display a drop down list. You can add individual objects with the Add method. You can delete items with the Remove method or clear the entire list with the Clear method.

To add the items into the drop down list:

```
comboBox1.Items.Add("Sunday");
comboBox1.Items.Add("Monday");
comboBox1.Items.Add("Tuesday");
```

Picture Box: The Windows Forms Picture Box control is used to display images in bitmap, GIF, icon, or JPEG formats**.** You can set the Image property to the Image you want to display, either at design time or at run time. You can programmatically change the image displayed in a picture box, which is particularly useful when you use a single form to display different pieces of information.

pictureBox1.Image = Image.FromFile("c:\\testImage.jpg");

The Size Mode property, which is set to values in the Picture Box Size Mode enumeration, controls the clipping and positioning of the image in the display area.

pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;

There are five different Picture Box Size Mode available under Picture Box control.

Auto Size - Sizes the picture box to the image.

Center Image - Centers the image in the picture box.

Normal - Places the upper-left corner of the image at upper left in the picture box.

Stretch Image - Allows you to stretch the image in code.

The Picture Box is not a selectable control, which means that it cannot receive input focus. The following C# program shows how to load a picture from a file and display it in stretch mode.

Progress Bar: A progress bar is a control that an application can use to indicate the progress of a lengthy operation such as calculating a complex result, downloading a large file from the web etc. Progress Bar controls are used whenever an operation takes more than a short period of time. The Maximum and Minimum properties define the range of values to represent the progress of a task.

Minimum : Sets the lower value for the range of valid values for progress.
Maximum : Sets the upper value for the range of valid values for progress.
Value : This property obtains or sets the current level of progress.

By default, Minimum and Maximum are set to 0 and 100. As the task proceeds, the Progress Bar fills in from the left to the right to delay the program briefly so that you can view changes in the progress bar clearly.

C# DateTimePicker Control: The Date Time Picker control allows you to display and collect date and time from the user with a specified format. The Date Time Picker control has two parts, a label that displays the selected date and a popup calendar that allows users to select a new date. The most important property of the Date Time Picker is the Value property, which holds the selected date and time.

```
dateTimePicker1.Value = DateTime.Today;
```

The Value property contains the current date and time the control is set to. You can use the Text property or the appropriate member of Value to get the date and time value.

```
DateTime iDate;
iDate = dateTimePicker1.Value;
```

The control can display one of several styles, depending on its property values. The values can be displayed in four formats, which are set by the Format property: Long, Short, Time, or Custom.

```
dateTimePicker1.Format = DateTimePickerFormat.Short;
```

Message Box: Displays a message window, also known as a dialog box, which presents a message to the user. It is a modal window, blocking other actions in the application until the user closes it. A Message Box can contain text, buttons, and symbols that inform and instruct the user.

Tree view: The Tree View control contains a hierarchy of Tree View Item controls. It provides a way to display information in a hierarchical structure by using collapsible nodes. The top level in a tree view are root nodes that can be expanded or collapsed if the nodes have child nodes. You can explicitly define the Tree View content or a data source can provide the content. The user can expand the Tree Node by clicking the plus sign (+) button, if one is displayed next to the Tree Node, or you can expand the Tree Node by calling the TreeNode. Expand method. You can also navigate through tree views with various properties: First Node, Last Node, Next Node, Prev Node, Next Visible Node, Prev Visible Node.

The full path method of tree view control provides the path from root node to the selected node.

treeView1.SelectedNode.FullPath.ToString();

Tree nodes can optionally display check boxes. To display the check boxes, set the Check Boxes property of the Tree View to true.

treeView1.CheckBoxes = true;

Example: Design a simple calculator using C#.

Open Visual studio. Click on **File->New->Project**. Select **windows form Application.** Change the **name of the project** in the **Name field** below and choose the **path** where the project has to be stored. Then Click on **OK** which will redirect to the form with a name **Form1** which acts as a panel where the user interface will be created. To do so, Tool Box is required which is displayed to the left side. (If it is not visible then go to **View->ToolBox.**)

From the **ToolBox** select **Button** control and drop it in the Form. Then in the **Properties window** (which is on the bottom right) go to **Text field** and change value-**textBox1** currently - to **1**. Text field in the properties window depicts the display name for the tool control.

Change the **Name** field of the button in the properties window to **cmd1**. This name is the name taken into the code. Similarly drag and drop the buttons for the rest of the numbers and the operators. Rename the text field and the Name field of the respective buttons according to the requirements. Now, drag and drop the **text field** from the tool box. Now we need to code the buttons so that we can make it usable. For that we will require few variable which are to be declared as given below:

public partial class Form1 : Form

```
{
string input = string.Empty; //to read the input when clicked
string Op1 = string.Empty; //First operand
string Op2 = string.Empty; //Second operand
char Operator; //Operator
double res = 0.0; //Final result
public Form1()
{
InitializeComponent();
}
```

Double click on button1 which will redirect you to **Form1.cs** from **Form1.cs[Design].** As you can see it has created a function with name **cmd1_Click** (cmd1 is the name which you had given in the name field of the properties window). The default event ofthe button is Click, that is why the word Click is attached withcmd1, which means **"On the event Click on the button"** the executable statements under cmd1_Click function should be executed. Type the following code under the created function

```
private void cmd1_Click(object sender, Event Args e)
{
this.textBox1.Text = string.Empty;
input = input + "1";
this.textBox1.Text += input;
}
```

Similarly, code for button 2 to button 9 by changing the values of input variable asshown in the code below:

```
private void cmd2_Click(object sender, EventArgs e)
{
this.textBox1.Text = string.Empty;
input += "2";
this.textBox1.Text += input;
}
```

Now it is time to code the operators. Similar to the previous step, double click on the operator button, maybe'+', which will be redirected to the function in Form1.cs. Type the following code under it.

```
private void add_Click(object sender, EventArgs e)
{
Op1 = input;
Operator = '+';
input = string.Empty;
}
```

When the operator button is clicked, it means that until then whatever the numbers are being pressed should be considered as the first operand. That's why the statement,**Op1=input.** And then input variable is cleared so that it starts reading the second operand. Similarly code for the rest of the operators.

Once the numbers and operators are coded, we need result. Double click on the equal to button and type the following code.

```
private void Ans_Click_Click(object sender, EventArgs e)
{
Op2 = input;
double num1, num2;
double.TryParse(Op1, out num1);
double.TryParse(Op2, out num2);
if (Operator == '+')
{
res = num1 + num2;
this.textBox1.Text = res.ToString();
}
else if (Operator == '-')
{
res = num1 - num2;
textBox1.Text = res.ToString();
}
else if (Operator == '*')
{
res = num1 * num2;
textBox1.Text = res.ToString();
}
```

```
else if (Operator == ’/’)
{
if (num2 != 0)
{
res = num1 / num2;
textBox1.Text = res.ToString();
}
else
{
textBox1.Text = "DIV/Zero!";
}
}
 input = string.Empty;
}
```

Lab exercises:

1. Design a scientific calculator using C#. Have atleast 4 different kinds of scientific functions.
2. Develop a simple form to enter necessary details for online registration of students. Also, display the message (confirm or not) along with details entered on submit. Perform necessary validation. Use Text Box, Radio button, Combo box, Check box, Calendar, Label, Button, and Message Box.

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

LAB NO.: 2 **Date:**

BASIC VC# PROGRAMS - II

Objectives:

- To learn the usage of MenuBar, rich text box and few other components of C#.

Introduction:

To create a notepad application, the primary requirements are to place a **MenuBar** control and a textbox control. A menubar is a collection of menus and a menu is a collection of menu items.

To create a simple notepad application, create a new project with a name NotepadApp. According to your requirement, you can add the submenus in a menu by typing the menu name and the sub menu name in the box which appears right below the main menu tab name you have given.

Under File menu: New, Open, Save, Print, Exit.
Under Edit menu: Cut, Copy, Paste, SelectAll.
Under Format menu: Font, Color.
Under Help menu: AboutUs.

Set the textbox property as: Name = txtContent, Dock = Fill and Multiline = true.

To work with the above application, 6 controls are required that is

- OpenFileDialog control : This control is used here to print an open dialog box.
- SaveFileDialog control : This control is used here to print a file dialog box.
- PrintDialog control : This control is used here to print a print dialog box.
- FontDialog control : This control is used here to print a font dialog box.
- ColorDialog control : This control is used here to print a color dialog box.
- MenuStrip control : This is used here to add different menu items.

So to add the above controls, Go to the tool box and add accordingly the controls as mentioned.

Rich Textbox: Rich Textbox is an advanced textbox control with very interactive features and large number of properties. It supports the RTF (Rich Text Format) format. Rich textbox have some advance properties and it is in multiline mode by default. To use the Rich Textbox in the project the assembly required to refer is Presentation Framework (in PresentationFramework.dll) and the namespace is System.Windows.Control. Below is some of its properties that can be used in the project:

- **Context menu strip:** By this property we can get or set the context menu element that should appear whenever the context menu is requested by the user. Context Menu strip is explained later.
- **Cursor:** Get and set the cursor that displays when the mouse pointer is over this element. In this project we set its value as IBeam. Other than IBeam there are many options like arrow, cross, default etc.
- **Dock:** This property automatically resizes the control when the size of the parent container is changed.
- **Tooltip:** Get or set the tool tip object that is displayed for this element in the user interface (UI).

