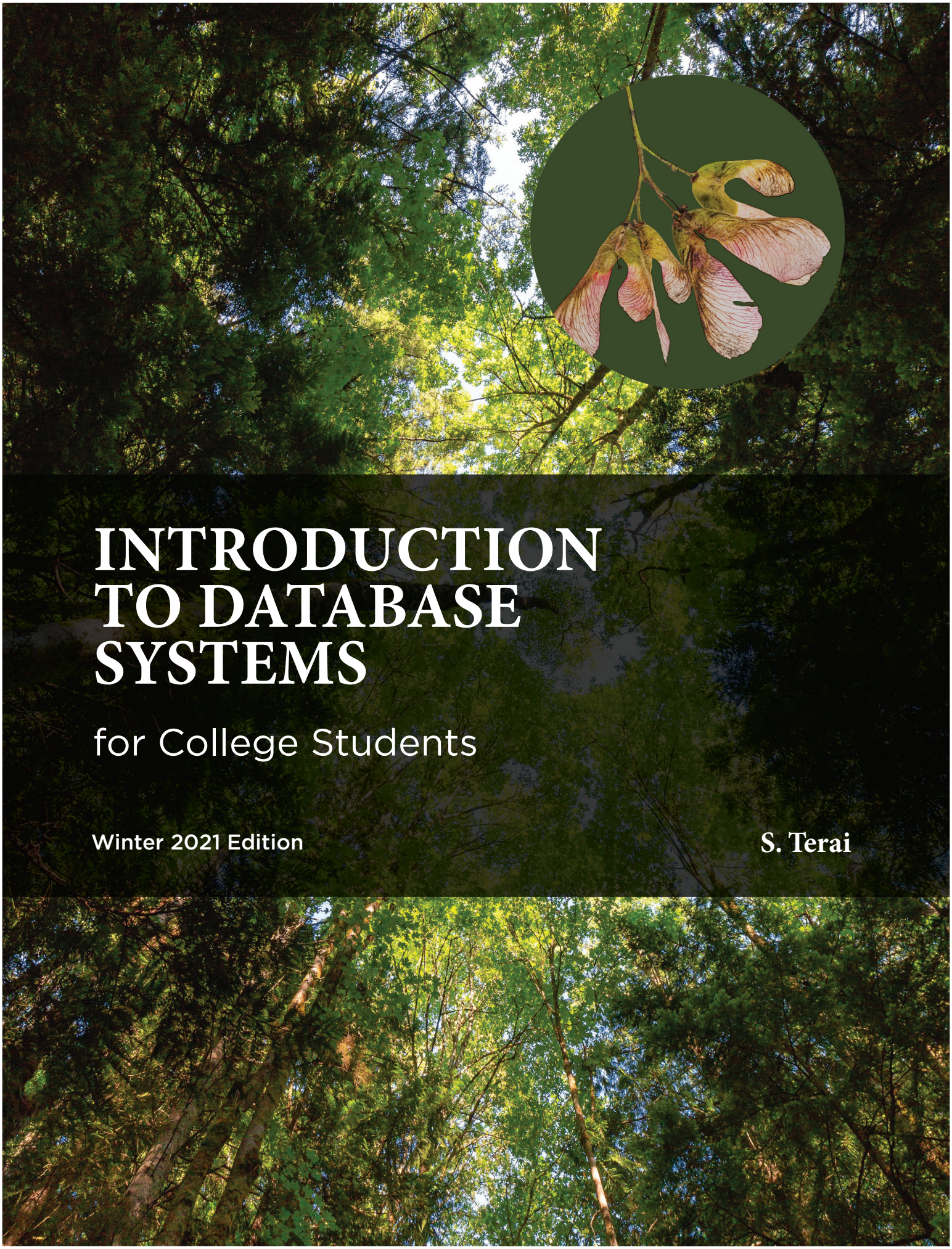


INTRODUCTION TO DATABASE SYSTEMS

for College Students

Winter 2021 Edition

S. Terai

Introduction to Database Systems

Algonquin College of Applied Arts and Technology
Ottawa, Ontario, Canada

Edition 2.1-Stu
Winter 2021

Contents

List of Figures

List of Tables

Preface

The sequence of reading assignments are deliberate. For example, reading the summary of a chapter earlier than later, is suggested. The order of chapters covered in lessons are not sequential, this facilitates completion of labs with practice in SQL early in the course followed by theory on normalization. During lecture we will reference sections or subsections from other lessons. Treat this as a reference manual, workbook and a guide. It is difficult to read the owners manual of a new car from start to finish, at different times we need sections relevant to our needs, some this this analogy applies to this book.

Students are encouraged to write on this workbook, space is provided for answers. Sections written as *Aside:* provide additional thought, analogy or comparison to the idea discussed. The reader may defer reading these on the second pass; continuity is maintained if these sections are not read.

The agony and joy of creating fictitious names, I have been spared with; the pattern for person names is `Consonant, Vowel` ending with a consonant, or other similar pattern. Names are either five or three letters. A spreadsheet generates the names for me; a familiar name is coincidental. This document is typeset in LaTeX.

Table names have a `_T` suffix, `_V` for dynamic view, `_MV` for Materialized View. Primary key constraints end in `_PK`, foreign key in `_FK`. These suffixes make it easy to identify database ojbects for a student new to the topic. Objects with the same name can be used freely, it is the suffix that differentiates them.

Feedback on improving this workbook including errors, I would welcome and gladly accept; please email me at, `terais@algonquincollege.com`.

Acknowledgments

Kumari Gurusamy has identified several corrections in the Spring 2015 edition. Sanaa Issa has used these notes for her labs and lectures, has helped in editing in the Fall 2017 Edition.

Lab 2 was first developed by Patricia Murphy and other instructors. It has been modified to work on databases that have changed over the years, in particular the user interface.

In an earlier writing assignment, Louisa Lambregts had provided valuable suggestions and comments, they are incorporated in this document.

He is eloquent, patient and knowledgeable. I am thankful that Mel Sanschagrin had agreed to review the document. All of his suggestions are indispensable.

Some topic reorganization has been suggested by Sarfraz Khan, it is based on his experience in teaching SQL. More work needs to be done; in later editions of this book I plan to implement more of his ideas.

Over the semesters, many students have identified errors and suggestions. They have been an invaluable source in improving this document.

In 2001 I had the opportunity to attend a seminar on *The ICE Approach* [14]. Many pedagogical ideas have been incorporated from their book in my teaching.

Dedicated to students – past and present.
This document has been possible because of you.
You have been patient – Thank You.

To the Student

Consistent and deliberate practice will give you confidence and a sound understanding of database theory and its query language.

Make full use of your lab hours and use them wisely. Your learning is consolidated by *doing* the lab exercises and repeating them.

Aquaint yourself with at least one of the following study techniques. The link leads you to a short document. Alternatively you may refer to wikipedia or search the web for the techniques.

1. SQ3R - Details can be found at http://www.wpi.edu/Images/CMS/ARC/SQ3R_At_A_Glance.pdf.
   Lately SQ4R is proposed as an extension to SQ3R. The last R in 4R is for `wRite`
2. PQRST - http://www.lethbridgecollege.net/elearningcafe/images/stories/pdf/pqrst.pdf

The lab and lecture time is intended to be collaborative, not competitive. You will learn the topic in a different way by helping your fellow students. Do not hesitate to ask questions and get clarifications.

Updates to 2021 Edition

Changes to the Winter 2021 Edition

1. `album-pg` database has been deprecated; it is replaced with `Practice` database.
2. Referred to The Art of PostgresSQL, D Fontaine, 2e
3. Made references to TCL.

Proposed Changes in Future Edition

1. Replace `world-pg.sql` with open source `GeoNames` geographical database. Develop labs and assignments around this database.
2. Move video files to Google Drive, provide links for video files in document.

1

Database - An Introduction

1.1 Introduction

The core of any information system is a database. It is used by all computing applications, its size, structure and type vary. Formal knowledge of a database, especially relational database, is an important skill in your IT career.

This lesson begins introducing a few terms, differentiating between data and information. Data when processed is information, information when processed is knowledge.

1.2 Objective

- Appreciate the role of databases in commercial, industrial and government organizations
- Describe components of database management system (DBMS) and their interaction
- Identify skill categories required at the workplace to run a database
- Differentiate between *data* and *information*
- Determine storage and use of metadata

1.3 Terms

Entity A person, place, object, event or concept; an entity becomes relevant if an organization has a need to maintain data on the entity. [3, Page 538]

Relation Refer Section 8.3, page 112 of these notes.

Data Element A smallest item in a database. Loosely referred as a *field*.

Attribute The name and description for a data element. *Aside:* Strictly, in a relational model, the term *attribute* is used instead of *field*. This difference is historical from the time the relational model was first proposed in 1969.

All data elements in a *column* have the same characteristics of the data, these characteristics are referred to as attribute. Loosely speaking an attribute corresponds to a column of a relation.

Entity Relationship Model A *logical* representation entities and their relationships.

Entity Relationship Diagram (ERD) A physical representation of a logical model.

Relational Database A database that stores data in relations associated with relationships.

Relationship An association between one, two or three entities. *Aside:* Each relationship has a cardinality and a degree. In this course we discuss binary degrees, and to some extent unary degrees, we do not discuss ternary degrees.

Cardinality The number of ***instances*** one entity associates with another entity. Examples of cardinality are `one:one`, `one:many`, `many:many`. The word *instances* is important. You should be able to differentiate between the two terms, **cardinality** and **degree**.

Degree The number of entities that participate in a relationship. There are three degrees: unary, binary and ternary.

Systems Development Life Cycle (SDLC) The traditional method used to plan, analyze, design, implement and maintain an information system.

1.4 Types of Relationship

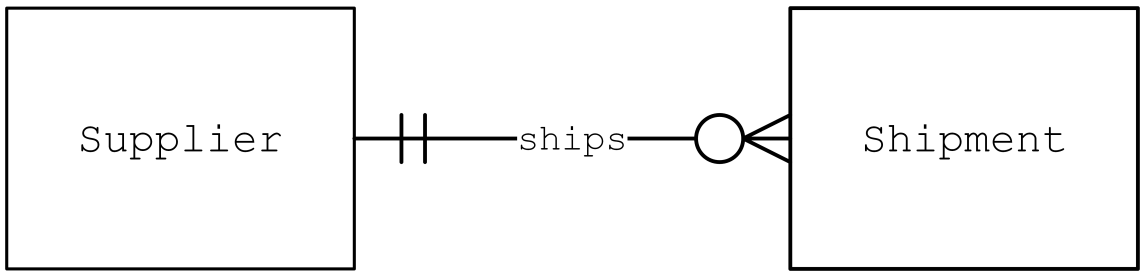


Figure 1.1: Supplier Shipment

The ERD in figure 1.4 shows a supplier can ship zero, one or many shipments. A shipment can be shipped by one and only one supplier. Stated differently, it is not possible for a shipment to arrive without a supplier name identified and a shipment cannot arrive from more than one supplier. The two bars on the Supplier side of the relationship is called mandatory one; i.e. must have one and only one.
The *degree* of the relationship is **binary**, it has two entities, `Supplier` and `Shipment`. Later we shall see two other degrees between entities. Refer figure 2.6 on page 28 for another example.

1.5 Learning Activities

Examples of small databases

List examples in daily life where you would use databases. Think about a hobby, small business, club activities or club membership, game statistics, travel. Choose an example that interests you and build it toward your assignment. Your application should have at least five entities. As we explore ideas in the classroom write them down in the space provided below.

1.5.1 A Table and its Components

Data is stored in a table; a table has rows and columns, similar to a spread sheet or a word processing document. Tables in a database have more meaning compared to the tables in a document. Even a small database has several tables which can interact with each other in storage and data retrieval.

Metadata Data that describes properties and characteristics of data is called metadata. Table 1.1 illustrates a small portion of metadata. Among other items metadata includes *a*) name of data element *b*) type, i.e. character, alphanumeric, date, decimal, boolean *c*) length *d*) minimum and maximum value, if applicable *e*) a short description *f*) the source of data, i.e. its origin. Examples of data sources are: customer, registrars office, accounting department.

Name	Type	Len	Description	Source
BookID	Fixed Character	13	Book ISBN	Assigned by publisher
Book Title	Variable Character	40	Title of the book	Usually given by author
AuthorID	Fixed Character	10	Surrogate Key	Assigned by system or by user
Author	Variable Character	30	Name of Author	Entered by user
BorrowerID	Fixed Character	7	Surrogate Key	Assigned by system or user defined
Borrower Name	Variable Character	30	Borrowers Name	Entered by user

Table 1.1: Metadata, an example

1.6 Data Model

A data model is a description of users data, relationship between data, constraints on data, among other details. This model translates into Data Definition Language (DDL). A data model is an abstraction that focuses on the essential aspects of an information system. It aids in communicating users requirements to stakeholders. Stakeholders include developers, end users, administrators and project sponsors among others. There are several methods used in representing a data model. A common diagramming method is an Entity Relation Diagram (ERD). ERD's have several different notation, in this course we use `Crows Feet` Notation.

Entity Relation Diagram An Entity Relationship Diagram is pictorial representation of data. It consists of entities and relationships between the entities. At least two types of ERD's are used - **Logical** and **Physical** ERD's. They are used in different stages of the system development life cycle (SDLC). A Logical ERD is created at the initial stages of the SDLC. This ERD is independent of the database model that will be used. When creating a logical diagram the system analyst does not consider the database *model* that will be used to implement the system. *Aside:* A database model is not the same as a data model.

As the development progresses, the logical ERD is refined and transformed into a physical ERD. In our study the database model is a Relational Database. An apparent transformation is resolving a many-to-many relation into one-to-many relation. Other details added to physical ERD's for a relational database model is determining prime key, foreign key and constraint. In most cases, there is a relation between two entities. In some cases an entity is related to itself, and in other cases it there are three entities that relate to each other.

Degree of a Relation When two entities are related in a binary relation, the degree is two. An entity related to itself it in a unary relation, i.e. of degree one. Three entities related in a ternary relation has a degree of three. In this course we shall work with entities of degree one and degree two only. We shall not study relationships of degree three.

Associative Entity An entity that is used to resolve a many-to-many `m:m` relationship. An associative entity is represented by rounded corners.
A Logical ERD may have two entities that are related by a `m:m` relationship. This relation cannot be implemented in a Relational Database. By introducing an associative entity an `m:m` relation is resolved into two `1:m` relations. An example of an associated entity in a Physical ERD is shown in figure 10.5 on page 161.

Forward Engineering An database designer could model data using a data modeling tool. The tool then creates DDL from the model. Compare this processes to an architect drawing a structure and the diagramming tools lists the materials needed to build the structure.

Reverse Engineering Often a database schema already exists, created by DDL, a database designer wants a pictorial representation of the code. A data modeling tool can create an ERD from the DDL. In our course we shall first draw a logical ERD using pencil and paper, determine the prime key and foreign key. Next we shall write the DDL using the editor (pgAdmin3) test and run the DDL. Finally we will use the data modeling tool to reverse engineer model from the DDL. In one or two instances we shall forward engineer the model, most of the time it will be reverse engineering. We need to be able to write DDL and DML statements fluently. *Aside:* DML statements are not required to reverse engineer an ERD, only DDL statements are needed.

1.7 ERD Exercises

Draw a logical ERD from the rules given.

I Country-City A country can have many cities. A country may not have any cities. A city must belong to one and only one country.

II A vegetable can have many nutrients. A vegetable must have at least one nutrient. A nutrient can be obtained from more than one vegetable.

III A household may have taxpayers. A household may not have any taxpayers. A taxpayer must belong to a house.

IV A student can have only one UPass. A student may not have a UPass. A UPass must belong to only one student.

1.8 Drawing Physical ERD

First draw a logical ERD and then a physical ERD for the given abstract and business rules.

I. Registration. Student-Course

Rules A Student can enroll in one or more courses. If a student is registered, she is considered a student, even though she may not have enrolled in any courses. A course may have more than one student enrolled. A course that has been offered may not have any students enrolled in it.

??

II. A Trading Company

Abstract The trading company sells computer parts. The company maintains a list of customers with some details such as address and phone numbers. A list of products in the inventory is maintained. Customer purchases are initiated by creating an invoice.

Rules A customer can have at least one invoice. A customer may not have any invoice. An invoice must have one and only one customer. Each product that is purchased is shown in an invoice. An invoice can have one or more products. A product can be purchased several times, i.e. a product can appear in one or more invoices. It is possible that a product may not be sold, i.e. it will not appear in a invoice at all. An invoice can have one or more products of the same type.

1.9 Client-Server Architecture of a DBMS

Use the space below to draw a schematic diagram of a client-server architecture.

1.10 Review Questions

1.10.1 Written Answers

1. Define the term *data*.

2. Explain what is meant by the term *database*.

3. Explain what is meant by a DBMS.

4. Name some database products and their vendors.

5. What is meant by *information*? Differentiate *information* from *data*.

6. Explain what is meant by *metadata*. Give two examples of data and metadata. What does metadata include? What does metadata not include?

1.10.2 Multiple Choice - Select one correct answer

1. Software used to create, maintain, and provide controlled access to databases is called
   (a) Computer Aided Software Engineering (CASE) Tools
   (b) Graphical User Interface (GUI)
   (c) Database Management System (DBMS)
   (d) Network Operating System (NOS)
   (e) Computer Assisted Design (CAD)
2. Which one of the listed task, or feature, is not the main purpose of a database management system (DBMS).
   (a) store data
   (b) create data
   (c) update data
   (d) provide an integrated development environment
3. A relationship is an association between entity types
   (a) True
   (b) False
4. The _____ of a relationship is the number of entity types that participate in the relationship
   (a) attribute
   (b) cardinality
   (c) degree
   (d) constraint
5. A rule that specifies the *number of instances* of one or more entities
   (a) attribute
   (b) cardinality
   (c) degree
   (d) constraint
   (e) relationship

1.10.3 Short answer questions

1. Processed data is called _____

2. Data in a table is stored in the form of _____ and _____

3. Description of properties of data is called _____

1.11 Summary

The term DBMS indicates it is a system, i.e. a DBMS consists of several components, not all components may be available in a package. Web development has added an additional dimension to databases.

1.12 Lab 1 - Install PostgreSQL & Data Modeler

Objective

1. Install PostgreSQL on your computer.
2. Install and populate `world` database and `Practice` database.
3. Create a group role and login role.
4. Install a Data Modeling Software.

Reference

1. `postgreSQL-10-US.pdf` and `postgreSQL-9.5-US.pdf`. *Aside:* These are your first references `postgreSQL-9.5-US.pdf` has been included for continuity as page references in this document. Use `postgresql-10-US.pdf` first.
2. http://www.postgresqltutorial.com/

Submission Show your lab instructor an installed copy on your laptop, of PostgreSQL, the two sample databases, the group role and login role. Take a screenshot (Window Key + Print Screen) of `pgAdmin` showing the two databases, group role and login role. Upload this file to LMS.

Procedure This section tells you about the procedure to install the software needed for the course. You will benefit by reading the entire lab first, even though you will be reading many new terms that are not familiar. Reading the entire lab first will give you an overview and prepare you for the steps that you need to follow. Read section titles *Best Practices* on page 171, before you begin this lab.

Install PostgreSQL

Postgres Download Visit `https://www.postgresql.org` Click on the download button. Scroll down for `Windows`. Click on `Windows`. You will be at `Windows Installers` Screen. Read the section `Interactive Installer by EnterpriseDB`. *Note:* `Stackbuilder` is not required for this course. Scroll down further, read the section `Graphical Installer by BigSQL`. In this course we are mainly working with the `graphical installer`. Click on `Download Graphical Installer` link. You are now at `installers.jsp` page. Chose `postgresql-9.5.21-3-win64.exe`. Do not chose version 10. The download should begin. The installation file is 67.5 MB, usually it is stored in the Windows `Downloads` folder. *Tip:* Keep you `Download` folder tidy. Create folders by year (or month) and transfer files to them.

Postgres Installation Open the `Downloads` folder and locate the file. Be patient, follow these next few instructions carefully. Right click on the file, in the menu option click on `Run as administrator`. Windows will ask your permission to install the file. Click `Yes`. Keep the default directory as `C:\PostgreSQL`. If you have an SSD drive and a conventional HDD you may want to install it on `D:\`, click `Next`. On the `Select Components` screen check `pgAdmin3 LTS`, this is the client software. *Note:* `PostgreSQL Database Server is selected for you.`

Password. During installation it is strongly recommended that for the first three labs you keep the password as `algonquin`, all lower case letters. You have many other passwords to remember. Keep this lab simple, you may change the password later.

Options Do not select `Advanced PostgreSQL Configuration Options`. Note the `PostgreSQL Port` it should likely be `5432`. Make a note of the data directory `C:\PostgreSQL\data\pg95`

Connecting the Server PostgreSQL 9.5 The tutorial tells you to connect (the client) to the server by double clicking on the PostgreSQL 9.5 link. You may also connect by right clicking and then selecting `Connect` from the pull down menu.

Create Group Role In pgAdmin3 expand the `Servers` leaf by clicking on the plus (+) sign. Right click on `Group Roles`, select `New Group Role` from the pop up menu. In the pop up dialog box type in `web`, as new group role.

Create a Login Role Creating a Login Role is similar to creating a Group Role. You need to associate the Login to a Group. Right Click on the `Login Roles` leaf. Chose `New Login Role` from the menu. In `Role Name` type in `your Algonquin LoginID`. Click on the forth tab is titled `Role Membership`. You will see two panels, move the Group Role `web` from the left `Not Member` panel to `Member` panel. Select `web` and then click on the right chevron to change the position.

Exercise Files. Download the exercise file titled `world-pg.sql` from OneDrive. *Aside:* The `world` database is adapted from `MySQL`, it is not perfectly designed, but it will serve us for the first few labs. It gives us an opportunity to critique it and suggest improvements.

Creating a Database Expand the `Servers` and `localhost (localhost:5432)` leaves. You should see a `Databases` leaf. Right Click on this leaf and select `New Database...`. In the dialog box, enter `world` as Name. For Owner select `web` from the pull down menu. It takes a few seconds to create a database. You should see the number incremented in the `Database` leaf. `world` database appears with (possibly) a small red cross. This symbol implies that the database is not connected/active. At this stage `postgres` has created an empty database, it does not have any data.

Populating the `world` database After creating a database the next step is to populate it with data. Ensure you have `world-pg.sql` on your computer. Click on the `world` leaf. The red cross should go away. Familiarize yourself with the Toolbar. The sixth icon from the left says `SQL`, it has an image of a magnifying glass on a cylinder. If you hover over the icon it will say `Execute arbitrary SQL queries.` Click on this icon. You now have a fresh canvas. This is the client part of the software, `pgAdmin`. Observe the top left hand corner of this screen. It will say `Query - world on postgres@localhost:5432`. Click on `Open File` icon. Find the file `world-pg.sql` on your computer and open it. `world-pg.sql` is a file with about 5370 lines of `SQL` code. It includes database objects and data. In the pgAdmin3 toolbar, locate the green play button. Hover your mouse on it, it should say `Execute query`. Click on this icon. Running the file for the first time will display `NOTICE:` later we shall interpret them.

The first icon on the toolbar of `pgAdmin` will show `New File` Click on this icon. You shall see another fresh canvas. On the top section type in `SELECT * FROM City;`, and click the green play button.. You should see about `4079` rows, the major cities in about `239` countries.

Installing `pgmodeler`

Note: `pgmodeler` is not requried until a later lab. You may postpone this installation. Download `pgmodeler-0.9.1-windows64` from LMS. Run the installation file. Windows Defender will give you a warning, select `More info` confirm it is the right software and click on `Run Anyway`. The the setup dialog, accept the license agreement. Chose the correct path, click `Next`. Let the installation create a folder, if it prompts you. Let it create a Smart Menu folder and a Shortcut (if you like). Finally click on `Install`. The software is small and will be quick to install. Before you click the `Finish` button, ensure that `.dbm` files are associated to `pgModeler`.

Running `pgmodeler`

Start pg Modeler. Remember to say hello to the elephant. From the left hand panel, click on `Manage`. If postgres has been installed on your computer, it will detect the connection. Chose the `localhost:5432` connection.

PostgreSQL Interface

PostgreSQL uses client-server architecture. Both the client and server are installed on your laptop. To access the server there are two client programs - `psql` and `pgAdmin III`. `pgAdmin III` has a graphical interface, we shall use this now and later use `psql`.

pgAdminIII & pgAdminIV

pgAdminIII (or pgAdmin3) and pgAdmin4 are both graphical user interfaces (GUI) for postgres server. `pgAdmin3` is an *application* on Win 10, pgAdmin4 is a web based GUI. Although it is matter of preference on usage, for our learning purpose an application based GUI, i.e. `pgAdmin3`, is easier to use. From a developers view a web based client, needs to be written once for a browser to function on all three operating systems, Windows, MacOS and Linux. An *application* has to be developed for each of the three operating systems. An application's interface is inherently elegant to use compared to a web based client.

2

SQL - SELECT Statement

2.1 Introduction

Our focus in this chapter is to get familiar with SQL and achieve a working knowledge of basic SQL commands and some intermediate commands. Several important SQL clauses are explained in this chapter, they will form the foundation of your SQL knowledge.

2.2 Objective

- Define and differentiate Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL) and Transaction Control Language (TCL).
- Identify milestones in the history of `SQL`, the official standard for relational databases.
- Establish relationships between the tables and write queries using tables.
- Implement constraints that maintain database integrity.
- Identify SQL 2008 standards and new commands.

2.3 The SELECT Statement

2.3.1 Processing Order

Although the `SELECT` keyword appears first in the statement it is the parsed much later. Figure 2.1 [1] illustrates the order of processing. The `FROM` keyword is evaluated first, this is the reason aliases in `WHERE` and `HAVING` clauses are not permitted. `GROUP BY` is evaluated first followed by `HAVING`, `ORDER BY` is evaluated first then `LIMIT`.

[1] Diagram adapted from Ref: [2] page 277

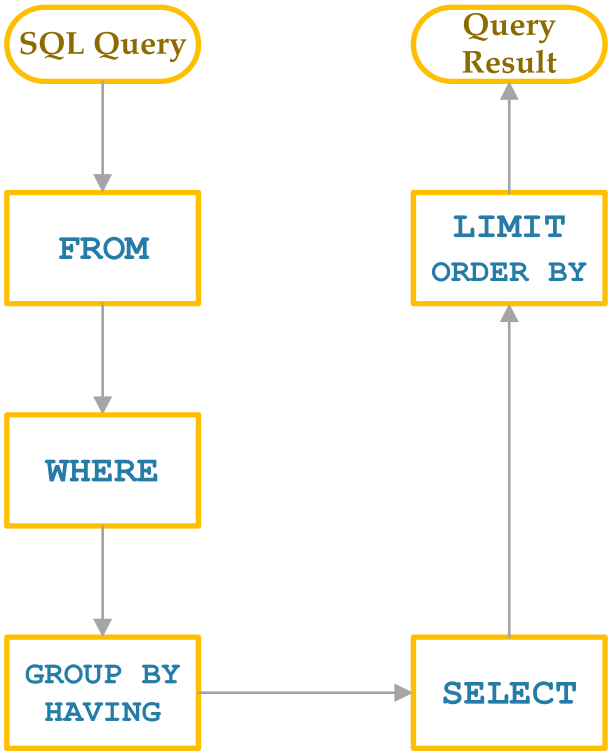


Figure 2.1: SQL Processing Order

2.4 DDL, DML, DCL & TCL

> Any given data sublanguage is a combination of atleast two subordinate languages: a *data definition language* (DDL) ... and a *data manipulation language* (DML)...
> Ref. [10, CJ Date, Page 36]

DDL supports the definition or declaration of database objects, DML supports processing of the objects. Ref. [10, CJ Date, Page 36]. Examples of DDL include `CREATE`, `DROP`, `CONSTRAINT`, `PRIMARY KEY`. Examples of DML include `INSERT`, `UPDATE`, `SELECT`.