Menu Strip: The next control is menu strip that should be used in the project. The menu strip adds the menu bar in Windows form and then add default menus and create custom menus directly in visual studio. The assembly required for this control is System.Windows.Form. We can add menu strip directly by drag and drop or at run time.

Adding the items under the submenu can also be done by selecting the menu strip. In the top right corner a black triangle symbol can be seen. **Click on the triangle**. Select **Edit items** option. In the dialog box which is open now, click on Add button which will add a new **Tool Strip Menu Item.** On the right of the pane, you can see the **property window** of the new tool strip which you have just created. Change the **Text** of the strip and the **Name** field of the strip. The name field depicts the displayed name in the menu.

To make the notepad work, you need to code the respective functionalities. To open a file, drag and drop the open file dialog component from the tool set which creates the

object for the same. It can be seen in the pane Form1. Now to make the open control work, write the following code under the open sub menu option. Before adding the below code include the namespace **using System.IO**;

```
private void openToolStripMenuItem_Click(object sender, EventArgs e)

    {
       OpenFileDialog dlg = new OpenFileDialog();
       dlg.Title = "Open";
      dlg.ShowDialog();

       string fName = dlg.FileName;
       StreamReader sr = new StreamReader(fName);
       richTextBox1.Text = sr.ReadToEnd();
       sr.Close();
}
```

To save a file:

```
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
      {
      saveFileDialog1.ShowDialog();
      string fName = saveFileDialog1.FileName;
      StreamWriter sw = new StreamWriter(fName);
      sw.Write(richTextBox1.Text);
      sw.Flush();
      sw.Close();
      }
private void fontToolStripMenuItem_Click(object sender, EventArgs e)
    {
       FontDialog fd = new FontDialog();
       fd.Font = richTextBox1.SelectionFont;
```

```
        fd.Color = richTextBox1.SelectionColor;
        if (fd.ShowDialog() == DialogResult.OK)
        {
            richTextBox1.SelectionFont = fd.Font;
            richTextBox1.SelectionColor = fd.Color;
        }
    }
```

Displaying One Form from Another:

To add a new form to the project, from the menu select **Project -> Add New Item -> Windows Forms** (under visual c#) **- >Windows Form**, click on Add after giving a suitable name.

Following example displays a form namely Form2 from Form1. This example requires two forms named Form1 and Form2. Form1 contains a Button control named button1. Set button1's Click event handler to button1_Click.