Data Control Language (DCL) is a set of commands that are used to allow or deny access to a database system. Examples of DCL include `CREATE ROLE`, `DROP ROLE`, `GRANT` and `REVOKE`. Privileges may be granted or revoked to/from a user or a *role*.

Definition: A *role* is a set of permissions assigned by an administrator to users or groups.

Transaction Control Language (TCL) . SQL statements need to be grouped together and run as a single unit. More details and examples are given in a later lesson.

Multiple Choice Questions.

Select the best answer. Each question has one correct answer.

1. Commands used to define a database, which includes creating, modifying and deleting tables, and establishing constraints.
   (a) Data Definition Language (DDL)
   (b) Data Manipulation Language (DML)
   (c) Data Control Language (DCL)
2. Commands used to maintain and query a database, which includes data insertion, deletion and modifications to data.
   (a) Data Definition Language (DDL)
   (b) Data Manipulation Language (DML)
   (c) Data Control Language (DCL)
3. Commands used to control a database, which includes access control.
   (a) Data Definition Language (DDL)
   (b) Data Manipulation Language (DML)
   (c) Data Control Language (DCL)
4. `CREATE TABLE <table name>`, `CONSTRAINT <label> PRIMARY KEY <attribute>` are examples of
   (a) DDL
   (b) DML
   (c) DCL
5. `INSERT`, `UPDATE`, `SELECT` are examples of
   (a) DDL
   (b) DML
   (c) DCL
6. `GRANT`, `ADD`, `REVOKE` are examples of
   (a) DDL
   (b) DML
   (c) DCL
7. Which term best describes the definition:
   A set of permissions assigned by an administrator to users or groups
   (a) root
   (b) role
   (c) user
   (d) account holder
   (e) superuser

2.5 Access Control

Consider the two scenarios:

Scenario 1: You are mandated to have only one (physical) key, this key works for the front door to your house, garage, car, mailbox, office door, safe deposit box in the bank and two other assets you own.

Question: How would you adapt? Think of some scenarios where this system would be difficult in practice.

Scenario 2: A manufacturing organization has about 8,000 employees and several departments, sales, marketing, design, inventory, service, payroll, executive management, finance, public relations. These departments have their systems on a computer using postgres. All these departments have one account called `postgres` with the same password. Clearly, this system be difficult to manage even if all employees were honest, trustworthy and diligent.

At installation, PostgreSQL provides a login role called `postgres`. *Aside:* It also creates a database called `postgres`.

2.6 Case Sensitivity in PostgreSQL

Keeping with the tradition of Ingres, Postgres is case sensitive to object names; many databases are not case sensitive. Two examples shown below illustrate the differences. Forward engineering a schema will put quotes before and after an object name (tables, constraints...). The label you provide to aa modeling tool such as `Toad` will affect your code.
Aside: There is a way to configure the modeler to generate code otherwise, this section considers the default setting.

Example 1. Using quotes to define objects Consider the following `SQL` code fragment.

```
CREATE TABLE "Customer_T" (
Cust_Id CHAR( 4 ),
Cust_Fname VARCHAR( 30 ) NULL, ...  ...
```

Figure 2.2: Using quotations to define an object

The table name `Customer_T` is within quotes, Postgres will preserve the object name you have provided.

The following `SELECT` statement will run on the `DML` for the code shown in figure 2.2

```
SELECT * FROM "Customer_T";
```

This statement will not run

```
SELECT * FROM Customer_T;
```

The error message shown below gives the insight to the inner working of postgreSQL.

```
ERROR:  relation "customer_t" does not exist
LINE 1: SELECT * FROM Customer_T;
^
ERROR: relation "customer_t" does not exist
SQL state: 42P01
```

Figure 2.3: SQL Case Sensitive Error

The error tells us the Postgres has converted the object name `Customer_T` to lower case, because there were no

quotes around it. The code in figure 2.2 will need DML statements shown below; the object name must be in quotes.

```
INSERT INTO "Customer_T" VALUES ('C001', ...  ...
```

Example 2. Not using quotes to define objects Consider the following SQL code fragment.

```
CREATE TABLE Customer_T (
Cust_Id CHAR( 4 ),
Cust_Fname VARCHAR( 30 ) NULL, ...  ...
```

Figure 2.4: Defining an object without quotes

Compare figure 2.4 and figure 2.2. What is the difference?

For the DDL statements in figure 2.4 the following DML statements will run.

```
SELECT * FROM Customer_T;
SELECT * FROM customer_t;
SELECT * FROM CUSTOMER_T;
```

Aside: In a later assignment you will query the INFORMATION_SCHEMA; it provides information about tables, views, columns and procedures. Querying the INFORMATION_SCHEMA requires that tables names be in lower case, the way Postgres stores them internally. Refer page 181 figure B.4 to see how the table names are in typed lowercase.

2.7 PostgreSQL Interface

PostgreSQL uses client-server architecture. Both the client and server are installed on your laptop. To access the server there are two client programs - psql and pgAdmin III. pgAdmin III has a graphical interface, we shall use this now and later use psql.

2.8 Scalar and Vector Aggregates

When a single value is returned from an SQL query it is called a *scalar*. For example, MIN, MAX, SUM are some functions that will return a single value. A GROUP BY clause will return several values, collectively these values are called *vector*.

2.8.1 SQL Aggregate Functions

Table 2.1 lists aggregate functions in SQL. An example illustrates usage in the `world` database.

Function	Description
`COUNT( * )`	Counts the number of rows in a table.
`COUNT( attribute )`	Counts the number of rows in a table that have values in the attribute specified. It does not match `NULL` values.
`SUM`	Adds all numeric values of an attribute in all rows of the table.
`AVG`	Calculates the average of numeric values of an attribute of all rows in the table.
`MIN`	Finds the smallest value of an attribute from all rows.
`MAX`	Finds the largest value of an attribute from all rows in a table.

Table 2.1: SQL Aggregate Functions

2.9 SQL String Functions

String functions manipulate and examine character data types - `CHAR`, `VARCHAR` and `TEXT`. A few commonly used string functions are shown in this section.

Concatenation

Concatenation is a string operation that joins two strings. Each language has its own string concatenation techniques. In SQL there are two ways you can join two or more character strings together. The first method is to use the function `CONCAT`. Try the following statements (one at a time) and observe the results.

```
SELECT CONCAT('Algonquin', 'College');
SELECT CONCAT('University', ' ', 'of', ' ', ' 'Ottawa');
```

The second method is to use the `||` operator. Type in the following statements,

```
SELECT ('Canada' || 'Day' );
SELECT ('Canada' || ' ' || 'Day');
```

Miscellaneous String Functions

Table 2.2 lists several comparison operators. An example illustrates usage in the `world` database.

Operator	Description

`UPPER( )`	Convert a string to uppercase. The SQL statement below will display country names in uppercase.
	`SELECT UPPER( Name )`
	`FROM Country;`
	`SELECT Name FROM City WHERE CountryCode = UPPER( 'mex ');`

`LOWER( )`	Convert string to lower case.
	`SELECT LOWER( Name ) AS "City", District`
	`FROM City`
	`WHERE CountryCode = 'RUS'`
	`AND District = 'Tomsk';`

`SUBSTRING( string`
`< FROM int >`
`< FOR int > )`
	Extract a substring.
	To display the `CountryCode` and first four characters of cities in Ecuador
	`SELECT CountryCode, SUBSTRING( Name FROM 1 FOR 4 )`
	`FROM City`
	`WHERE CountryCode = 'ECU';`

Table 2.2: SQL String Manipulation Functions

2.10 SQL Comparison Operators

Table 2.3 lists several comparison operators. An example illustrates usage in the `world` database.

Operator	Description
Operator	**Description**
=	equal to.
⟨ ⟩ or !=	not equal to.
⟨	Less than.
⟩	Greater than.
⟨=	Less than or equal to.
⟩=	Greater than or equal.
IN	Matches a set of values.
NOT IN	Excludes values in a given set.
BETWEEN	Matches values between two given boundary values, it includes end values.
NOT BETWEEN	All values except those in the given range.
LIKE	Matches a set of characters in a pattern. Refer section 2.11 for an explanation.
NOT LIKE	Does not match a pattern of characters.
IS NULL	Use when filtering NULL values. Note: = sign is not functional when filtering NULL values.
IS NOT NULL	Result set shows rows that have data values in field, i.e. excludes NULL values.

Table 2.3: SQL Comparison Operators

2.11 Difference between LIKE and =

SQL Tutorial provides this definition of the `LIKE` operator

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

`LIKE` is a commonly used pattern matching operator
Aside: `LIKE` allows matching text with simple patterns that can include two wild cards `%` and `_`

- `%` matches zero or more characters
- `_` (underscore) matches a single character

The three queries listed below differentiates between `LIKE` and `=`

```
-- this code will work
SELECT * FROM city
WHERE CountryCode LIKE 'CHN%';


-- this code will not work
SELECT * FROM city
WHERE CountryCode = 'CHN%';


-- this code will work
SELECT * FROM city
WHERE CountryCode = 'CHN';
```

Aside: `-- this query does not return any rows, % is replaced by *`

```
SELECT * FROM city
WHERE countrycode LIKE 'CHN*';
```

2.12 SQL Logical Operators

Operators run in the order: `NOT`, `AND`, `OR`. Expressions in brackets overrides all other orders of execution, i.e. expressions in brackets are executed first. Table 2.4 lists several comparison operators. operator

Operator	Description
`AND`	List rows that match both conditions.
`OR`	List rows that match any one condition.
`NOT`	Negates an operand.

Table 2.4: SQL Logical Operators

2.13 Alias

An alias serves several needs in an `SQL` statement, it improves readability by assigning meaningful names in a query, especially when aggregate functions are used. For example, in the following query the `City` table is aliased as `Ci` and `Country` table as `Co`. The last clause is abbreviated as `Ci.CountryCode = Cn.Code` - a compacted form.

```
SELECT Ci.Name AS "City", District, Cn.Name AS "Country"
FROM City Ci, Country Cn
WHERE CountryCode = 'RUS'
AND District = 'Tomsk'
```

```
AND Ci.CountryCode = Cn.Code;
```

2.14 Data Types

Data Type is classification of data. Some considerations when choosing a datatype are: the *kind* of data, how the user/programmer intends to *use* the data, the *operations* that are needed to perform on the data, the *range* - i.e. maximum or minimum values that the data value could take. Postgres supports a variety of datatypes. This section briefly describes the common datatypes. The sample `DDL` statements are used to illustrate the explanation that follows.

```
-- Demonstrate and document DataTypes.
-- DROP TABLE IF EXISTS Student_T;

CREATE TABLE Student_T(
   ID          CHAR( 9 ),
   FirstName   VARCHAR( 20 ),
   LastName    CHARACTER VARYING( 20 ),
   DOB         DATE,
   CareerGoal  TEXT,
   Balance     DECIMAL( 10, 2 )
);
-- eof: DataType.sql
```

Figure 2.5: Datatype, an Illustration

Character Data Types

This category has three main types, `CHAR`, `VARCHAR` and `TEXT`.

`CHARACTER` and `CHARACTER VARYING` These are two basic character data types; shortened as `CHAR` and `VARCHAR`. The number within brackets indicate the maximum length. For example:

`CHAR( 9 )` will store a maximum of 9 characters. The storage space on the storage media required for `CHAR`, in this case will be 9 characters. A data item of less than 9 characters will still take up 9 characters of storage space. `CHAR` can store a maximum of 2000 characters.

`VARCHAR( 20 )` In both data types, `CHAR( n )` and `VARCHAR( n )`, storing more than the number of maximum characters will result in an error; unless the data is all spaces, it will truncate the number of spaces stored to *n*. Specifying `CHAR` without a length will default to one character. Not specifying the number of characters in `VARCHAR`, a very large string, upto 1GB, can be stored.

Aside: By explicitly casting `CHAR` or `VARCHAR` to `n` characters no error is raised. This topic is not covered in this course. Refer to the postgreSQL reference manual for details.

`TEXT` is not an SQL standard, postgres and other DBMS support it.

Numeric Data Types

This data type allows storage of numerical values - integer and floating point. Integer values can take 2, 4 or 8 bytes to store, floating point values can take 4 or 8 bytes.

`INTEGER` takes 4 bytes of storage space, the range of values are from `-2,147,483,648` to `+2,147,483,647`. *Aside:* The total number of unique values are `2,147,483,648 + 2,147,483,647 = 4,294,967,295`, including the zero value. This value is equal to $2^{32} - 1$, 32 bits correspond to 4 bytes.

`BIGINT` takes 8 bytes of storage space, it stores values from -2^{63} to $+2^{63} - 1$.

`DECIMAL` The storage required to store a decimal number is dependent on the definition. For example, `DECIMAL( 5, 2 )`, will store a dollar values from `$999.99` to `$-999.99`. Entering a value `99.999` will round it to `100.00`, similarly `-99.999` will round it to `-100.0`. To store the value of your home you may need `DECIMAL( 6, 0 )`, or `DECIMAL( 8, 0 )` after the stock options from Silicon Valley have materialized.

Storing Date, Date & Time

`DATE` and `DATETIME` The two most common data type for storing date are `DATE` and `TIMESTAMP`. `DATE` will take 4 bytes of storage, it can record dates from 4713 BC until 5874897 AD, for all business applications `DATE` is sufficient, it does not record time. `TIMESTAMP` takes 8 bytes of storage, it stores date *and* time. These two data types are not sufficient for storing very small time values, such as in physics or large durations for astronomical calculations. Postgres does have other datatypes that could possible handle these values.

Storing Binary and Character Objects

`BLOB` is an acronym for Binary Large Object; this data type used to store binary data for example - image, audio, video or executable files. Its use is not standard across DBMS's. `BLOB`'s may be used to store a few critical data items, for large applications such as social media sites using a BLOB data types will have data management concerns such as backup and recovery. In such cases other methods are used.
`CLOB` or Character Large Object is another data type similar to `BLOB`, but stores character data. Its maximum size is limited to 4GB, per record (compare to 2000 bytes for `CHAR`); this makes a SELECT clause slow and impractical. An alternative arrangement could be to store a reference to the data item in the `CLOB` field.

2.15 Review Questions

The SQL Environment

Answer True or False

For the relationship represented in figure 2.6, answer the following three questions

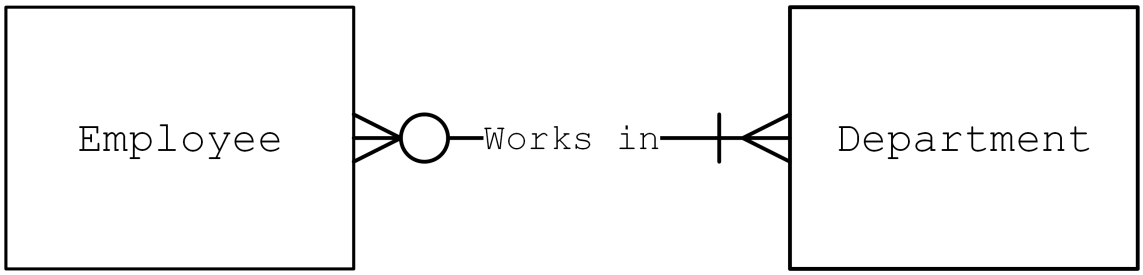


Figure 2.6: Employee-Department Relationship

1. A department can have more than one employee
   (a) True
   (b) False
2. It is possible that an employee does not belong to any department
   (a) True
   (b) False
3. It is possible that a department may not have any employees
   (a) True
   (b) False

Multiple Choice Questions.

Select the best answer. Each question has one correct answer.

1. The first in a series of steps to follow when creating a table is to:
   (a) identify columns that must be null
   (b) create an index
   (c) identify columns that must be unique
   (d) identify each attribute and its characteristics
2. Identify the qualifier that will not display duplicate rows in a SQL query.
   (a) `SPECIFIC`
   (b) `NO DUPLICATE`
   (c) `ALTER`
   (d) `DISTINCT`
   (e) `UNIQUE`

3. Data combined from several observations is called

   (a) summary data
   (b) aggregate
   (c) score
   (d) standard deviation
   (e) average

4. A single value returned from an SQL query that includes an aggregate function

   (a) schema
   (b) dynamic view
   (c) base table
   (d) scalar
   (e) vector

5. Multiple values returned from an SQL query that includes an aggregate function

   (a) schema
   (b) dynamic view
   (c) base table
   (d) scalar
   (e) vector

6. Which SQL clause will sort the rows in ascending or descending order?

   (a) `GROUP BY`
   (b) `HAVING`
   (c) `ORDER BY`

Operator Precedence

7. Identify the correct processing order of boolean operators

   (a) `NOT`, `OR`, `AND`
   (b) `NOT`, `AND`, `OR`
   (c) `AND`, `OR`, `NOT`
   (d) `AND`, `NOT`, `OR`
   (e) `OR`, `AND`, `NOT`

8. What is the result of the SQL statement?

```
SELECT Part
FROM Inventory
WHERE Part = 'Bolt' OR Part = 'Nut' AND Material = 'Brass' OR Material = 'Steel'
```

   (a) Nuts or Bolts made from Brass or Steel
   (b) Nuts made from Brass, all bolts and all steel parts
   (c) Bolts made from Brass and Nuts made from Steel
   (d) Either nuts or bolts, made from either brass or steel
   (e) Bolts made from Steel or Nuts made from Brass

Written Questions

1. Phone numbers are represented by numerals but are defined as characters in a database. Give reasons to support this statement.

2. Write three advantages of using a standard language such as SQL.

3. Briefly explain Data Definition Language. Give examples of commands that qualify as Data Definition Language.

2.16 Summary

SQL, though not a perfect query language has its benefits. It is a declarative language with procedural elements; you do not have to declare variables before using them. Its widespread use in industry makes it an important skill to have.

General Syntax

A general syntax for the `SELECT` statement is:

```
SELECT [ ALL | DISTINCT ] { column / expression }
FROM { table }
[ WHERE conditional expression ]
[ GROUP BY group_by_column_list ]
[ HAVING conditional expression ]
[ ORDER BY order_by_column_list ]
```

Figure 2.7: SQL SELECT Statement General Syntax. Ref: [2, Page 248.]

General syntax of a basic SQL statement and the processing order is shown in figure 2.7. Note the three types of brackets that are used.

{ } - the braces indicate repeating values. One or more items of that type can be included in the command. In the example below `column` is within braces, it means one or more columns can be specified in the `SELECT` statement.

`[ ]` - square brackets indicate optional values.

< > angular brackets indicate mandatory values. Items contained within these brackets must be specified for the command to work correctly.

The vertical bar `|` implies that one item from the list is required. For example, in `ALL | DISTINCT` you need to specify only one of the two items, not both. *Note:* The `SELECT` clause accepts column name and expression.

2.17 Review Questions

1. In an SQL statement, identify the clause that will specify conditions for row selection:
   (a) `FROM`
   (b) `SELECT`
   (c) `WHERE`
   (d) `GROUP BY`
   (e) `ORDER BY`

2. In an SQL statement, identify the clause that will select tables:
   (a) `FROM`
   (b) `ORDER BY`
   (c) `WHERE`
3. Which one of the two SQL statements will filter countries with the letter `C` occurring in the middle of the three letter `CountryCode`
   (a)
   ```
   SELECT CountryCode, Population
   FROM City
   WHERE CountryCode LIKE '%C%';
   ```
   (b)
   ```
   SELECT CountryCode, Population
   FROM City
   WHERE CountryCode LIKE '_C_';
   ```

2.18 Exercise 1 - `Practice` Table

Objective

1. Work on a small table titled `Practice`, to reinforce basic SQL syntax
2. Match expected output with results from `SQL` queries on Order of Precedence

Reference

1. References in 2.21 on page 39 of these notes.

Submission

Complete a short online quiz on SQL, make sure you understand all queries in this lab.

Background

You are working on a small, simple table with 12 rows and two columns. Study the queries provided and then write a similar query on your own. The purpose is to review basic `SQL` syntax. The twelve rows and two columns in the `Practice` table are shown in table 2.5. The table has two columns `Part` and `Material`. It stores four different parts `BOLT`, `NUT`, `SCREW` and `WASHER` in three different materials `BRASS`, `STEEL` and `ALUMINIUM`. Compare your expected answer with each query. Type and run each query first and then write your own query.

Procedure

First create a database in PostgreSQL then run the script `Practice-DDL-DML.sql`. This will create the practice table and add data to the table, i.e. populate the table.

A note on case sensitivity Data is inserted in upperase, you need to use uppercase to match `Part` and `Material`, and use single quotes, for example type `SCREW` instead of `Screw`, type `ALUMINIUM` not `aluminium`.

List all `BOLTS` Type in the following query.

```
SELECT * FROM Practice WHERE Part = 'BOLT';
```

How many rows were shown in the results? Write your answer.

Now modify the above query to list all `SCREW`s Type in your query, test it and then write it down in the space below after you know it runs correctly.

Query 1. List all `SCREW`s from the table.

Part	Material
BOLT	BRASS
BOLT	STEEL
BOLT	ALUMINIUM
NUT	BRASS
NUT	STEEL
NUT	ALUMINIUM
SCREW	BRASS
SCREW	STEEL
SCREW	ALUMINIUM
WASHER	BRASS
WASHER	STEEL
WASHER	ALUMINIUM

Table 2.5: Practice

List all parts made from `STEEL` Type in the following query.

```
SELECT * FROM Practice WHERE Material = 'STEEL';
```

Notice the use of `WHERE` keyword, it *filters* rows from the table. You should now see parts made from `STEEL` How many rows were shown in the results?

Modify the above query to list all parts made from `BRASS` Type in your query, test it and then write it down in the space below after you know it runs correctly.

Query 2. List all parts made from `BRASS`.

List all `BOLT`s, all `NUT`s. Using the logical `OR` operator The following query lists all `BOLT`s and `NUT`s. Type and run the query.

```
SELECT * FROM Practice WHERE Part = 'BOLT' OR Part = 'NUT';
```

Modify the above query to list all `SCREW`s and `WASHER`s Type in your query, test it and then write it down in the space below after you know it runs correctly.

Query 3. List all SCREWs and WASHERs.

Using the COUNT function. The following query will count the number of parts made from ALUMINIUM.
`SELECT COUNT( * ) FROM Practice WHERE Material = 'ALUMINIUM';` How many parts are made from ALUMINIUM? Now modify the query to count the number parts made from STEEL. Write your query here after you have written and tested it.

Query 4. Write a query to count all parts made from STEEL.

List all parts not made from ALUMINIUM The following query will list all parts not made from ALUMINIUM.

```
SELECT Part, Material
FROM Practice
WHERE Material != ( 'ALUMINIUM');
```

Notice this query has the asterisk * has been replaced by the names of the columns, Part and Material after the SELECT keyword. Use this query as the basis to answer the next question.

Query 5. Write a query to list all parts not made from BRASS.

Pause, and answer the quiz. You will need the five answers you have written down.

2.19 Exercise 2 - Queries on Part_T Table

Objective

1. Work on a table titled Part_T to reinforce basic SQL syntax
2. Match expected output with results from SQL
3. Use column names
4. Use simple aggregate functions
5. Use string functions

Reference

1. Section 2.2 Concepts postgresql-9.5-US.pdf
2. Section 2.4 Populating a table with rows postgresql-9.5-US.pdf
3. Section 9.4 String Functions and Operators in postgreSQL-9.5-US.pdf.

Submission Complete the online quiz for this exercise.

Background This table is called Part_T, it has four columns they are Part, Material, Size and Cost. The DDL and DML scripts are provided. *Note:* There is a difference between Part and Part_T. Part is a column name, Part_T is the name of the table. Type in each query, do not cut and paste the queries from the .pdf file, characters in this .pdf document are not compatible with the postgreSQL editor.

Procedure Create a database, call it Part. Run the DDL script first, it will create the table. Then run the DML script, it will add data to the table.

Query 1. Choosing columns List all rows and columns from the table. Type in the query shown below, make sure you type in the semicolon at end of query.

```
SELECT * FROM Part_T;
```

Type in

```
SELECT Part, Material, Size, Cost FROM Part_T;
```

In the first query the symbol * represents all columns, in the second query you have listed all the column names explicitly.

Write a query to list only Part, Material and Cost from the Part_T table.

Query 2. Using comparison operator In this query we want to filter rows, i.e. list only those rows that meet a certain criteria.

Type in the query shown below, it will list all parts that cost less than $1.00. *Note:* If we want to include parts that cost $1.00 or less than $1.00 then we would use the <= operator.

```
SELECT Part, Material, Cost
FROM Part_T
WHERE Cost < 1.00;
```

Write a query to list `Part`, `Size` and `Cost` for Parts that cost greater than $2.10.

Query 3. Using an aggregate function Here is a query to find the average cost of all parts made from `BRASS`.

```
SELECT AVG( Cost )
FROM Part_T
WHERE Material = 'BRASS';
```

Write a query to find the average cost of all screws.

Query 4. Using Substrings The following query will is the first three letters of Parts. Notice the keyword `FROM` inside the function `SUBSTRING`, it has a different purpose from the second `FROM`, which is used to chose table names.

```
SELECT SUBSTRING( Part FROM 1 FOR 3 ) FROM Part_T;
```

Write a query to list the first five letters of all `Material`.

2.20 Exercise 3 - Queries on Part_T Table - Order of Precedence

Objective

1. Write queries using Part_T table using logical operators
2. Learn Order of Precedence
3. Use brackets to change Precedence

Reference

1. Postgres documentation postgresql-9.5-US.pdf

Submission Complete a short online quiz on SQL, make sure you understand all queries.

Background Read the background from the previous lab.

Procedure Complete the queries by running them, then write your own queries to produce the required result.

Example to Demonstrate Order of Precedence - Filtering rows We begin with 36 rows in the Part_T table. Verify the data in the Part_T table. Type in the query shown below.

```
SELECT * FROM Part_T;
```

If we need to display all rows with Part as BOLT and NUT the query is

```
SELECT Part, Material, Size
FROM Part_T
WHERE Part = 'BOLT' OR Part = 'NUT';
```

Run the above query, it should display 18 rows. It displays BOLTS and NUTS of all sizes small, medium and large. If we want to display all MEDIUM sized NUTs and BOLTs we intuitively add another condition to the query - as shown below.

```
-- Incorrect query
SELECT * FROM Part_T
WHERE
Part = 'BOLT' OR Part = 'NUT'
AND
Size = 'MEDIUM';
```

Figure 2.8: Query does not display the desired result

Run the query in figure 2.8, it should display 12 rows. It displays only NUTs of MEDIUM size, BOLTs and still shown in all sizes. The query has the correct syntax but does not give the desired result. We are expecting only six rows. There are two ways to get the desired result.

```
-- Solution 1
SELECT * FROM Part_T
WHERE
Size = 'MEDIUM' AND Part = 'BOLT'
OR
Size = 'MEDIUM' AND Part = 'NUT';
```

Figure 2.9: Solution 1

```
-- Solution 2
SELECT * FROM Part_T
WHERE ( Part = 'BOLT' OR Part = 'NUT' )
AND Size = 'MEDIUM';
```

Figure 2.10: Solution 2

For the query in figure 2.8 the `AND` operator has precedence over the `OR` operator. Brackets have the highest precedence - the query in figure 2.10 illustrates this. The order or precedence of logical operators is `NOT`, `AND`, `OR`. *Aside:* We do not use the exclusive `OR` operator `XOR` in this exercise. The order is, `NOT`, `AND`, `XOR`, `OR`.