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        Form2 frm = new Form2();
        frm.Show();
    }
```

Lab exercises:

1. Develop a notepad application using Rich Text Box, Menu Strip, File Dialogue, Color Dialog, Font Dialog components.

2. Develop a user interface for a banking application. A customer should be able to login with his/her credentials. Also, customer should be able to change his/her password. The second form should display the customer's user name, balance, last access, date and last 5 transactions. The third form should facilitate money transfer by adding beneficiary. The amount transferred and the current balance in the account should be displayed as a message.

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

LAB NO.: 3 **Date:**

DATA DEFINITION AND MANIPULATION LANGUAGE

Objectives:

- To learn Data Definition Language and Data Manipulation Language

Introduction:

IBM developed the original version of SQL, called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language). Many products now support the SQL language. SQL has clearly established itself as the standard relational database language. In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86. ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, followed by SQL:1999, SQL:2003, SQL:2006, and most recently SQL:2008.

The SQL language has several parts:

- **Data-definition language** (DDL). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language** (DML). 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.
- **Integrity**. The SQL DDL includes commands for specifying integrity constraints that must be satisfied by the data stored in the database. Updates that violate integrity constraints are disallowed.
- **View definition**. The SQL DDL includes commands for defining views.
- **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

SQL Data Definition: The set of relations in a database must be specified to the system by means of a data- definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Basic Types

The SQL standard supports a variety of built-in types, including:

- **char**(*n*): A fixed-length character string with user-specified length *n*. The full form, **character**, can be used instead.
- **varchar**(*n*): A variable-length character string with user-specified maximum length *n*. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric**(*p, d*): A fixed-point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float**(*n*): A floating-point number, with precision of at least *n* digits.

Each type may include a special value called the **null** value. A null value indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered.

Basic Schema Definition

We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database.

create table *department (*
deptname **varchar** (20),
building **varchar** (15),
budget **numeric** (12,2),
primary key (*deptname*));

The relation created above has three attributes, *deptname*, which is a character string of maximum length 20, *building*, which is a character string of maximum length 15, and *budget*, which is a number with 12 digits in total, 2 of which are after the decimal point. The **create table** command also specifies that the *deptname* attribute is the primary key of the *department* relation. The general form of the **create table** command is:

create table *r*
($A_1 D_1$,
$A_2 D_2$,
. . . ,
A_n D_n,
_integrity-constraint$_1$
_,
. . . ,
_integrity-constraint$_k$
_);

where r is the name of the relation, each A_i is the name of an attribute in the schema of relation r, and D_i is the domain of attribute A_i; that is, D_i specifies the type of attribute A_i along with optional constraints that restrict the set of allowed values for A_i. The semicolon shown at the end of the **create table** statements, as well as at the end of other SQL statements later in this chapter, is optional in many SQL implementations. SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

- **primary key** (A_{j1}, A_{j2}, . . . , A_{jm}): The **primary-key** specification says that attributes A_{j1}, A_{j2}, . . . , A_{jm} form the primary key for the relation. The primary key attributes are required to be *non-null* and *unique*; that is, no tuple can have a null value for a primary-key attribute, and no two tuples in the relation can be equal on all the primary-key attributes.

- **foreign key** (A_{k1}, A_{k2}, . . . , A_{kn}) **references** *s*: The **foreign key** specification says that the values of attributes (A_{k1}, A_{k2}, . . . , A_{kn}) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation *s*.
- **not null**: The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute. The **not null** constraint on the *name* attribute of the *instructor* relation ensures that the name of an instructor cannot be null. Kindly refer page no. 39 for instructor schema.
- **Unique Constraint: The unique** (*Aj*1 , *Aj*2, . . . , *Ajm*) says that attributes *Aj*1 , *Aj*2, . . . , *Ajm* form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes. However, candidate key attributes are permitted to be *null* unless they have explicitly been declared to be **not null**.
- **The check Clause: T**he clause **check**(*P*) specifies a predicate *P* that must be satisfied by every tuple in a relation. A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause **check** (*budget* > 0) in the **create table** command for relation *department* would ensure that the value of **budget** is nonnegative. For example,

  create table *department (*
  deptname **varchar** (20),
  building **varchar** (15),
  budget **numeric** (12,2),
  primary key (*deptname*),
  check (*budget* > 0));

SQL prevents any update to the database that violates an integrity constraint. For example, if a newly inserted or modified tuple in a relation has null values for any primary-key attribute, or if the tuple has the same value on the primary-key attributes as does another tuple in the relation, SQL flags an error and prevents the update. Similarly, an insertion of a *course* tuple with a *deptname* value that does not appear in the *department* relation would violate the foreign-key constraint on *course*, and SQL prevents such an insertion from taking place. A newly created relation is empty initially.

Insertion:

We can use the **insert** command to load data into the relation. For example, if we wish to insert the fact that there is an instructor named Smith in the Biology department with *instructor_id* 10211and a salary of $66,000, we write:

insert into *instructor* **values** (10211, 'Smith', 'Biology', 66000);

The values are specified in the *order* in which the corresponding attributes are listed in the relation schema.

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes. The simplest **insert** statement is a request to insert one tuple. Consider *course* relation schema as *course* (*course_id, title, deptname, credits*) and suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title "Database Systems", and 4 credit hours. We write:

insert into *course*
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

In this example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. For the benefit of users who may not remember the order of the attributes, SQL allows the attributes to be specified as part of the **insert** statement. For example, the following SQL **insert** statements are identical in function to the preceding one:

insert into *course* (*course id, title, deptname, credits*)
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to make each student (kindly refer page no. 55 for the student schema) in the Music department who has earned more than 144 credit hours, an instructor in the Music department, with a salary of
$18,000. We write:

insert into *instructor*
select *ID, name, deptname,* 18000
from *student*
where *deptname* = 'Music' **and** *tot cred* >144;

SQL evaluates the **select** statement first, giving a set of tuples that is then inserted into the *instructor* relation. Each tuple has an *ID*, a *name*, a *deptname* (Music), and a salary of $18,000.

Deletion

A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

delete from *r*

The **delete** statement first finds all tuples *t* in *r* for which *P*(*t*) is true, and then deletes them from *r*. The **where** clause can be omitted, in which case all tuples in *r* are deleted. Note that a **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation. The predicate in the **where** clause may be as complex as a **select** command's **where** clause. Here are examples of SQL delete requests:

- Delete all tuples in the *instructor* relation pertaining to instructors in the Finance department.

  delete from *instructor*
  where *deptname*= 'Finance';

- Delete all instructors with a salary between $13,000 and $15,000.

  delete from *instructor*
  where *salary* **between** 13000 **and** 15000;

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

  delete from *instructor*
  where *deptname* **in** (**select** *deptname*
  from *department*
  where *building* = 'Watson');

This **delete** request first finds all departments located in Watson, and then deletes all *instructor* tuples pertaining to those departments.

Updates

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used. As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query. Suppose

that annual salary increases are being made, and salaries of all instructors are to be increased by 5percent.We write:

update *instructor*
set *salary*= *salary* * 1.05;

The preceding update statement is applied once to each of the tuples in *instructor* relation. If a salary increase is to be paid only to instructors with salary of less than $70,000, we can write:

update *instructor*
set *salary* = *salary* * 1.05
where *salary* <70000;

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **select**s). As with **insert** and **delete**, a nested **select** within an **update** statement may reference the relation that is being updated. As before, SQL first tests all tuples in the relation to see whether they should be updated, and carries out the updates afterward.

For example, we can write the request "Give a 5 percent salary raise to instructors whose salary is less than average" as follows:

update *instructor*
set *salary* = *salary* * 1.05
where *salary* <(**select avg** (*salary*)
from *instructor*);

Let us now suppose that all instructors with salary over $100,000 receive a3 percent raise, whereas all others receive a 5 percent raise. We could write two **update** statements:

update *instructor*
set *salary* = *salary* * 1.03
where *salary* >100000;
update *instructor*
set *salary* = *salary* * 1.05
where *salary* <= 100000;

Note that the order of the two **update** statements is important. If we changed the order of the two statements, an instructor with a salary just under $100,000 would receive an

over8 percent raise. SQL provides a **case** construct that we can use to perform both the updates with a single **update** statement, avoiding the problem with the order of updates.

update *instructor*
set *salary* = **case**
when *salary* <= 100000 **then** *salary* * 1.05
else *salary* * 1.03
end

The general form of the case statement is as follows.

case
when $pred_1$ **then** $result_1$
when $pred_2$ **then** $result_2$
. . .
when $pred_n$ **then** $result_n$
else $result_0$
end

The operation returns $result_i$, where i is the first of $pred_1$, $pred_2$, . . . , $pred_n$ that is satisfied; if none of the predicates is satisfied, the operation returns $result_0$. Case statements can be used in any place where a value is expected.

To remove a relation from an SQL database, we use the **drop table** command. The **drop table** command deletes all information about the dropped relation from the database.

drop table *r*;

We use the **alter table** command to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **altertable** command is

alter table *r* **add** *A D*;

where *r* is the name of an existing relation, *A* is the name of the attribute to be added, and *D* is the type of the added attribute. We can drop attributes from a relation by the command

alter table *r* **drop column** *A*;

where *r* is the name of an existing relation, and *A* is the name of an attribute of the relation. Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

Lab Exercises:

1. Consider the insurance database given below:

PERSON (driver_id#: varchar(30), name: varchar(50), address:varchar(100)

CAR (regno: varchar(20), model: varchar(30), Year:int)

ACCIDENT (report_number: int, accd_date: date, location: varchar(50))

OWNS (driver_id#: varchar(30), regno: varchar(20))

PARTICIPATED (driver_id#: varchar(30), regno: varchar(20), report_number: int, damage_amount: int)

i. Create the above tables by properly specifying the primary keys and the foreign keys.

ii. Enter at least five tuples for each relation.

(**Hint**: Date format is 'yyyy-mm-dd')

iii. Update the damage amount to 25000 for the car with a specific reg. no in a PARTICIPATED table with report number 12.

iv. Delete the accident and related information that took place in a specific year.
(**Hint**: Command to extract year component from the date attribute is,

extract (year from accd_date))

v. Alter table to add and delete an attribute.

vi. Alter table to add Check constraint.

Additional Exercises:

Consider the following relations for an order processing database application in a company.

CUSTOMER (cust#: int, cname: varchar(50), city: varchar(30))

ORDERS (order#:int, odate: date, cust#: int, ordamt: int)

ITEM (item#: int, unitprice: int)

ORDER_ITEMS (order#:int, qty:int, item#:int)

SHIPMENT (order#: int, warehouse#: int, shipdate: date)

WAREHOUSE (warehouse#:int, city: varchar(30))

i. Create the above tables by properly specifying the primary keys and the foreign keys.

ii. Enter at least five tuples for each relation.

iii. Execute following queries on the database:

   a. Produce a listing: CUSTNAME, No. of Orders, AVG_ORDER_AMT, where the middle column is the total number of orders by the customer and the last column is the average order amount for that customer.

   b. List the order no for the orders that were shipped from all the warehouses that the company has in a specific city.

   c. Decrease the order_amount by 10% if ordered quantity is greater than ten or else by 5% using Case construct.

[OBSERVATION SPACE – LAB 3]

[OBSERVATION SPACE – LAB 3]

[OBSERVATION SPACE – LAB 3]

[OBSERVATION SPACE – LAB 3]

[OBSERVATION SPACE – LAB 3]

[OBSERVATION SPACE – LAB 3]

[OBSERVATION SPACE – LAB 3]

LABNO.: 4 **Date:**

BASIC OPERATIONS OF SQL QUERIES

Objectives:

- To work on basic operations of SQL queries

Introduction:

The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**. The query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result.

Queries on a Single Relation

Let us consider a simple query using Department table created in Lab-3, "Find the names of all departments." Department names are found in the department relation, so we put that relation in the **from** clause. The department name appears in the *name* attribute, so we put that in the **select** clause.

select *deptname*
from *department*;

The result is a relation consisting of a single attribute with the heading *deptname*.

Now consider Instructor table, as follows:

create table *instructor(*
ID **varchar** (5),
name **varchar** (20) **not null**,
deptname **varchar** (20),
salary **numeric** (8,2),
primary key (*ID*),
foreign key (*deptname*) **references** *department*);

The **where** clause allows us to select only those rows in the result relation of the **from** clause that satisfy a specified predicate. Consider the query "Find the names of all instructors in the Computer Science department who have salary greater than $70,000." This query can be written in SQL as:

select *name*
from *instructor*
where *deptname* = 'Comp. Sci.' **and** *salary* >70000;

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 4.1: the *instructor* relation

If the *instructor* relation is as shown in Figure 4.1, then the relation that results from the preceding query is shown in Figure 4.2.

name
Katz
Brandt

Figure 4.2: Result of "Find the names of all instructors in the Computer Science department who have salary greater than $70,000."

SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>. SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

Queries on Multiple Relations

So far our example queries were on a single relation. Queries often need to access information from multiple relations. We now study how to write such queries. For example, suppose we want to answer the query "Retrieve the names of all instructors, along with their department names and department building name."

Looking at the schema of the relation *instructor*, we realize that we can get the department name from the attribute *deptname*, but the department building name is present in the attribute *building* of the relation *department*. To answer the query, each tuple in the *instructor* relation must be matched with the tuple in the *department* relation whose *deptname* value matches the *deptname* value of the *instructor* tuple.

In SQL, to answer the above query,we list the relations that need to be accessed in the **from** clause, and specify the matching condition in the **where** clause. The above query can be written in SQL as

select *name, instructor.deptname, building*
from *instructor, department*
where *instructor.deptname= department.deptname*;

If the *instructor* and *department* relations are as shown in Figures 4.1 and 4.3 respectively, then the result of this query is shown in Figure 4.4.

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 4.3: the *department* relation

Note that the attribute *deptname* occurs in both the relations *instructor* and *department*, and the relation name is used as a prefix (in *instructor.deptname*and *department.deptname*) to make clear to which attribute we are referring. In contrast, the attributes *name* and *building* appear in only one of the relations, and therefore do not need to be prefixed by the relation name. This naming convention *requires* that the relations that are present in the **from** clause have distinct names. We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause.

name	*dept_name*	*building*
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 4.4: Result of "Retrieve the names of all instructors, along with their department names and department building name."

The role of each clause is as follows:

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the **from** clause.

A typical SQL query has the form

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P;

Each A_i represents an attribute, and each r_ia relation. P is a predicate. If the **where** clause is omitted, the predicate P is **true**. Although the clauses must be written in the order **select**, **from**, **where**, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first **from**, then **where**, and then **select**.

The **from** clause by itself defines a Cartesian product of the relations listed in the clause. The result relation has all attributes from all the relations in the **from** clause. Since the same attribute name may appear in both r_i and r_j, as we saw earlier, we prefix the name of the relation from which the attribute originally came, before the attribute name.

For example, the relation schema for the Cartesian product of relations *instructor* and *department* is:*(instructor.ID, instructor.name, instructor.deptname, instructor.salary,department.deptname,department.building,department.budget*)

With this schema, we can distinguish *instructor.deptname*from *department.deptname*. The Cartesian product by itself combines tuples from *instructor* and *department* that are unrelated to each other. Each tuple in *instructor* is combined with *every* tuple in *department*, even those that refer to a different instructor. The result can bean extremely large relation, and it rarely makes sense to create such a Cartesian product.

Instead, the predicate in the **where** clause is used to restrict the combinations created by the Cartesian product to those that are meaningful for the desired answer. We would expect a query involving *instructor* and *department* to combine a particular tuple *t* in *instructor* with only those tuples in *department* that refer to the same department to which *t* refers. That is, we wish only to match *department* tuples with *instructor* tuples that have the same deptnamevalue. The following SQL query ensures this condition.

select *name, instructor.deptname, building*
from *instructor, department*
where *instructor.deptname= department.deptname*;

Additional Basic Operations

The Rename Operation

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause. However, we cannot always derive names in this way, for several reasons:

First, two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result. Second, if we used an arithmetic expression in the **select** clause, the resultant attribute does not have a name. Third, even if an attribute name can be derived from the base relations as in the preceding example, we may want to change the attribute name in the result. Hence, 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 can appear in both the **select** and **from** clauses. For example, if we want the attribute name *name* to be replaced with the name *instructor name*, we can write the query as:

select *name* **as** *instructor_name, course_id*
from *instructor, teaches*
where *instructor.ID= teaches.ID*;

The teaches relation schema is as follows: *teaches* (*ID, course id, sec id, semester, year*).

The **as** clause is particularly useful in 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';

Observe that we could not use the notation *instructor.salary*, since it would not be clear which reference to *instructor* is intended.

In the above query, *T* and *S* can be thought of as copies of the relation *instructor*, but more precisely, they are declared as aliases, that is as alternative names, for the relation *instructor*. An identifier, such as *T* and *S*, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

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 My SQL 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. There are variations on the exact set of string functions supported by different database systems. 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.

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all departments whose building name includes the substring 'Watson'." This query can be written as:

select *deptname*
from *department*
where *building* **like** '%Watson%';

For patterns to include the special pattern characters (that is,%and), SQL allows the specification of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a **like** comparison using the **escape** keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

- **like** 'ab\%cd%' **escape** '\' matches all strings beginning with "ab%cd".
- **like** 'ab\\cd%' **escape** '\' matches all strings beginning with "ab\cd".

SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator.

Attribute Specification in Select Clause

The asterisk symbol " * " can be used in the **select** clause to denote "all attributes." Thus, the use of *instructor*.* in the **select** clause of the query:

select *instructor*.*
from *instructor, teaches*
where *instructor.ID= teaches.ID*;

indicates that all attributes of *instructor* are to be selected. A **select** clause of the form **select** * indicates that all attributes of the result relation of the **from** clause are selected.

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 *deptname* = '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 several instructors 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**;

Where Clause Predicates

SQL includes a **between** comparison operator to simplify **where** clauses which specifies that a value be less than or equal to some value and greater than or equal to some other value. If we wish to find the names of instructors with salary amounts between $90,000 and $100,000, we can use the **between** comparison to write:

select *name*
from *instructor*
where *salary* **between** 90000 **and** 100000;

instead of:

select *name*
from *instructor*
where *salary* <= 100000 **and** *salary* >= 90000;

Similarly, we can use the **not between** comparison operator. SQL permits us to use the notation (*v*1, *v*2, . . . , *vn*) to denote a tuple of arity *n* containing values *v*1, *v*2, . . . , *vn*. The comparison operators can be used on tuples, and the ordering is defined lexicographically. For example, (*a*1, *a*2) <= (*b*1, *b*2) is true if *a*1 <= *b*1 **and** *a*2 <= *b*2; similarly, the two tuples are equal if all their attributes are equal. Thus, the preceding SQL query can be rewritten as follows:

select *name, course_id*
from *instructor, teaches*
where (*instructor.ID, deptname*) = (*teaches.ID,* 'Biology');

Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −. We shall now construct queries involving the **union**, **intersect**, and **except** operations over two sets. Consider each course in a university which may be offered multiple times, across different semesters, or even within a semester. We need a relation schema to describe each individual offering, or section, of the class in the university. The schema is *section* (*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*) Figure 4.5 shows a sample instance of the *section* relation.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 4.5. The *section* relation

The Union Operation

To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both, we write:

```
(select course_id
from section
where semester = 'Fall' and year= 2009)
union
(select course_id
from section
where semester = 'Spring' and year= 2010);
```

The **union** operation automatically eliminates duplicates, unlike the **select** clause. Thus, using the *section* relation, where two sections of CS-319 are offered in Spring 2010, and a section of CS-101 is offered in the Fall 2009 as well as in the Fall 2010 semester, CS-101 and CS-319 appear only once in the result. If we want to retain all duplicates, we must write **union all** in place of **union**:

(**select** *course_id*
from *section*
where *semester* = 'Fall' **and** *year*= 2009)
union all
(**select** *course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010);

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both input sets. So, in the above query, each of CS-319and CS-101 would be listed twice.

The Intersect Operation

To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

(**select** *course_id*
from *section*
where *semester* = 'Fall' **and** *year*= 2009)
intersect
(**select** *course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010);

The result relation, contains only one tuple with CS-101. The **intersect** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **intersect all** in place of **intersect**.

The Minus Operation

To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

(**select** *course_id*
from *section*
where *semester* = 'Fall' **and** *year*= 2009)

minus
(**select** *course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010);

The result of this query contains CS-347 and PHY-101. The **minus** operation outputs all tuples from its first input that do not occur in the second input; that is, it performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference. If we want to retain duplicates, we must write **minus all** in place of **minus**:

Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations. The result of an arithmetic

expression (involving, for example +, −, ∗, or /) is null if any of the input values is null. For example, if a query has an expression *r.A*+ 5, and *r.A* is null for a particular tuple, then the expression result must also be null for that tuple. Comparisons involving nulls are more of a problem.

For example, consider the comparison "1 <**null**". It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, "**not** (1 <**null**)" would evaluate to true, which does not make sense. SQL therefore treats as **unknown** the result of any comparison involving a *null* value. This creates a third logical value in addition to *true* and *false*.

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.

- **and**: The result of *true* **and** *unknown* is *unknown*, *false* **and** *unknown* is *false*, while *unknown* **and** *unknown* is *unknown*.
- **or**: The result of *true* **or** *unknown* is *true*, *false* **or** *unknown* is *unknown*, while *unknown* **or** *unknown* is *unknown*.
- **not**: The result of **not** *unknown* is *unknown*.

If the **where** clause predicate evaluates to either **false** or **unknown** for a tuple, that tuple is not added to the result.

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the *instructor* relation with null values for *salary*, we write:

select *name*
from *instructor*
where *salary* **is null**;

The predicate **is not null** succeeds if the value on which it is applied is not null.

Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

Basic Aggregation

Consider the query "Find the average salary of instructors in the Computer Science department". We write this query as follows:

select avg (*salary*)
from *instructor*
where *deptname*= 'Comp. Sci.';

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department. The database system may give an arbitrary name to the result relation attribute that is generated by aggregation; however, we can give a meaningful name to the attribute by using the **as** clause as follows:

select avg (*salary*) **as** *avg salary*
from *instructor*
where *deptname*= 'Comp. Sci.';

In the *instructor* relation of Figure 4.1, the salaries in the Computer Science department are$75,000, $65,000, and $92,000. The average balance is $232,000/3= $77,333.33.Retaining duplicates is important in computing an average. Suppose the Computer Science department adds a fourth instructor whose salary happens to be $75,000. If duplicates were eliminated, we would obtain the wrong answer ($232,000/4 = $58.000) rather than the correct answer of $76,750.There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the key word **distinct** in the aggregate expression. An example arises in the query "Find the total number of instructors who teach a course in the Spring 2010 semester." In this case, an instructor counts only once, regardless of the number of course sections that the instructor teaches. The required information is contained in the relation *teaches*, and we write this query as follows:

select count (**distinct** *ID*)
from *teaches*
where *semester* = 'Spring' **and** *year* = 2010;

Because of the keyword **distinct** preceding *ID*, even if an instructor teaches more than one course, she is counted only once in the result. We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count** (*). Thus, to find the number of tuples in the *instructor* relation, we write

select count (*)
from *instructor*;

SQL does not allow the use of **distinct** with **count** (*). It is legal to use **distinct** with **max** and **min**, even though the result does not change. We can use the keyword **all** in place of **distinct** to specify duplicate retention, but, since **all** is the default, there is no need to do so.

Aggregation with Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average salary in each department."We write this query as follows:

select *deptname*, **avg** (*salary*) **as** *avg salary*
from *instructor*
group by *deptname*;

Figure 4.6 shows the tuples in the *instructor* relation grouped by the *deptname* attribute, which is the first step in computing the query result. The specified aggregate is computed for each group, and the result of the query is shown in Figure 4.7.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 4.6: the tuples in the *instructor* relation grouped by the *deptname* attribute

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 4.7: The result relation for the query "Find the average salary in each department".

In contrast, consider the query "Find the average salary of all instructors." We write this query as follows:

select avg (*salary*)
from *instructor*;

In this case the **group by** clause has been omitted, so the entire relation is treated as a single group.

The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used. We express this query in SQL as follows:

select *deptname*, **avg** (*salary*) **as** *avg salary*
from *instructor*
group by *deptname*
having avg (*salary*) >42000;

The result is shown in Figure 4.8.

dept_name	*avg(avg_salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 4.8: The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is treated as erroneous.

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.
4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "For each course section offered in 2009, find the average total credits (*tot cred*) of all students enrolled in the section, if the section had at least 2 students.", using *takes* and *student* schema as given below.

student (*ID, name, deptname, tot cred)*
takes (ID, course_id,sec_id, semester, year, grade)

Note: Underlined attributes together form the Primary key of the schema.

select *course_id, semester, year, sec id,* **avg** (*tot cred*)
from *takes* **natural join** *student*
where *year* = 2009
group by *course_id, semester, year, sec_id*
having count (*ID*) >= 2;

Note that all the required information for the preceding query is available from the relations *takes* and *student*, and that although the query pertains to sections, a join with *section* is not needed.

Aggregation with Null and Boolean Values

Aggregate functions treat nulls according to the following rule: All aggregate functions except **count (*)** ignore null values in their input collection. As a result of null values being ignored, the collection of values may be empty. The **count** of an empty collection

is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection.

Lab Exercises:

For the insurance database demonstrate how you

1. Find the **total** number of people who owned cars that were involved in accidents in 2008.
2. Find the number of accidents in which cars belonging to a specific model were involved.
3. Produce a listing with **header as** OWNER_NAME, **No. of** Accidents, and **Total Damage** Amount in a descending **order** on total damage.
4. List the Owners who made **more than 2** accidents in a year.
5. List the owners who are **not involved** in any accident.

Additional Exercises:

1. For the order processing database demonstrate how you

   a. Display the names of the customers who have purchased items on 09/05/2015.
   b. Count the total number of items in each order.
   c. Find the order with maximum number of items in it.
   d. Find the date on which maximum number of orders were shipped.
   e. Demonstrate the handling of data in ORDER_ITEMS relation on deletion of any item from the ITEM relation.
   f. List the order no for the orders that were shipped from all the warehouses that the company has in a specific city.
   g. List the customer with a specific surname.
   h. List the customers in descending order of their total order amount.
   i. Identify the customer with at least three orders that shipped on the particular date.

[OBSERVATION SPACE -LAB 4**]**

[OBSERVATION SPACE – LAB 4**]**

[OBSERVATION SPACE – LAB 4**]**

[OBSERVATION SPACE – LAB 4**]**

[OBSERVATION SPACE – LAB 4**]**

[OBSERVATION SPACE – LAB 4**]**

LAB NO.: 5 **Date:**

NESTED SUBQUERIES

Objectives:

- To work on nested subquery concept of SQL.

Introduction:

Nested Sub queries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is *to perform tests for set membership, make set comparisons, and determine set cardinality,* by nesting subqueries in the **where** clause.

Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership. As an illustration, reconsider the query "Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters." Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall2009 and the set of courses taught in Spring 2010.We can take the alternative approach of finding all courses that were taught in Fall 2009 and that are also members of the set of courses taught in Spring 2010. Clearly, this formulation generates the same results as the previous one did, but it leads us to write our query using the **in** connective of SQL.

We begin by finding all courses taught in Spring 2010, and we write the subquery

(**select***course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010)

We then need to find those courses that were taught in the Fall 2009 and thatappear in the set of courses obtained in the subquery. We do so by nesting the subquery in the **where** clause of an outer query. The resulting query is

select distinct *course_id*
from *section*

where *semester* = 'Fall' **and** *year*= 2009 **and**
course id **in** (**select** *course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010);

This example shows that it is possible to write the same query several ways in SQL. This flexibility is beneficial, since it allows a user to think about the query in the way that seems most natural.

We use the **not in** construct in a way similar to the **in** construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010semester, we can write:

select distinct *course_id*
from *section*
where *semester* = 'Fall' **and** *year*= 2009 **and**
course_id **not in** (**select** *course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010);

The **in** and **not in** operators can also be used on enumerated sets. The following query selects the names of instructors whose names are neither "Mozart" nor "Einstein".

select distinct *name*
from *instructor*
where *name* **not in** ('Mozart', 'Einstein');

In the preceding examples, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. For example, we can write the query "find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011" as follows:

select count (**distinct** *ID*)
from *takes*
where (*course_id, sec_id, semester, year*) **in** (**select** *course_id, sec_id, semester, year*
from *teaches*
where *teaches.ID*= 10101);

Set Comparison: As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."

select *name*
from *instructor*
where *salary* >**some** (**select** *salary*
from *instructor*
where *deptname* = 'Biology');

The subquery:

(**select** *salary*
from *instructor*
where *deptname* = 'Biology')

generates the set of all salary values of all instructors in the Biology department.

The phrase "greater than at least one" is represented in SQL by >**some**. The >**some** comparison in the **where** clause of the outer **select** is true if the *salary* value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology. SQL also allows <**some**, <= **some**, >= **some**, = **some**, and <>**some** comparisons. As an exercise, verify that = **some** is identical to **in**, whereas <>**some** is *not* the same as **not in**.

Now we modify our query slightly. Let us find the names of all instructors who have a salary value greater than that of each instructor in the Biology department. The construct>**all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

select *name*
from *instructor*
where *salary* >**all** (**select** *salary*
from *instructor*
where *dept name* = 'Biology');

As it does for **some**, SQL also allows <**all**, <= **all**, >= **all**, = **all**, and <>**all** comparisons. As an exercise, verify that <>**all** is identical to **not in**, whereas =**all** is *not* the same as **in**.

As another example of set comparisons, consider the query "Find the departments that have the highest average salary. "We begin by writing a query to find all average

salaries, and then nest it as a subquery of a larger query that finds those departments for which the average salary is greater than or equal to all average salaries:

```
select deptname
from instructor
group by deptname
having avg (salary) >= all (select avg (salary)
                            from instructor
                            group by deptname);
```

Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty. Using the **exists** construct, we can write the query "Find all courses taught in both the Fall 2009semester and in the Spring 2010 semester" in still another way:

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
exists (select *
        from section as T
        where semester = 'Spring' and year= 2010 and
        S.course_id= T.course_id);
```

The above query also illustrates a feature of SQL where a correlation name from an outer query (*S* in the above query), can be used in a subquery in the **where** clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery**. We can test for the nonexistence of tuples in a subquery by using the **not exists** Construct. To illustrate the **not exists** operator, consider the query "Find all students who have taken all courses offered in the Biology department." Using the **except** construct, we can write the query as follows:

```
select distinct S.ID, S.name
from student as S
where not exists ((select course_id
                   from course
where deptname = 'Biology')
except
```

(**select** *T.course_id*
from *takes* **as** *T*
where *S.ID = T.ID*));

Here, the subquery:

(**select** *course_id*
from *course*
where *deptname* = 'Biology')

finds the set of all courses offered in the Biology department. The subquery:

(**select** *T.course_id*
from *takes* **as** *T*
where *S.ID = T.ID*)

finds all the courses that student *S.ID* has taken. Thus, the outer **select** takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department.

Test for the Absence of Duplicate Tuples

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query "Find all courses that were offered at most once in 2009" as follows:

select *T.course_id*
from *course* **as** *T*
where unique (**select** *R.course_id*
from *section* **as** *R*
where *T.course_id= R.course_id* **and**
R.year = 2009);

Note that if a course is not offered in 2009, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set.

Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that

a relation can appear. Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000." We wrote this query using the **having** clause. We can now rewrite this query, without using the **having** clause, by using a subquery in the **from** clause, as follows:

select *deptname, avg_salary*
from (**select** *dept_name,* **avg** (*salary*) **as** *avg salary*
from *instructor*
group by *deptname*)
where *avg_salary* >42000;

The subquery generates a relation consisting of the names of all departments and their corresponding average instructors' salaries. The attributes of the subquery result can be used in the outer query, as can be seen in the above example. Note that we do not need to use the **having** clause, since the subquery in the **from** clause computes the average salary, and the predicate that was in the **having** clause earlier is now in the **where** clause of the outer query. We can give the subquery result relation a name, and rename the attributes using the **as** clause, as illustrated below.

select *deptname, avg_salary*
from (**select** *deptname,* **avg** (*salary*)
from *instructor*
group by *deptname*)
as *deptavg* (*deptname, avg_salary*)
where *avg_salary* >42000;

The subquery result relation is named *deptavg*, with the attributes *deptname* and *avg_salary*.

As another example, suppose we wish to find the maximum across all departments of the total salary at each department. The **having** clause does not help us in this task, but we can write this query easily by using a subquery in the **from** clause, as follows:

select max (*tot salary*)
from (**select** *deptname,* **sum**(*salary*)
from *instructor*
group by *deptname*) **as** *dept total* (*deptname, tot salary*);

The with Clause

The **with** clause provides away of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which finds those departments with the maximum budget.

with *max_budget* (*value*) **as**
(**select max**(*budget*)
from *department*)
select *budget*
from *department, max_budget*
where *department.budget = max budget.value*;

The **with** clause defines the temporary relation *max_budget*, which is used in the immediately following query. The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.

Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries**. For example, a subquery can be used in the s**elect** clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

select *deptname*,
(**select count**(*)
from *instructor*
where *department.deptname = instructor.deptname*)
as *num_instructors*
from *department*;

The subquery in the above example is guaranteed to return only a single value since it has a **count**(*) aggregate without a **group by**. The example also illustrates the usage of correlation variables, that is, attributes of relations in the **from** clause of the outer query, such as *department.deptname* in the above example. Scalar subqueries can occur in s**elect**, **where**, and **having** clauses. Scalar subqueries may also be defined without aggregates. It is not always possible to figure out at compile time if a subquery can return more than one tuple in its result; if the result has more than one tuple when the subquery is executed, a run-time error occurs.