Query 2. Part_T Table List all `SMALL` parts made from `BRASS` or `STEEL`.

Query 3. Part_T table List all `SMALL` parts made from `BRASS` or `STEEL` costing less then one dollar.

2.21 Lab 2 - Retrieve data from world database.

Objective

1. Retrieve data from world database.
2. Use String functions and pattern matching to filter rows.
3. Use aggregate functions for simple statistical operations.

Reference

1. General SQL reference URL http://www.w3schools.com/sql/default.asp
2. postgres Reference Manual postgresql-9.6-US.pdf or version 10.
3. PostgreSQL reference sheet. URL http://www.petefreitag.com/cheatsheets/postgresql/ This file contains psql commands as well as pgAdminIII commands, exclude psql commands for this lab.

Background

You will need world database to complete this lab. You would have installed it in Lab 01. Before answering a question you should try the example provided. This example will lead you to the answer to the question.

Submission

Complete online Lab quiz.

Procedure

In this lab you will be using simple SQL statements to manipulate data from the world database. Read the questions and try the SQL statements. Make a note of each command or statement you execute on paper or in a text editor. You need to answer online questions to be awarded the marks for this lab. Attempt the quiz after you have completed all questions in this lab. Make sure you sign the attendance sheet. You are encouraged to consult with your colleague and seek assistance from your instructor to complete this lab.

Clauses of the SELECT statement: SELECT - List columns, and expressions, to be returned from the query
FROM - Indicate table(s) or view(s) from which data will be obtained
WHERE - Indicate conditions for a row that will be included in the result
GROUP BY - Indicate categorization of results
HAVING - Indicate the conditions under which a category (group) will be included. HAVING is used with GROUP BY.
ORDER BY - Sorts result according to specified criteria
GROUP BY and HAVING commands are done in later labs.

Type in the following SQL query and observe the results.

```
SELECT * FROM country;
```

Query 1. How many rows are displayed from the country table?

Try the following statement.

```
SELECT name, surfacearea, lifeexpectancy FROM country;
```

Try to match this simple statement with the SQL General Syntax. Notice the `column_list` in this statement.

Query 2. The SQL statement that displays `code`, `name` and `continent` from the country table is?

Sort using `ORDER BY` Try the following statement.

```
SELECT id, name
FROM city
ORDER BY name;
```

To sort in reverse alphabetical order use the keyword `DESCENDING` or `DESC` for short. By default the sort order is `ASCENDING`, you may insert the keyword `ASCENDING` to explicitly declare the sort order. Postgres permits only short forms `ASC` or `DESC`, other `SQL` implementations allow both the full word and the short form.

Try the following statement, observe the results, compare with the previous statement:

```
SELECT id, name
FROM city
ORDER BY name DESC;
```

Query 3. Write a SQL statement to display `code`, `name` and `continent` from the `country` table, with country name in alphabetical order. *Reference:* To review general syntax of SQL statement, refer page 30, figure 2.7 and page 18, figure 2.1, SQL Statement Processing Order.

Query 4. Write a SQL statement to display the country name and surface area from the country table. The country with largest surface area should display first, the country with the smallest area is displayed last.

Pattern Matching Try the following statements, observe the two different results.

```
SELECT Code, Name FROM Country WHERE Name LIKE 'G%';
SELECT Code, Name FROM Country WHERE Name LIKE '%g';
```

Query 5. Write a SQL statement that displays countries with name ending in the letter `m`?

Query 6. What is the average LifeExpectancy of people of the world? Stated differently, find the average life expectancy of people of all the countries in the world.

Query 7. How many Canadian cities are listed? Hint: Use the country table to find the country code for `Canada`, then use this code to query the city table.

Query 8. Write a `SQL` statement to list all cities in Canada, using the `city` table. Do not use the clause `CountryCode = 'CAN'` to filter the rows, instead use the text `'Canada'` in the country table, retrieve the `Code` and equate it to `CountryCode`.

Query 9. List all cities from Quebec in Canada and all cities from Puebla district in Mexico. There are atleast two ways you can write this query. *Note:* Quebec is written with `e` as an accute accent (aigu). If your keyboard cannot produce this character then use the following syntax

```
District LIKE 'Qu_bec'
```

The underscore character will match all characters at that position.

Query 10. List all countries, display only the first 5 characters of the country name. Refer query titled *Using Substrings* on page 36

3

SQL - INSERT, UPDATE, DELETE Statements

Objective

- Type in and run sample SQL queries to add, delete and modify data.
- Construct queries based on sample queries.

3.1 INSERT Statement

There are several ways to add data to a table. Using a DML `INSERT` statement is the simplest method. An `INSERT` statement has table specifications and data values. Table specifications include table name and attributes (column list), insert specification includes a `VALUES` clause followed by data items. Specifying the list of attributes with the `VALUES` clause is a robust method. When the `VALUES` clause is not specified the statement relies on the order of the attributes as defined in the DDL statement. The `INSERT` statement will fail if the attributes are rearranged in the table definition. Character, i.e. `CHAR` and `VARCHAR` data items are enclosed in single quotes. Quotes are not used for numeric values.

3.1.1 General Syntax for INSERT

```
<insert statement> ::=
INSERT INTO <table specification> <insert specification>

<insert specification> ::=
[ <column list> ] <values clause> |
[ <column list> ] <table expression>
```

Figure 3.1: SQL INSERT Statement General Syntax. Ref: [16, Page 932.]

3.2 UPDATE Statement

As the name implies, and UPDATE statement is used to modify data values in an existing record. It can add data if there is no previous data or modify already existing data. Be careful, you must specify a row, or rows, that you want changed. Not specifying the condition will change all rows in the table.

3.3 DELETE Statement

Use the DELETE statement to remove rows from a table. Similar to an `UPDATE` statement, be careful and specify the rows that you want deleted. A common practice in the workplace is to first test the filter, i.e. the `WHERE` clause to confirm the rows that will be deleted and then use the condition in the `DELETE` statement.
The SQL statement `DELETE FROM <tablename>` will remove all rows in a table. For example,
`DELETE FROM Part_T` will delete all rows in the `Part_T` table. If you do run this statement, data can be restored by running the `Part-DDL.sql` file.

3.4 Review Questions

1. Identify the statement that inserts a row in the city table with city Oshawa in Ontario.
   (a) `INSERT CITY VALUES( '6002', 'Ontario', 'CAN', 'Oshawa', 140000 );`
   (b)
   ```
   ADD INTO CITY( CountryCode, District, ID, Name, Population )
   VALUES ( 'CAN', 'Ontario', 6002, 'Oshawa', 140000 );
   ```
   (c)
   ```
   MODIFY INTO CITY( CountryCode, District, ID, Name, Population )
   VALUES( 'CAN', 'Ontario', 6002, 'Oshawa', 140000 );
   ```
   (d) `INSERT INTO CITY VALUES( '6002', 'Oshawa', 'CAN', 'Ontario', 140000 );`
   (e)
   ```
   INSERT INTO CITY( CountryCode, District, ID, Name, Population )
   VALUES( 'CAN', 'Ontario', 6002, 'Oshawa', 140000 );
   ```

2. The command
   `INSERT INTO CITY VALUES( 'British Columbia', 'Kelowna' );` is used to add city Kelowna in the Name column, and British Columbia in the district column. The SQL command will not work because
   (a) The keyword table is missing
   (b) The correct command is `MODIFY`
   (c) The field names must be specified if inserting some of the fields
   (d) Possibly because `Kelowna` already exists in the table

3. What will the following statement do

   ```
   DELETE FROM City WHERE CountryCode = 'NDL';
   ```

   (a) Deletes all countries which have code beginning with `NDL`

   (b) Deletes all records (rows) from the city table which has `CountryCode` as `NDL`

   (c) Deletes all columns (attributes) from the city table which has `CountryCode` as `NDL`

   (d) Delete all tables which begin with `NDL`

4. Identify the statement that will change the Population of city `Zaanstad` to `136621` in the City table

   (a) `UPDATE CITY SET Population = '136621' WHERE Name = 'Zaanstad';`

   (b) `UPDATE CITY SET Population = 136621 WHERE Name = 'Zaanstad';`

   (c) `UPDATE COUNTRY SET Population = 136621 WHERE Name = 'Zaanstad';`

5. Identify the statement that will delete all rows from the `City` table for cities in `USA` in District `Utah`

   (a) `DELETE FROM CITY WHERE CountryCode = 'USA'OR DISTRICT = 'Utah';`

   (b) `DELETE FROM CITY WHERE CountryCode = 'USA'AND DISTRICT = 'Utah';`

   (c) `DELETE FROM CITY WHERE DISTRICT = 'Utah';`

   (d) `DELETE FROM CITY WHERE CountryCode = 'USA';`

3.5 Exercise 4 - Adding Data to Part_T Table

Objective

1. Work on a table titled Part_T to reinforce basic SQL syntax
2. Use SQL INSERT statements to add data to a table.
3. Compare INSERT statements with column names and INSERT statements without column names,

Submission Complete the online quiz for this exercise.

Background This table is called Part_T, it has four columns they are Part, Material, Size and Cost. The DDL and DML scripts are provided. *Note:* There is a difference between Part and Part_T. Part is a column name, Part_T is the name of the table. Type in each query, do not cut and paste the queries from the .pdf file, characters in this .pdf document are not compatible with the postgreSQL editor.

Procedure Create a database, call it Part. Run the DDL script first, it will create the table. Then run the DML script, it will add data to the table. You may use the database created from an earlier exercise if you have it on your computer.

Query 1. Adding a row to the Part_T table. Type in the following DML statement. It will add a column to the PART_T table. *Note:* At this stage you may run this statement more than once, later we shall learn about CONSTRAINTS that will prevent the user from adding duplicate rows to tables.

```
INSERT INTO Part_T( Part, Material, Size, Cost )
VALUES ( 'NAIL', 'STEEL', 'SMALL', 0.15 );
```

Type in a SQL statement to check your results.

Construct, type and test a query in postgres to add a row to the Part_T table that contains the following data values. In your own handwriting copy the query in the space provided.
Part: HAMMER, Material: STEEL, Size: SMALL, Cost: 7.15

Query 2. Changing the order of column names. Study the following statement, notice that the order of attributes have changed. Will the statement run successfully? Type in and observe the results. Use an appropriate SELECT statement to check the results.

```
INSERT INTO Part_T( Size, Part, Material, Cost )
VALUES( 'SMALL', 'SCREW', 'TITANIUM', 3.5 );
```

Write your observation in the space below.

Query 3. Adding data without specifying attributes. Type in the following `INSERT` statement and run it. Notice, it does not have the attributes listed.

```
INSERT INTO Part_T VALUES( 'TITANIUM', 'BRACKET', 'SMALL', 7.5 );
```

Did the data for Part and Material get added in the correct column? Write your observations.
This method is not robust it is not recommended. What is another disadvantage in using this method?

Query 4. Adding data to selected attributes. In this query size and cost of the part are not entered.

Type this query and confirm the results. How is the missing data represented in the table? Does the cost show as zero or a blank value?

```
INSERT INTO Part_T( Part, Material ) VALUES( 'RIVET', 'TITANIUM');
```

Write a statement to add the following data.
Part: `RIVET`, Cost: `3.75`

Query 5. Adding multiple rows with a single `INSERT` statement. Type in the following statement and observe the results. A single insert statement is used, it adds four rows to the table.

```
INSERT INTO Part_T( Part, Material, Size, Cost )
VALUES( 'Tire', 'V. Rubber', '29 in', 12.00 ),
('Tube', 'Rubber', '29 in', 6.00 ),
('Wheel Tape', 'Fabric', '29 in', 1.25 ),
('Pump', '', 'Regular', 35.99 );
```

3.6 Exercise 5 - The UPDATE Statement

Objective

1. Change data in a table using SQL UPDATE statements on Part_T table.
2. Change data in a single row.
3. Change data in specific rows using a filter.
4. Change data in all rows, with *caution*.

Reference

1. Postgres documentation postgresql-9.5-US.pdf

Submission Complete a short online quiz on SQL UPDATE.

Procedure Type and run the sample queries, then write your own queries to produce the required result. Write the queries in the space provided. Observe the changes made by the UPDATE statement *before* and *after* the statement is run.

Change in cost of a part. We begin with 36 rows in the Part_T table. Run Part-DDL.sql and Part-DML.sql in a database titled Part, if not done from the previous lab. Verify the data in the Part_T table. Type in the query shown below.

```
SELECT * FROM Part_T;
```

Filter the rows by typing:

```
SELECT * FROM Part_T WHERE PART = 'BOLT' AND MATERIAL = 'BRASS';
```

Reminder: Type in the data values in upper case, i.e. BOLT and BRASS You should see three rows in the result set. The cost of a medium size brass bolt has reduced to $1.35 from $1.40. We need to change this amount. Here is the SQL UPDATE statement to make the change.

```
UPDATE Part_T SET Cost = 1.35
WHERE PART = 'BOLT'
AND MATERIAL = 'BRASS'
AND SIZE = 'MEDIUM';
```

Verify that the change has been made, using the following statement.

```
SELECT * FROM Part_T WHERE PART = 'BOLT' AND MATERIAL = 'BRASS';
```

Observation: The UPDATE statement has three components, the table name, SET clause to specify the new value, the filter conditions using WHERE clause. A single row is filtered in the SELECT statement because all three conditions are specified, part, size and material.

Query 1. - Changing Data, single row Write a query to change the cost of a small aluminium washer to $2.55 from the current $2.60

Use appropriate case in the data values, translate the three conditions, small, aluminium and washer into appropriate words. Write your UPDATE statement in the space below.

Change in cost of a part, multiple rows. Type the following query.

```
-- Updating three rows
SELECT * FROM Part_T
WHERE PART = 'NUT'
AND SIZE = 'MEDIUM';
```

Figure 3.2: Updating multiple rows.

How many rows do you see in the result set from the query in figure 3.2?

The next update statement will change all three rows in the table, i.e. nuts of medium size.

```
UPDATE Part_T SET Cost = 1.05
WHERE PART = 'NUT'
AND SIZE = 'MEDIUM';
```

Verify the change by typing the query in figure 3.2.

Query 2. - Changing Data, multiple rows. Write a query to change all small washers to TINY washers. Use appropriate case in the data values, translate the two conditions, small, washer into appropriate data values. Write your SQL statement below.

Alternative Statement: Study the following statement, it has the same result as the previous statement, but is terse.

```
UPDATE Part_T
SET SIZE = 'TINY'
WHERE( SIZE, PART ) = ( 'SMALL', 'WASHER');
```

The following statement alters two fields with two conditions. It changes the cost of all medium size bolts to $1.30 and changes the material to COPPER.

```
UPDATE Part_T
SET( COST, MATERIAL ) = ( 1.30, 'COPPER' )
WHERE( SIZE, PART ) = ( 'MEDIUM', 'BOLT' );
```

Using UPDATE without a WHERE clause. The following SQL statement would very rarely be required, it should be used with caution. It will update all rows in a table.

```
UPDATE Part_T SET Cost = 0.00;
```

Run the statement, you have the DML file which you can run and restore the data. This example illustrates the consequences of an UPDATE statement without filtering rows. *Aside:* There is some merit to using

```
UPDATE Part_T SET Cost = Cost * 1.13;
```

The statement will add 13% to the cost of all items, it must still be used carefully.

3.7 Exercise 6 - Deleting Data from Part_T Table

Objective

1. Use SQL DELETE statements to remove data from a table.
2. Using WHERE clause to filter rows to delete.

Submission Complete the online quiz for this exercise.

Background Attempt this exercise after you have completed the INSERT and UPDATE exercises. Use the Part_T table for this exercise.

Query 1. Deleting a row from the Part_T table. First let us confirm that the row filter works as expected. Type in the statement `SELECT * FROM Part_T WHERE Part = 'BOLT'AND Material = 'STEEL'AND Size = 'SMALL';`

After confirming type the following DELETE statement.

```
DELETE FROM Part_T WHERE Part = 'BOLT'AND Material = 'STEEL'AND Size = 'SMALL';
```

Read the message in the output pane of pgAdmin. Can you confirm that only one row was deleted?

Write a DELETE statement to remove a row with the following data. Part: SCREW, Material: BRASS, Size: SMALL.

Query 2. Deleting more than one row. This time the row filter is more relaxed, it will give you three rows. Type in the following statement.

```
SELECT * FROM Part_T WHERE Part = 'BOLT' AND Material = 'STEEL' AND Size = 'SMALL';
```

Now run the following DELETE statement.

```
DELETE FROM Part_T WHERE Part = 'NUT' AND Material = 'BRASS';
```

Write a DELETE statement to remove rows with the following condition.
Material: TITANIUM.
How many rows were deleted?

3.8 Lab 3 - Add, Modify and Delete data from `world` database.

Objective

1. Add, edit and remove data to `world` Database.
2. Appreciate constraints.

Reference

1. postgres Reference Manual `postgresql-9.6-US.pdf` or version 10.

Submission Write your answers to each of the queries in the workbook. Complete the online quiz.

Background Complete all previous exercises in this chapter before attempting this lab. Some queries will depend on previous queries that you have successfully run. Do the queries in the order given.

Procedure After completing these queries, the world database will be altered, run `world-pg.sql` to restore the database to its original state.

Query 1. Adding data to `city` table. The following statement will add a row to the city table.

```
INSERT INTO City( ID, Name, CountryCode, District, Population )
VALUES( 7000, 'Guelph', 'CAN', 'Ontario', 131794 );
```

Type in and run the statement, observe the results.

Add a row with the following data.
Id: `7002`, Name: `Montebello`, CountryCode: `CAN`, District: `Quebec`, Population: `983`.

Query 2. Add Data to `Country` Table. Add the following data to the `Country` table.
Code: `SRB`, Name: `Serbia`, Continent: `Europe`, Region: `Eastern Europe`,
Surface Area: `88361`, IndepYear: `2006`, Population: `6963764`

Query 3. Correcting a query. The following insert statement is incorrect, there are two syntax errors. Correct it by making the appropriate changes and run the statement. Write the corrected statement in the space below.

```
INSERT INTO Country( Code, Name, Continent, Region, SurfaceArea, IndepYear, Population )
VALUES( 'MNE, 'Montenegro', 'Europe', 'Eastern Europe' 13812, 2006, 622359 );
```

Query 4. Update a row. Determine the fields in the country table. There are several ways to find out. One simple method is to list a few rows with all attributes in a table and see the top row of the result set. For example the SQL statement `SELECT * FROM Country;` will list all columns.

Update the field `GovernmentForm` of Serbia to `Republic`. The `UPDATE` statement should change only *one* row.

Query 5. Changing the population. Change the population of Canada to `37971020`.

Query 6. Removing rows. List all rows from the `CountryLanguage` table from Canada. Observe that the list is outdated. Write a SQL statement to remove all entries from Canada in the `CountryLanguage` table.

Query 7. Adding current data in `CountryLanguage` table. The folder `./Lab/Lab - Insert Update Delete` has a file titled `CanadianLanguages2020.sql` run the file, it updates current data into the table. Write an update statement to list rows than have English and French as languages, you should get two rows. Your result should be similar to table 3.1. *Tip:* Note the operator precedence in this query.

CountryCode	Language	Isofficial	Percentage
CAN	English	0	56.8
CAN	French	0	20.9

Table 3.1: Official Languages in Canada

Notice the `Isofficial` field is 0, implying the language is not an offical language. Write a SQL statement that will update these two rows with `1` in the official language. *Tip:* The `WHERE` condition from the previous `SELECT` statement can be used in this query.

Query 8. Single update statement to change more than one field. Write an SQL statement to change the population of `Algeria` to `43600000` and `gnp` to `684689`; use only one update statement.

Query 9. In city table the District `British Columbia` has been misspelt as `British Colombia`. Identify the statement that will correct the District Name.

Query 10. Write an Update or Delete Query of your choice.

4

SQL - CREATE, ALTER, DROP Statements

4.1 Introduction

Creating and removing database objects are covered in this chapter. We begin by creating tables within a database and removing them.

4.2 SQL CREATE TABLE clause

A simple example is shown in figure 4.1

```
DROP TABLE IF EXISTS Part_T;

CREATE TABLE Part_T(
Part VARCHAR( 20 ),
Material VARCHAR( 20 ),
Size VARCHAR( 10 ),
Cost NUMERIC( 9,2 )
);
```

Figure 4.1: SQL CREATE TABLE

A few items to note in the code.

1. The statement CREATE TABLE is followed by the name of the table.
2. The last line does not have a comma
3. The closure of the CREATE statement is ended by a close bracket followed by a semicolon `);`
4. There cannot be two tables with the same name in a database. We are learning the SQL syntax, and we want to run the DDL file several times. Before we write CREATE TABLE we first DROP it using the syntax `DROP TABLE IF EXISTS`. When the statement is run the first time, SQL will issue a *notice*.

`NOTICE: table "part_t" does not exist, skipping`. When the same statement is run the second time the notice does not appear, because it first removes the table and then creates it. Try to comment the `DROP TABLE` statement and run the DDL statement. You will get an error.

5. The IF EXISTS clause in DROP TABLE runs conditionally, only if the table exists, it will attempt to `DROP` it.

4.3 SQL ALTER TABLE

If an additional attribute needs to be added to the table after it is created the following statement is used.
For example, to add Quantity on Hand `QoH` the following statement is used.

```
ALTER TABLE Part_T ADD COLUMN QoH Numeric( 5, 2 );
```

An additional column is created. Existing data in the table is not affected, the new column will not have any data.

To remove an existing column the `DROP COLUMN` clause is used.
For example to remove the newly created `QoH` column

```
ALTER TABLE Part_T DROP COLUMN QoH;
```
is used.

4.3.1 Comments and Observations

1. A table cannot have two column with the same name, `ADD COLUMN` statement will fail the second time it is run.
2. For the `DROP COLUMN` the data type `Numeric( 5, 2)` is not required.
3. A non existent column cannot be dropped, SQL will flag an error. A conditional statement can be written such as
   `ALTER TABLE Part_T DROP COLUMN IF EXISTS QoH;`
4. ALTER TABLE can be used to add `CONSTRAINT`s to a table. Constraints are discussed in a later chapter.

4.4 Review Questions

1. Referring to the world database - Which SQL statement will remove the column `IsOfficial` from table `CountryLanguage`
   (a) `ALTER TABLE CountryLanguage.IsOfficial DROP COLUMN;`
   (b) `ALTER TABLE CountryLanguage DELETE COLUMN IsOfficial;`
   (c) `ALTER TABLE CountryLanguage DROP COLUMN IsOfficial;`
   (d) `ALTER TABLE CountryLanguage DROP FIELD CountryLanguage.IsOfficial;`
   (e) `ALTER TABLE CountryLanguage.IsOfficial DELETE COLUMN;`
2. Identify the command that adds a column to the `CITY` table in the world database. The column is titled `Area` and stores the area in square kilometers rounded to the nearest kilometer, and able to easily perform simple arithmetic calculations on the area.
   (a) `ALTER TABLE CITY ADD Area INTEGER;`
   (b) `ALTER TABLE CITY ADD Area CHAR( 10 );`
   (c) `ALTER TABLE CITY ADD Area VARCHAR( 10 );`
3. The _____ is the structure that contains descriptions of objects such as tables and views created by users.
   (a) SQL
   (b) schema
   (c) master view
4. A `CREATE` command adds a table. Identify the command that will remove the table.
   (a) `DELETE`
   (b) `DROP`
   (c) `TRUNCATE`
   (d) `UNPACK`
5. Identify the SQL command that adds or deletes a column from a table.
   (a) `CREATE VIEW`
   (b) `CREATE TABLE`
   (c) `ADD COLUMN` or `DELETE COLUMN`
   (d) `ALTER TABLE`
   (e) `MODIFY TABLE`

5

SQL - Constraints

5.1 Introduction

Data can be stored in rows and columns in application such as a word processor or a spreadsheet. It is a relation database with *constraints* that can ensure high data quality. Some constraints are specified in DDL using the CONSTRAINT key word, other constraints have their own clauses. This chapter introduces constraints, the `NOT NULL`, `DEFAULT`, `PRIMARY KEY` and `FOREIGN KEY` `UNIQUE KEY` and `CHECK CONSTRAINT` to validate data.

5.2 `NULL` values

A datatype in SQL can have a NULL value, there can be three possibilities;

1. a value is inappropriate or not possible
2. a value is not known at the time other data items are available, the value is appropriate
3. a value that has not been entered in the database, it may be known and is appropriate

Consider the following data items in the Patient table shown in figure 5.1

Patient Name	Date of Birth	Date of Death	Phone
Wut Wutuaz	21 Dec 1901	23 Sept 2001	NULL
Kef Kefaeq	23 Mar 1956	NULL	613-725-2981
Yoy Yoyoiz	NULL	NULL	NULL
Lum Lumuef	5 Apr 1946	NULL	NULL

Table 5.1: Patient Table

In the first case Patient `Wut` is deceased, a phone number for that person may not be appropriate. The second patient `Kef` is alive, she has all details except date of death. Patient `Yoy` does not have any details except her name,

and Lum does not have Date of Death and Phone number.

An example of the DDL statement to create a table to store the above data is shown in figure 5.1. A NULL specification does not mean that all values have to be NULL, it means NULL values are allowed in the field. In the DDL statement it is mandatory to have Patient Name, this field is explicitly defined NOT NULL, without a name it is not possible to add a record for that patient. Stated differently, it is not possible to add a record for a patient with the fields Date of Birth, Date of Death and Phone.

A NULL cannot be replaced by a zero or a space. A well deigned database will have only a few selected attributes allowed to accept NULL values.

5.2.1 Implementing NULL in SQL

Figure 5.1 illustrates SQL code with NULL values. NOT NULL for Name makes data entry in the field mandatory. A record cannot be inserted in the table without this data item. The NULL for DateOfBirth and Phone indicates that data in these field may not be entered. DateOfDeath has DEFAULT NULL; the DEFAULT keyword will automatically fill the data element with NULL if no data is explicitly entered by the user.

```
CREATE TABLE Patient(
PatientName  VARCHAR( 25 ) NOT NULL,
DateOfBirth  DATE          NULL,
DateOfDeath  DATE          DEFAULT NULL,
Phone        VARCHAR( 25 ) NULL );
```

Figure 5.1: DDL statement to create Patient Table

Exercise: Study table 8.2 on page 115, planets Mercury and Venus have NULL values for Satellies, what can you infer?

5.3 NOT NULL constraint

By specifying NOT NULL the database designer forces the user to enter data into the field. For example, consider the table definition in figure 5.1

The statement

```
INSERT INTO Patient( Name, DateofBirth, DateofDeath, Phone )
VALUES( NULL, '26-Dec-1961', NULL, '905-753-2819');
```

will fail with an error message

```
ERROR: null value in column "patientname" violates not-null constraint
```

Aside: Notice postgres converts field names to lowercase.