Lab Exercises:

1. Consider the following database of student enrollment in courses and books adopted for each course

   STUDENT (regno: varchar(20), name: varchar(50), major: varchar(20), bdate:date)

   COURSE (course#:int, cname: varchar(30), dept: varchar(30))

   ENROLL (regno: varchar(20), course#:int, sem:int, book_isbn:int)

   BOOK_ADOPTION (course#:int, sem:int, book_isbn:int)

   TEXT (book_isbn:int, booktitle: varchar(50), publisher: varchar(50), author: varchar(50))

   i. Create the above tables by properly specifying the primary keys and the foreign keys.

   ii. Enter at least five tuples for each relation.

   iii. Execute following queries on the database using nested subquery concept:

      a. List the courses which uses more than 1 text book.

      b. List the departments whose all course text books are published by a particular publisher.

      c. Find the students who have enrolled for course of more than one department

      d. Produce a list of students who are not enrolled.

      e. List the department which adopts all the books from the particular publisher.

      f. List the books which are adopted by the course as well as enrolled by the student.

      g. List the courses which has adapted at least two books from a specific publisher.

      h. Identify the students who are enrolled for maximum number of books.

      i. List the publishers along with the number of books published by them.

      j. List the students who enrolled for all the books adopted by their course.

Additional Exercises:

For the order processing database demonstrate using nested subquery how you

i. Find the customers whose address is not known yet.
ii. Find the customer who has made a maximum purchase till date.
iii. Display all those orders which have not been shipped yet.
iv. Select all those items which have not been bought by any customer.
v. Find the item which has been bought by most of the customers.
vi. List the orders which are not shipped on a particular date.
vii. Identify the customers whose orders are shipped on a both specified days.
viii. Give the customer information whose all orders are shipped from a single warehouse.
ix. List the customers with maximum number of orders.

[OBSERVATION SPACE – LAB 5**]**

[OBSERVATION SPACE – LAB 5**]**

[OBSERVATION SPACE – LAB 5**]**

[OBSERVATION SPACE – LAB 5**]**

[OBSERVATION SPACE – LAB 5**]**

LAB NO.: 6 **Date:**

PROCEDURAL LANGUAGE

Objectives:

- To learn how to use PL/SQL concepts.

Introduction to PL/SQL:

PROCEDURES

A procedure is a module performing one or more actions; it does not need to return any values. The syntax for creating a procedure is as follows:

```
CREATE OR [REPLACE] PROCEDURE name
        [(parameter[, parameter, ...])]
AS
        [local declarations]
BEGIN
        executable statements
[EXCEPTION
        exception handlers]
END [name];
/
```

- A procedure may have 0 to many parameters.
- Every procedure has two parts:
  1. The header portion, which comes before **AS** (sometimes you will see IS—they are interchangeable), keyword (this contains the procedure name and the parameter list),
  2. The body, which is everything after the AS keyword.
- The word REPLACE is optional.

When the word REPLACE is not used in the header of the procedure, in order to change the code in the procedure, it must be dropped first and then re-created.

Example:

```
CREATE OR REPLACE PROCEDURE insert Person
(id IN VARCHAR, dob IN DATE, fname IN VARCHAR,  lname IN VARCHAR)  IS
```

```
        counter  INTEGER;          --declaration part
BEGIN
        SELECT COUNT(*) INTO counter FROM person p WHERE        p.pid = id;
        IF (counter > 0) THEN
        -- person with the given pid already exists
DBMS_OUTPUT.PUT_LINE ('WARNING Inserting person: person with pid ' || id || '
already exists!');
        ELSE
        INSERT INTO person VALUES (id, DOB, fname, lname);
                DBMS_OUTPUT.PUT_LINE ('Person with pid ' || id || ' is inserted.');
        END IF;
        /
```

- In order to execute a procedure in SQL*Plus use the following syntax:
  EXECUTE Procedure_name
  SQL> EXECUTE insertPerson (‘p1’, ’10-10-2000’, ‘John’,’Smith’);

PARAMETERS

- Parameters are the means to pass values to and from the calling environment to the server.
- These are the values that will be processed or returned via the execution of the procedure.
- There are three types of parameters:
- IN, OUT, and IN OUT.
- Modes specify whether the parameter passed is read in or a receptacle for what comes out.
- IN passes value into the procedure, OUT passes back from the procedure and INOUT does both.

FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL **block that *returns* a single value.**
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.

- The data type of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: It must be used in some context. You can think of it as a sentence fragment.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.
- The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
        (parameter list)
        RETURN datatype
IS
BEGIN
        <body>
        RETURN (return_value);
END;
/
```

- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.
- The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).
- A function may have IN, OUT, or IN OUT parameters.

```
CREATE OR REPLACE FUNCTION show_description
        (i_course_no  IN  number)
RETURN varchar2
AS
        v_description varchar2(50);
BEGIN
        SELECT description
                INTO v_description
                FROM course
                WHERE course_no = i_course_no;
        RETURN v_description;
EXCEPTION
```

```
        WHEN NO_DATA_FOUND
        THEN
                RETURN('The Course is not in the database');
        WHEN OTHERS
        THEN
                RETURN('Error in running show_description');
  END;
/
```

Making Use Of Functions

- **Using an anonymous block**

```
SET SERVEROUTPUT ON     -- Setting the server output on
DECLARE
        v_description
        VARCHAR2(50); BEGIN
        v_description := show_description(&sv_cnumber);
        DBMS_OUTPUT.PUT_LINE(v_description);
END;
/
```

- **Using SQL statements**

```
SELECT course_no, show_description(course_no)
FROM course;
        OR
SELECT show_description(course_no) FROM Dual;
```

Where Dual is dummy table.

TRIGGERS

A database trigger is a stored PL/SQL program unit associated with a specific database table. ORACLE executes (fires) a database trigger automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table. Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.

Database triggers can be used to perform any of the following:

- Audit data modification
- Log events transparently
- Enforce complex business rules
- Implement complex security authorizations
- You can associate up to 12 database triggers with a given table.
- A database trigger has three parts:
- A **triggering event**, an **optional trigger constraint**, and a **trigger action**.
- When an event occurs, a database trigger is fired, and a predefined PL/SQL block will perform the necessary action.

SYNTAX:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER} triggering_event ON table_name
[FOR EACH ROW]
DECLARE
   Declaration statements
BEGIN
   Executable statements
EXCEPTION
   Exception-handling statements
END;
/
```

The trigger_name references the name of the trigger. **BEFORE** or **AFTER** specify when the trigger is fired (before or after the triggering event). The triggering_event references a DML statement issued against the table (e.g., INSERT,DELETE, UPDATE). The table_name is the name of the table associated with the trigger.

The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row. Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.

Triggers may be called **BEFORE** or **AFTER** the following events:

INSERT, UPDATE and DELETE. The before/after options can be used to specify when the trigger body should be fired with respect to the triggering statement. If the user indicates a BEFORE option, then Oracle fires the trigger before executing the triggering statement. On the other hand, if an AFTER is used, Oracle fires the trigger after executing the triggering statement. A trigger may be a ROW or STATEMENT type. If the statement FOR EACH ROW is present in the CREATE TRIGGER clause of a trigger, the trigger is a row trigger. A row trigger is fired for each row affected by an triggering statement. A statement trigger, however, is fired only once for the triggering statement, regardless of the number of rows affected by the triggering statement

Example: statement trigger

```
CREATE OR REPLACE TRIGGER mytrig1
BEFORE DELETE OR INSERT OR UPDATE ON employee
BEGIN
IF(TO_CHAR(SYSDATE, 'day') IN ('sat', 'sun')) OR
(TO_CHAR(SYSDATE,'hh:mi') NOT BETWEEN '08:30' AND '18:30')
THEN RAISE_APPLICATION_ERROR(-20500, 'table is secured');
END IF;
END;
/
```

The above example shows a trigger that limits the DML actions to the employee table to weekdays from 8.30am to 6.30pm. If a user tries to insert/update/delete a row in the EMPLOYEE table, a warning message will be prompted.

Example: ROW Trigger

```
CREATE OR REPLACE TRIGGER mytrig2
AFTER DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
BEGIN
IF DELETING THEN
INSERT INTO xemployee (emp_ssn, emp_last_name,emp_first_name, deldate)
VALUES (:old.emp_ssn, :old.emp_last_name,:old.emp_first_name, sysdate);
ELSIF INSERTING THEN
```

```
INSERT INTO nemployee (emp_ssn, emp_last_name,emp_first_name,
adddate)
VALUES (:new.emp_ssn, :new.emp_last_name,:new.emp_first_name, sysdate);
ELSIF UPDATING('emp_salary') THEN
INSERT INTO cemployee (emp_ssn, oldsalary, newsalary, up_date)
VALUES (:old.emp_ssn,:old.emp_salary, :new.emp_salary, sysdate);
END IF;
END;
/
```

:OLD and :NEW

- When a DML statement changes a column the old and new values are visible to the executing code
- This is done by prefixing the table column with :old or :new
- **:new** is useful for INSERT and UPDATE
- **:old** is useful for DELETE and UPDATE
- triggers may fire other triggers in which case they are CASCADING. Try not to create too many interdependencies with triggers!

```
CREATE OR REPLACE TRIGGER faculty_after_update_row
AFTER UPDATE ON faculty EDB
    FOR EACH ROW
    BEGIN
      IF UPDATING ('dept') AND :old.dept <> :new.dept
      THEN UPDATE department SET chair = NULL WHERE chair = :old.pid;
END IF;
END;
/

DECLARE
            percent_id      agents.percent%TYPE;
BEGIN
    SELECT percent INTO percent_id FROM agents WHERE aid = 'a02';
    IF percent_id > 0 THEN
    INSERT INTO agents (aid, aname, city) VALUES ('a07', 'John', 'Corpus');
    END IF;
```

```
    END;
/
```

- The previous trigger is used to keep track of all the transactions performed on the employee table. If any employee is deleted, a new row containing the details of this employee is stored in a table called x employee. Similarly, if a new employee is inserted, a new row is created in another table called n employee, and so on.
- Note that we can specify the old and new values of an updated row by prefixing the column names with the :OLD and :NEW qualifiers.

ENABLING, DISABLING, DROPPING TRIGGERS

```
SQL>ALTER TRIGGER trigger_name DISABLE;
SQL>ALTER TABLE table_name DISABLE ALL TRIGGERS;
SQL>ALTER TABLE table_name ENABLE trigger_name;
SQL> ALTER TABLE table_name ENABLE ALL TRIGGERS;
SQL> DROP TRIGGER trigger_name
```

VC# code snippet to execute PL/SQL procedure

```
OracleCommand c= new OracleCommand("Procedure name", connObject);
c.CommandText="Procedure name":
c.CommandType=CommandType.StoredProcedure;
c.ExecuteNonQuery();
```

CURSOR

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active* set. There are two types, explicit cursor and implicit cursor. User created cursor is called explicit cursor. Implicit cursors are automatically created by the Oracle whenever an SQL statement is executed. Programmers cannot control the implicit cursors and the information in it.

Example: Cursor declaration

DECLARE CURSOR emp_cur IS

*SELECT * FROM emp_tbl WHERE salary > 5000;*

In the above example we are creating a cursor 'emp_cur' on a query which returns there cords of all the employees with salary greater than 5000.

Once the cursor is created in the declaration section we can access the cursor in the execution section of the PL/SQL program. There are three steps involved in accessing the cursor, which are mentioned below,

- Open the cursor.
- Fetch the records in the cursor one at a time.
- Close the cursor.

General form of using a cursor

```
DECLARE
        Variables;
        Records;
        Create a Cursor;
BEGIN
        OPEN Cursor;
        FETCH Cursor;
                Process the records;
        CLOSE Cursor;
END;
/
```

Example: CURSOR

Give the details of the persons who are involved in more than two accidents in a year 2015.

```
CREATE OR REPLACE PROCEDURE Disp_person IS
CURSOR emp_cur IS
        Select * from Person p1
        Where p1.driver_id#  IN  (Select unique p.driver_id#
                        From Personp, Accident a, Participatedpa
                        Where p.driver_id# = pa.driver_id# and
                        pa.report_number = a.report_number and
```

```
                    Extract (Yearfrom a.accd_date)=2015
                    Group by p.driver_id# Having count(*) > 2 );
emp_rec emp_cur%rowtype;
BEGIN
OPEN emp_cur;
LOOP
FETCH emp_cur INTO emp_rec;
EXIT WHEN emp_cur%NOTFOUND;
    dbms_output.put_line(emp_rec.driver_id# || ' ' || emp_rec.name || ' '
    ||emp_rec.address);
END LOOP;
CLOSE emp_cur;
END;
/
```

Lab Exercises:

1. Submission of the abstract for the database mini project.
2. Generate a trigger displaying driver information, on participating in an accident
3. Create a function to return total number of accidents happened in a particular year.
4. Create a procedure to display total damage caused due to an accident for a particular driver on a specific year.
5. Create a procedure to display accident information which took place in a particular location.

Additional exercises:

1. Write a PL/SQL function to withdraw money from the bank account.
2. Generate a trigger intimating the driver regarding the accidents if the number of accidents by the driver reaches the count of two.
3. Create a procedure to display 20% discount amount on each order provided order has at least five items.

[OBSERVATION SPACE – LAB 6**]**

[OBSERVATION SPACE – LAB 6**]**

[OBSERVATION SPACE – LAB 6**]**

[OBSERVATION SPACE – LAB 6**]**

[OBSERVATION SPACE – LAB 6**]**

[OBSERVATION SPACE – LAB 6**]**

LAB NO.: 7 **Date:**

E-R MODEL AND USER INTERFACE DESIGN

Objectives:

- To understand and able to draw the E-R diagram for the given problem statement.

Introduction:

Sample E-R Diagram for Hospital Management System:

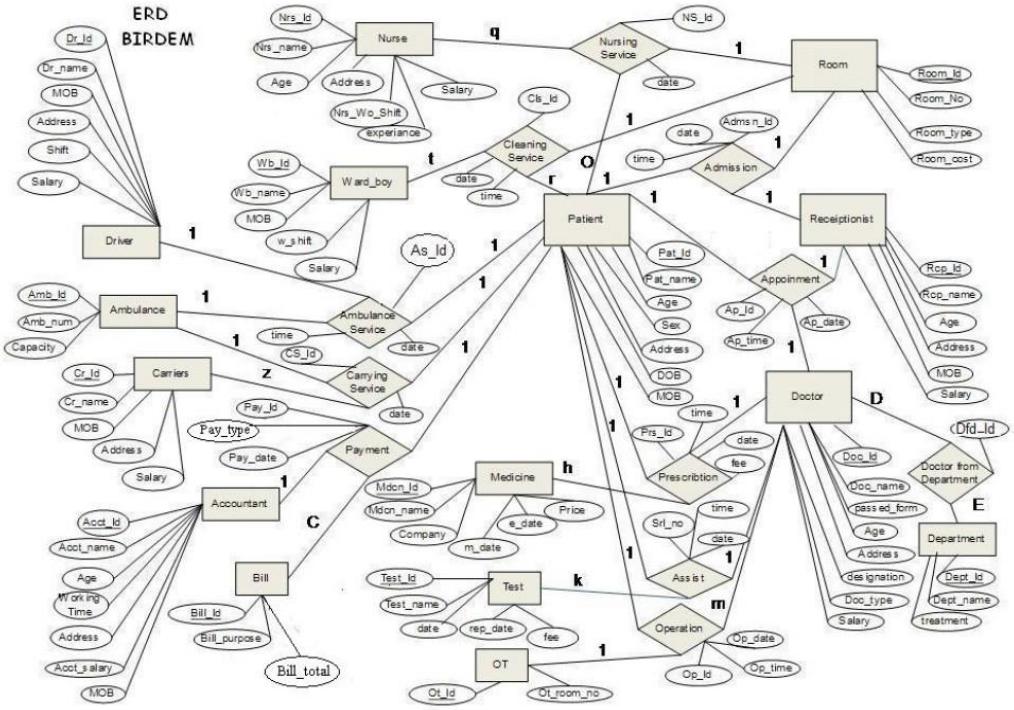

Figure 7.1: E-R Diagram for Hospital Management System**.**

Lab exercises:

1. Submission of ER diagram of the mini project
2. Submit the design of the front end of the mini project
3. Specifications for the information to be retrieved from the database. Ex: When the Author's name is specified, all the text books written by him should be retrieved.

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

LAB NO.: 8 **Date:**

DATA ACCESS FROM VC#

Objectives:

- To connect the front end to the back end database.

Introduction:

Database connectivity steps for Visual Studio 2012 version.

Select **Tools** Menu item

Select **Connect to database** option

A new window with name **Add connection** should open,

Then click on **change** button **to** change the **Data Source** which will redirect to the window with the name **change data source**.

In the Change Data Source window, choose **Oracle Database** as the **Data Source.**

Set **Data Provider** to **.NET Framework Data Provider for Oracle**

Click on **OK**. You will be redirected to **Add Connection Window.**

Set **Server Name** to **ictorcl**

Set **User Name** to **database login id** (e.g it1234, without using @orcl)

Set **Password** to 'student'

Then click on **Test Connection.** You should get a message saying **Test Connection Succeeded.**

Then Click on **OK.**

Right click on **Data connection** option under **Server Explorer**

Copy **Connection string** value of the **Properties** parameter, which is to be used in coding section.

Add **Oracle.DataAccess** package into the project which is available at **References->Assemblies->Extensions** option.

In the solution explorer there is field named **References**. Expand it and check for **Oracle. Data Access**. If it is not present, then **right click on References** and click on **Add Reference** (or Go to Project and select Add Reference). Reference manager window will be opened. Select **Extensions** under **Assemblies** which lists out various component names along with their versions. Select **Oracle. Data Access** from the list. (Two versions will be listed- 2.112.3.0and 4.112.3.0. You can choose any one of it by ticking the checkbox.) Click on **OK**.