5.4 Prime Key Constraint

A more formal discussion on prime key is done in a later chapter. Prime Key constraint is written as `CONSTRAINT PatientName_PK PRIMARY KEY PatientName;`

A prime key constraint will guarantee that there will be only one occurrence of the data item, in this case there can be only record with a patient name. Too restrictive in a real situation, but fine for this introduction.
Compare figure 5.2 and 5.1, a new constraint has been added. PatientName_PK is an object name we provide.
Important: Notice the comma at the end of the line
`Phone VARCHAR( 25 ) NULL,`

```
CREATE TABLE Patient(
PatientName VARCHAR( 25 ),
DateOfBirth DATE NULL,
DateOfDeath DATE DEFAULT NULL,
Phone VARCHAR( 25 ) NULL,
CONSTRAINT PatientName_PK PRIMARY KEY( PatientName ) );
```

Figure 5.2: DDL statement to create Patient Table with Primary Key Constraint

After the constraint has been added, add the four patient records, the following `INSERT` statement will fail.

```
INSERT INTO Patient( PatientName, DateofBirth, DateofDeath, Phone )
VALUES( 'Yoy Yoyoiz', '26-Dec-1961', NULL, '905-753-2819');
```

The error message is `ERROR: duplicate key value violates unique constraint "patientname_pk"`
`DETAIL: Key (patientname)=(Yoy Yoyoiz) already exists.`

Experiment with the DDL-DML file titled `Patient-DDL-DML-PK-Constraint.sql`, this file can be found in the folder `Course Content->Lecture Demonstration->Constraints`. Remove the PRIMARY KEY constraint (and the comma from the previous line), then the above insert statement will run, you will have two patient records with the same `PatientName`.

5.5 Foreign Key Constraint

A data value in the foreign key in the child table refers to the same data value in the parent table. In our example, say, a patient has more than one address, home address, work address and more, as shown in figure 5.5. We create

an additional table titled `PatientAddress` as follows.

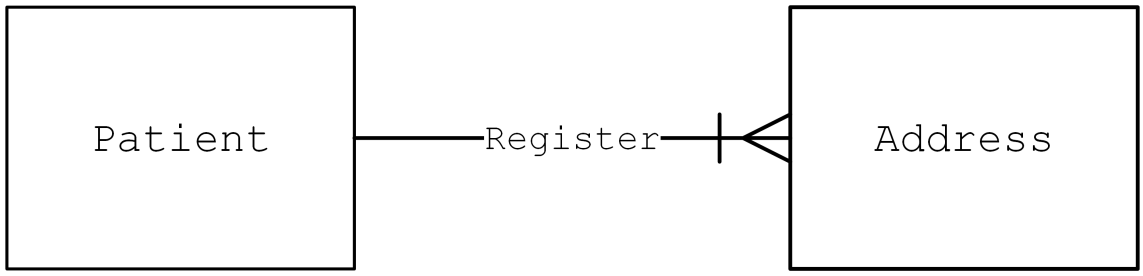


Figure 5.3: Patient Address ERD

```
-- the following statements will succeed, because PatientName exists in the parent table
INSERT INTO PatientAddress( PatientName, Address, City, PostalCode )
VALUES( 'Yoy Yoyoiz', '72 Marine Drive', 'Toronto', 'M4J5C2');
INSERT INTO PatientAddress( PatientName, Address, City, PostalCode )
VALUES( 'Yoy Yoyoiz', '220 Elm Street', 'Gananoque', 'K7G9T5');

-- the following statements will fail, because PatientName does not exist in the parent table
INSERT INTO PatientAddress( PatientName, Address, City, PostalCode )
VALUES( 'Syd Sydney', '763 Fort Crescent', 'Perth', 'E7H2C6');
```

Experiment with the file titled
`Patient-DDL-DML-FK-Constraint.sql` in folder `Course Content->Lecture Demonstration->Constraints`.

5.6 Unique Key Constraint

A unique key constraint ensures that an attribute or a set of attributes are unique in the relation. A unique key constraint allows `NULL` values; this differentiates it from a Primary key Constraint. There can be more than one unique key constraint, compare this feature with a Primary Key Constraint; there can be only *one* Primary Key Constraint.

A unique key constraint:

1. is optional, a relation may not have any unique key constraint.
2. can be composite, in this case the complete set of attributes should be unique. Additionally, it is possible to have part of the unique key to be null.
3. can be referred by a foreign key in another relation.

5.6.1 Example, unique key constraint

Two tables are shown in this example, the `Citizen_T` table, the parent table, has two unique key constraints. The table is based on these rules:

- each citizen record in the table must have a Social Insurance Number (SIN).
- a social insurance number is unique to a citizen.
- a citizen may not have a Drivers Licence, if a citizen has a drivers licence it must belong to only one citizen.
- a citizen may not a Health Card, if a citizen has a health card it must belong to only one citizen.

Note:

– the `DEFAULT NULL` in the `DriverLicence` attribute.

– two different methods to enforce the Unique Key constraint. The HealthCard attribute has a Unique Key constraint too. This method does not have a user defined constraint name. The DBMS will provide a constraint name; in this case it is `citizen_t_healthcard_key`. Observe the icon in `pgAdminIII` on the constraint name.

– by defining a constraint name, such as `D_Licence_UK` the database administrator can `DROP` the constraint at a later time by using the `ALTER TABLE` statement.

– A Unique Key constraint cannot be dropped if there are foreign key constraints on that constraint. The foreign key constraint must be dropped first.

```
CREATE TABLE IF NOT EXISTS Citizen_T(
   SIN            CHAR( 9 ),
   FirstName      VARCHAR( 30 ),
   LastName       VARCHAR( 30 ),
   DriverLicence  VARCHAR( 40 ) DEFAULT NULL,
   HealthCard     VARCHAR( 40 ) UNIQUE,
   CONSTRAINT SIN_PK PRIMARY KEY( SIN ),
   CONSTRAINT D_Licence_UK UNIQUE( DriverLicence )
);

-- create AutoInsurance_T table
-- a driver in the citizen table can have one or more vehicles insured
CREATE TABLE IF NOT EXISTS AutoInsurance_T(
   PolicyID       CHAR( 20 ) PRIMARY KEY,
   DriverLicence  CHAR( 40 ),
   VIN            CHAR( 35 ),
   CONSTRAINT D_Licence_FK FOREIGN KEY( DriverLicence) REFERENCES Citizen_T( DriverLicence )
);
```

Figure 5.4: Unique Key Constaint, Example

5.6.2 Exercise

Study the file `Unique-Key-Constraint-DDL-DML.sql` in the folder
Course Content→Lecture Demonstration→Constraints, run the `ALTER TABLE` statements in this file, observe the reason for failure, if any, or success.

5.7 Validating Data

Validating user data before it is stored in a database increases data reliability. The `CHECK` constraint in SQL is used for this purpose it is done at the DDL level.

5.7.1 Validating a Numeric Value

```
--  Example to demostrate a check on a numeric value.
DROP TABLE IF EXISTS Product_T;
CREATE TABLE Product_T(
Prod_Code CHAR( 5 ) PRIMARY KEY,
Prod_Description VARCHAR( 25 ),
Prod_QoH NUMERIC( 7, 3 ) CONSTRAINT Positive_QoH_validate CHECK( Prod_QoH >= 0 ) );

-- Data successfully inserted
INSERT INTO Product_T( Prod_Code, Prod_Description, Prod_QoH )
VALUES( 'K001', 'Organic Kale', 30 );

--  This statement will fail, due to the check constraint
--  run the statement and notice the error message
INSERT INTO Product_T( Prod_Code, Prod_Description, Prod_QoH )
VALUES( 'C001', 'Collard Green', -15 );
```

Figure 5.5: `CHECK` constraint

In 5.5 `Positive_QoH_validate` is the name given to the constraint. Postgres will provide its own name if the user does not provide it. The second statement will fail; Product Quantity, (`Prod_QoH`), should be greater than or equal to zero.

Exercise: Drop the constraint using the statement.

```
ALTER TABLE Product_T DROP CONSTRAINT Positive_QoH_validate;
```

The statement with negative value for `Prod_QoH` now inserts data.
Can you add the constraint again? Try to run the statement following and determine the result.

```
-- Can you put the constraint back?
ALTER TABLE Product_T ADD CONSTRAINT Positive_QoH_validate CHECK( Prod_QoH >= 0 );
```

Aside: Alternatively, a constraint can be added using `ALTER TABLE` outside the `CREATE TABLE` statement.

5.7.2 Validating a Date Expression

```
-- Example to demostrate a date check in postgres at the DDL level.

DROP TABLE IF EXISTS Vaccine_T;

CREATE TABLE Vaccine_T(
FirstName VARCHAR( 25 ),
DOB DATE CHECK( EXTRACT ( YEAR FROM CURRENT_DATE ) - EXTRACT( YEAR FROM DOB ) > 10 )
);

INSERT INTO Vaccine_T( FirstName, DOB ) VALUES( 'Paas', '1997-12-21');
-- INSERT INTO Vaccine_T( FirstName, DOB ) VALUES( 'Fayel', '2012-02-25');

SELECT * FROM Vaccine_T;
```

A certain vaccine is available to patients above 10 years of age. `Paas` and `Fayel` try to get the vaccine. When the Date of Birth, `(DOB)` is entered, data for Paas goes through, but the system does not accept the data for Fayel. Uncomment the second `INSERT` statement, rerun the file and test the syntax. The function `CURRENT_DATE` fetches the date from the operating system.

Alternatively a CONSTRAINT can be added explicitly as shown below.

```
CREATE TABLE Vaccine_T(
FirstName VARCHAR( 25 ),
DOB DATE,
CONSTRAINT OVER_19 CHECK( EXTRACT ( YEAR FROM CURRENT_DATE ) - EXTRACT( YEAR FROM DOB ) > 19 )
);
```

Exercise: Put a constraint on `Prod_Price` to be greater than zero in `Inventory_DDL.sql` in the lab on page 68. Test your constraint by adding a product in `Inventory_DML.sql` file with a negative value or a zero. Then complete a short quiz on RANGE CONSTRAINT.

Thought Question: What would be an advantage in enforcing a constraint at the DDL level? It could have been enforced at the programming stage.

5.8 Review Questions

1. A prime key
   (a) can be `NULL`
   (b) cannot be `NULL`
2. An attribute or a combination of attributes that uniquely identifies each row in a relation is
   (a) Foreign Key

(b) Prime Key

(c) Composite Attribute

(d) Multivalued Attribute

3. An attribute in a relation that serves as the primary key of another relation in the same database.

   (a) Foreign Key

   (b) Prime Key

   (c) Composite Attribute

   (d) Multivalued Attribute

4. A composite key is a primary key that consists of more than one attribute

   (a) True

   (b) False

5. A rule that states that each foreign key value must match a primary key value in the other relation is called the referential integrity constraint.

   (a) True

   (b) False

6. Referring to the `world` database, a code segment in `DDL` to create the `City` table is shown below. What does the term `SERIAL` indicate?

```
CREATE TABLE City (
ID SERIAL PRIMARY KEY,
... ...
```

   (a) it has no meaning, `SERIAL` is an outdated syntax

   (b) city ID is a surrogte key and values are added automatically in serial order

   (c) it is a special case, primary key can be `NULL`

7. It is possible to have more than one Unique Key constraint in a table.

   (a) True

   (b) False

8. A Unique Key constraint cannot have `NULL` values

   (a) True

   (b) False

9. A unique key in a parent table can be used as a reference by a foreign key from a child table

   (a) True

   (b) False

10. Each table must have at least one Unique Key constraint.

    (a) True

    (b) False

11. A Unique Key constraint cannot be composite.

    (a) True

    (b) False

5.9 Lab 4 - Inventory Database

Objective

1. Create Database
2. Create Tables, study `DDL`
3. Write constraints in `DDL` statements
4. Add user data using `INSERT` statements
5. Differentiate between *identifying* and *non-identifying* relationships
6. Reverse Engineer Inventory database using a diagramming tool

Reference List 2.21 on page 39 of these notes.

Submission Complete the quiz. Note the queries. You will need them for the Lab exam. Write your answers to this exercise in a MS Word, or similar text document such as `Notepad++`. Upload the ER Diagram in `.pdf` or `.png` format.

Background

Scenario The model represents a small trading company dealing in (say) computer components. The company maintains a list of customers with some details such as address and phone numbers. A list of products in the inventory is maintained. Customer purchases are initiated by creating an invoice. A purchase can have one or more products, an invoice can have one or more products of the same type.

Implementation This is a small database of three tables and one associative entity, a total of four tables. The three tables are `Customer_T`, `Product_T` and `Invoice_T`. `Invoice_Line_T` is an associative entity. Study the ER diagram provided and try to identify the `Prime Keys` and `Foreign keys`.

Requirements

Create a database called `Inventory` in postgres. First load and run `Inventory-DDL`, the DDL script will create tables. Load the `Inventory-DML` script. Locate the following line,

```
INSERT INTO Customer_T  VALUES( 'C002', 'Your First Name',
'Your Last Name', '613-727-4723', '1385 Woodroffe Ave',
'Ottawa','ON', 'K2G1V8',0 );
```

A note on `CONSTRAINTS`. You can run the `DML.sql` file only once, the constraints will prevent you from running it again. If you need to run the `DML` the second time, first run the `DDL`. This will drop all existing tables and recreate them. Now you will be able to run your modified `DML.sql`.

Query 1. Modify data. Replace the text `Your First Name`, `Your Last Name` with *your* first name and *your* last name in the DML file. *Aside:* Last Name is the Family Name or Surname. Replace 0 in Balance field with another number. Replace the other five entries in the `Customer_T` table with your own data. Write the names of your colleagues sitting besides you in the lab. Do not skip this step.