Code for Connecting the GUI to the oracle database.

```
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;
namespace StudentDetails
{
public partial class Form1 : Form
{
OracleConnection conn;
OracleCommand comm;
OracleDataAdapter da;
DataSet ds;
DataTable dt;
DataRow dr;
int i = 0;
public Form1()
{
InitializeComponent();
}

// Click on Close button should close the complete application.
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
DialogResult dr = MessageBox.Show("Are you sure you want to exit the
Application?", "Exit", MessageBoxButtons.YesNoCancel);
if (dr == DialogResult.Yes)
//e.Cancel = true;
Application.Exit();
}
```

```
//Connecting to the database through the connection string
public void connect1()
{
String oradb = "Data Source=Oracle server name;User ID=Oracle login ID;Password=student";
conn = new OracleConnection(oradb); // C#
conn.Open();
}
```

NOTE:

string oradb = "Data Source=Oracle Server Name; User ID=Oracle login ID; Password=password"; this is called connection string. This can be written manually if you know the data source, user id and password beforehand. If not known then, it can be obtained by following the steps: Once you open your project on the left palette, there will be a tab **Data Source.** Click on that will show you another option **Add new Data Source.** This can also be obtained by selecting the **Data** tab in the menu, under which you get Add new data source.

When clicked on that option you get a dialog box **Data Source Configuration Wizard.** Select **Database -> Next -> Data Set.** Now you can see an option **Connection String.** Click on '+' sign to expand the view. Copy the connection string which is available when you expand the view. After getting the connection string **DONOT CLICK ON FINISH.** Click cancel.

```
//On Button click connects to database and fetches the first tuple data
private void Connect_Click(object sender, EventArgs e)
{
connect1();
comm = new OracleCommand();
comm.CommandText = "select * from instructor";
comm.CommandType = CommandType.Text;
ds = new DataSet();
da = new OracleDataAdapter(comm.CommandText, conn);
da.Fill(ds, "instructor");
dt = ds.Tables["instructor"];
int t = dt.Rows.Count;
MessageBox.Show(t.ToString());
dr = dt.Rows[i];
textBox1.Text = dr["id"].ToString();
textBox2.Text = dr["name"].ToString();
textBox3.Text = dr["deptname"].ToString();
textBox4.Text = dr["salary"].ToString();
conn.Close();
}
//On Next click it displays the next tuple in the database. And repeats in a loop once
it reaches the last tuple.
private void Next_Click(object sender, EventArgs e)
{
i++;
if (i >= dt.Rows.Count)
i = 0;
dr = dt.Rows[i];
textBox1.Text = dr["id"].ToString();
textBox2.Text = dr["name"].ToString();
textBox3.Text = dr["deptname"].ToString();
textBox4.Text = dr["salary"].ToString();
}
```

```
//Display previous tuple.
private void Previous_Click(object sender, EventArgs e)
{
i--;
if (i < 0)
i = dt.Rows.Count - 1;
dr = dt.Rows[i];
textBox1.Text = dr["id"].ToString();
textBox2.Text = dr["name"].ToString();
textBox3.Text = dr["deptname"].ToString();
textBox4.Text = dr["salary"].ToString();
}
//Insert into the table
private void Insert_Click(object sender, EventArgs e)
{
connect1();
int sal = int.Parse(textBox4.Text);
OracleCommand cm = new OracleCommand();
cm.Connection = conn;
cm.CommandText = "insert into instructor values('" + textBox1.Text + "',
'" + textBox2.Text + "','" + textBox3.Text + "','" + textBox4.Text + "')";
cm.CommandType = CommandType.Text;
cm.ExecuteNonQuery();
MessageBox.Show("Inserted!");
conn.Close();
}
//Updates into a table
private void Update_Click(object sender, EventArgs e)
{
connect1();
int v = int.Parse(textBox2.Text);
OracleCommand cm = new OracleCommand();
cm.Connection = conn;
cm.CommandText = "update instructor set salary=:pb where deptname =:pdn";
```

```
cm.CommandType = CommandType.Text;
//Uses OracleParameter to read the parameter from the GUI
OracleParameter pa1 = new OracleParameter();
pa1.ParameterName = "pb";
pa1.DbType = DbType.Int32;
pa1.Value = v;
OracleParameter pa2 = new OracleParameter();
pa2.ParameterName = "pdn";
pa2.DbType = DbType.String;
pa2.Value = textBox1.Text;
cm.Parameters.Add(pa1);
cm.Parameters.Add(pa2);
cm.ExecuteNonQuery();
MessageBox.Show("updated");
conn.Close();
}
//Below is the code snippet to illustrate the use of DataGridView. Have a button and
//datagridview control on the form. On click of the button below code is called
 private void GetGrid_Click(object sender, EventArgs e)
 {
 connect1();
 comm = new OracleCommand();
 comm.CommandText = "select * from instructor";
 comm.CommandType = CommandType.Text;
 ds = new DataSet();
 da = new OracleDataAdapter(comm.CommandText, conn);
 da.Fill(ds, "instructor");
 dt = ds.Tables["instructor"];
 int t = dt.Rows.Count;
 MessageBox.Show(t.ToString());
 dr = dt.Rows[i];
 dataGridView1.DataSource = ds;
 dataGridView1.DataMember = "instructor";
 conn.Close();
 }
//Following is the code snippet to populate a combo box from database.
//Have a combobox on the form and call below code on form load.
private void Form1_Load(object sender, EventArgs e)
```

```
 {
 connect1();
 comm = new OracleCommand();
 comm.CommandText = "select deptname from instructor";
 comm.CommandType = CommandType.Text;
 ds = new DataSet();
 da = new OracleDataAdapter(comm.CommandText, conn);
 da.Fill(ds, "instructor");
 dt = ds.Tables["instructor"];
 int t = dt.Rows.Count;
 MessageBox.Show(t.ToString());
 comboBox1.DataSource = dt.DefaultView;
 comboBox1.DisplayMember = "deptname";
 conn.Close();
}
}
```

Lab exercises:

1. Connect the VC# front end of INSURANCE database with the back end. Execute the queries given under Lab Exercises (Lab 3 and Lab 4) through front end.

Additional Exercises:

1. Connect the VC# front end of ORDER PROCESSING database to its back end. Execute the queries given under additional exercise (Lab 4 and Lab 5) for the same database.

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

LAB NO.: 9 **Date:**

RELATIONAL DATABASE DESIGN

Objectives:

- To analyze the functional dependency and able to design normalized relational schema.

Introduction:

Sample Relational Model for the E-R diagram given in Figure 7.1

Patient(Pat_id, Pat_name, age, sex, Address, DOB, MOB)
Room(Room_id, Room_No, Room_type, Room_cost)
Admission(admsn_id, Pat_id, Room_id, Rcp_id, date, time)
Receptionist(rcp_id, rcp_name, Age, Address, MOB, Shifting)
Here Shifting refers to morning, afternoon or night shifts.
Doctor Table(Doc_id, Doc_name, Age, Address, Salary, MOB, Designation, Passed_from)
Here Passed_from indicates passed from which institution.
Appointment(Ap_id, Pat_id, Doc_id, Rcp_id, Ap_date, Ap_time)
This is a junction table between Patient, Receptionist & Doctor tables.
Bill(Bill_id, Bill_purpose, Bill_total)
Here Bill_purpose refers to the cause e.g blood test for which the bill is paid.
Accountant(Acct_id, Acct_name, Age, Address, MOB, Working_time, Acct_salary)
Payment Table (Pay_id, Bill_id, Pat_id, Acct_id, Pay_type, Pay_date)
This is a junction table between Patient, Bill & Accountant tables.

Note: For further details on the relational model, normalization of the E-R diagram given in Figure 7.1, refer: http://www.enggjournals.com/ijcse/doc/IJCSE10-02-08-050.pdf

Lab exercises:

1. Submission of relational model and functional dependency for the mini project.
2. Submission of tables designed for the mini project in minimum BCNF along with the normalization process.

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

LAB NO.: 10 **Date:**

DATABASE IMPLEMENTATION AND DATA POPULATION

Objectives:

- To implement and populate the database of the mini project.

Lab exercises:

1. Implement the database using the normalized relational schema designed in Lab 9. Populate the database with suitable data.

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

LAB NO.: 11 **Date:**

PROJECT IMPLEMENTATION

Objectives:

- To analyze the working of the front end and back end altogether.

Lab exercises:

1. Give the implementation details regarding your project.

[OBSERVATION SPACE – LAB 11]

[OBSERVATION SPACE – LAB 11]

[OBSERVATION SPACE – LAB 11]

[OBSERVATION SPACE – LAB 11]

[OBSERVATION SPACE – LAB 11]

LAB NO.: 12 **Date:**

TESTING AND VALIDATION

Objective:

- To test and validate their mini project.

Lab Exercises:

1. Give the testing and validation details for your project.

[OBSERVATION SPACE – LAB 12]

[OBSERVATION SPACE – LAB 12]

[OBSERVATION SPACE – LAB 12]

[OBSERVATION SPACE – LAB 12]

REFERENCES

1. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, (6e), McGraw Hill Education (India) Edition, 2013.

2. Ramez Elmasri, Shamkant B. Navathe, “Fundamentals of Database Systems”, Addison-Wesley Publications, 2013.