After modifying the script run it. Type and run each of the statements to verify that the tables are created successfully.

```
SELECT * FROM Customer_T;
SELECT * FROM Invoice_T;
SELECT * FROM Product_T;
SELECT * FROM Invoice_Line_T;
```

Query 2. List data in sorted order. List all customers (last name and then first name), sorted by last name. Hint: Use the clause `ORDER BY`.

Query 3. List year of invoice number. List only invoice numbers and invoice dates (4-digit year) from the `Invoice_T` table. Tip: Use the `EXTRACT` function.

Query 4. List Data, identify correct table. List Invoice Number, Product Code, and Line Price. Identify the table that has these columns?

Query 5. List Data. Associate common terms with database field ames. List all product code and product description. Choose the appropriate table to complete the query.

Query 6. Filter rows. List customer names (last and first name) and city for those customers who do not live in Ottawa. Hint: Use `WHERE` clause.

Query 7. Use String functions. The city name is stored as `Ottawa`. Modify the query to list the same number of rows when the condition is written as `OTTAWA`. Tip: Use the appropriate string function shown in Table 2.2 on page 23.

Question 8. Reverse Engineer the database. Reverse Engineer the Inventory database using a data modeling tool. Rearrange the entities and relationships. Insert a text box with your name, section number, student number, course code, and course name.

Question 9. Determine identifying and non-identifying entities. Determine the *identifying* and *non-identifying* relationships. Note your answers, you will need them for the quiz. Refer to the documentation on identifying and non-identifying relationships.

Question 10. Determine the associative entity. Which one of the four entities is an associative entity? Give a reason.

Notes on Relationships An identifying relationship is represented by a solid line; a row in a child table can be uniquely identified *only* if an entry exists in the parent table. For example, in table `Invoice_Line_T` an entry can be identified only if an `Invoice_Number` exists in the `Invoice_T` table. *Aside:* The prime key `Invoice_Number` in the `Invoice_T` Table is part of the prime key in the `Invoice_Line_T` table.

A non-identifying relationship is represented by a broken (dashed) line. In a non-identifying relationship, an entry in a child table can be uniquely identified without a corresponding entry in the parent table. For example, an entry in the `Invoice_T` table can be uniquely identified by `Invoice_Number` alone, it does not need an entry from the customer table. Similarly an entry in the `Invoice_Line_T` table can be uniquely identified by a combination of `Invoice_Number` and `Invoice_Line` it does not depend on an entry in the `Product_T` table.

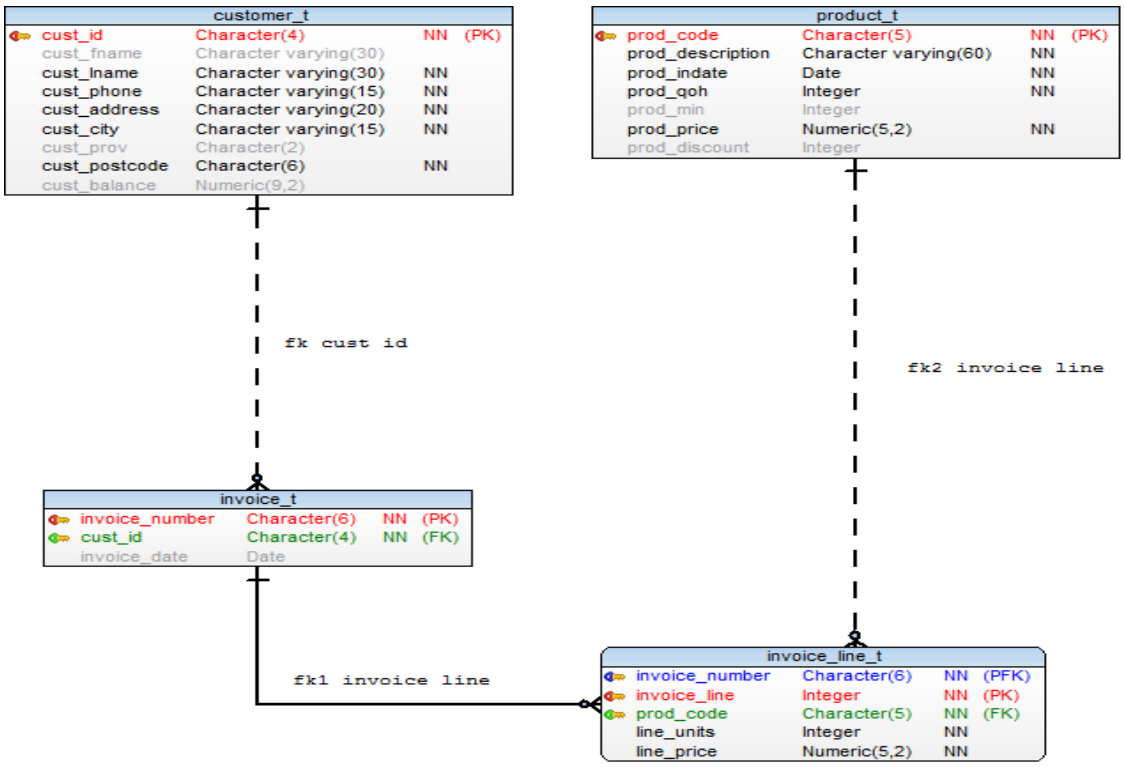


Figure 5.6: Inventory Database

5.10 Lab 5 - Query the Inventory Database

Objective

1. Explore `SQL` statement further using the `Inventory` Database
2. Use functions in `SQL`

Reference

1. Use `Inventory` database to write the SQL queries.

Submission Refer page 171 of this workbook.

Requirements

Tip: Manually look at the tables and determine what the result should be. Compare the solution you get when you run the query. If they are different, re-evaluate your SQL statements. An SQL statement may be syntactically correct, but can give unwanted results.

1. List all customers (concatenate `last name`, `first name`), and balances (`Cust_Balance`), for those customers with balances not equal to zero. Sort in descending order by balance. Make sure you have inserted some non zero values in the customer table database. For an explanation on concatenation refer section 2.9 on page 22.
2. List invoice numbers, product codes, line price, line units (quantity) and line cost (line price * quantity) for lines with line costs greater than `$500`. Sort the result in descending order by line cost. Secondary ordering by invoice number in descending order.
3. List product code, product description, product quantity on hand, product price and the inventory cost of the product (price * quantity on hand), for those products with quantity on hand greater than 50. Sort by product description in ascending order.
4. List customer last name, customer first name and customer balance for those customers with zero balance, who are in Ottawa. Sort by customer last name in an ascending order.

5. List invoice number, product code, line units and line price for those invoice lines that have line prices greater than `$600` or less than `$100`. Sort line price in descending order.
6. List customer name (concatenate `First name Last name`), and address for those customers who have last names beginning with the same first letter of your family name (last name) and have customer balances between 0 and 100 (inclusive).
7. List product description, quantity on hand and product price for those products with quantity on hand equal to 60, 70, 80, 90. Sort by quantity on hand in an ascending order.
8. List product description, quantity on hand, product price and product discount for all products with null values for product discount.
9. List names of all cities where customers are located (city names only, not customer names). Do not repeat any rows. Sort by city name in an ascending order.

10. Match all customers who are in Ottawa. Using the syntax shown below. You will need to add an appropriate string function to the code fragment. Refer in Table 2.2 on page 23. `WHERE Cust_City = 'ottawa';` Note: `ottawa` is in lowercase.

5.11 Lab 6 - Country-City Database

Objective

1. Draw a Logical ERD, using paper and pencil, from abstract and rules.
2. Write `DDL` statements with constraints.
3. Write `DML` statements.
4. Test and Verify database.
5. Reverse Engineer the database using data modeling software, verify with logical model.
6. Write simple `SQL` statements to query the database.

Reference

1. `postgresql-9.5-US.pdf`
2. Refer `Inventory-DDL.sql` from the previous lab for an example on writing `PRIMARY KEY` and `FOREIGN KEY` constraints.

Submission

1. Complete the online quiz.
2. Submit Reverse Engineered ERD of Country-City Database. Submit a `.png` or `.jpg` file that has been exported. Do not submit a screenshot.

Requirements

1. Draw a logical ERD from the abstract for **Country-City** in section 1.7 on page 6.
2. Create a database titled `Country-City`.
3. Write the `DDL` statements using metadata shown in figure 5.7 and 5.8. Write all constraints in both the tables. This indicates the details required to be stored in the country table and city table, the data type and the required length. Write the primary key constraint in `COUNTRY_T` table and primary and foreign key constraints in `CITY_T` table. Note: The datatype `INT` will automatically use 4 bytes and `BIGINT` will automatically use 8 bytes. Do not type in the size for these two data types. Mandatory data translates to `NOT NULL` in `SQL`.
4. `DDL` Files should have `DROP TABLE IF EXISTS <tablename>;`, this statement allows the creation of tables by dropping them first.
5. Run and test your `DDL` statements in the database.
6. Reverse Engineer the database and verify it with the hand drawn logical ERD.
7. Write `DML` statements to populate the data, you may use the data provided on page 175, figure B.1, or you may use your own data.
8. `INSERT` statements should be preceeded by `DELETE FROM <tablename>;`. Use current, relevant data for population, you may use online references or wikipedia. Create two or three cities for each of the countries you have chosen, use current, relevant data for city population.
9. All `DML` statements must have *all* field names. Test and run the `DML` statements.

10. Write simple `SQL` statements to verify your data.

Abstract

A country can have many cities. A country may not have any cities. A city must belong to one country.

Data Element	**Attribute**	**Data Type**	**Length**	**Constraint**
Country Code	`Cntry_Code`	`CHAR`	3	Prime Key
Country Name	`Cntry_Name`	`VARCHAR`	30	Mandatory
Country Population	`Cntry_Population`	`BIGINT`	8	DEFAULT NULL

Figure 5.7: Country Table - `COUNTRY_T`

Data Element	**Attribute**	**Data Type**	**Length**	**Constraint**
City ID	`City_ID`	`INT`	4	Prime Key
City Name	`City_Name`	`VARCHAR`	30	Mandatory
Country Code	`Cntry_Code`	`CHAR`	3	Foreign Key, references `Cntry_Code` in `COUNTRY_T` Table
City Population	`City_Population`	`BIGINT`	8	DEFAULT NULL

Figure 5.8: City Table - `CITY_T`

Note: The specifications shown above need to be translated to DDL statements. Determine i) how mandatory data translates to a DDL statement, ii) What is the keyword for variable length 30? iii) The size of `INT` and `BIGINT` are for reference, it does not have to be specified in DDL statements.

6

SQL - VIEW, GROUP BY & HAVING

6.1 Introduction to View

There are two types of views - Dynamic View and Materialized View. Each of the views, dynamic and materialized have an advantage, and a unique purpose.

6.1.1 An analogy

Imagine you are watching a landscape, you can view this landscape in two ways. (i) Use a binocular. In this case every event that takes place can be seen in the moment. A passing bird or a drifting cloud. These moments are transient, each time you pick up the binoculars you see the changed landscape, in that moment in time. (ii) Use a camera to capture the landscape. The moment is stored as an image on the device.
The first option does not take up space, it is transient and there is no persistence; compare this to a *dynamic* view. In the second option there is a snapshot of the landscape in that moment in time, it takes up storage; compare this to a *materialized* view. These views are explained below.

6.1.2 Dynamic View

A Dynamic view is also called virtual table or logical view. Some authors call it a *derived table* [16, Page 83]. The result set does not take hard disk space. Dynamic view occupies space in the DBMS as an *object*. Each time a dynamic view is invoked, the query (view) is run by referring the tables.

A complex, or cumbersome, query can be simplified by creating a dynamic view. When the view is invoked as a `SELECT` statement the query is run. Any change made to the tables will affect the result set each time the view is invoked, hence the name *dynamic* view. A dynamic view will always provide the most recent data.
Example: To create a dynamic view you would type

```
CREATE VIEW Canada_V AS
SELECT * FROM City
WHERE CountryCode = 'CAN';
```

A view is dropped by the following command line `DROP VIEW IF EXISTS Canada_V;`

The result set from the view created above can be observed using
`SELECT * FROM Canada_V;`

Achieving Physical Data Independence is an important advantage of a dynamic view. An appreciation of data independence will come after database design topics have been covered.

6.1.3 Materialized View

is a persistent view. A data set is created from the tables and stored in the database. The result set takes storage space, unlike a dynamic view where the result set does not occupy hard disk space. Each time a materialized view is run the result set that is stored is referenced.

Example: To create a materialized view you would type

```
CREATE MATERIALIZED VIEW Canada_MV AS
SELECT * FROM City
WHERE CountryCode = 'CAN';
```

A materialized view is dropped by the following command line
`DROP MATERIALIZED VIEW IF EXISTS Canada_MV;`

The result set from the materialized view created above can be observed using `SELECT * FROM Canada_MV;`

At this stage both views give the same result. Observe the schema in `pgAdmin`, materialized view has a different icon. *Note:* Both `CREATE` and `DROP` are `DDL` statements, `SELECT` are `DML` statements.

The difference between both these views are observed using the example below.

Observe the results from two `SELECT` statements above, pay attention to the number of rows in each case, they should be the same.

Add a new city to the city table, for example.

```
INSERT INTO City( ID, Name, CountryCode, District, Population )
VALUES( 8000, 'Gananoque', 'CAN', 'Ontario', 50000 );
```

Now run each of the two VIEWs again.
First from the dynamic view
`SELECT * FROM Canada_V;`
and then from the materialized view
`SELECT * FROM Canada_MV;`

The dynamic view will show the updated result from the city table, the materialized view will *not* show the newly added row.

A materialized view can be refreshed using the following syntax:

```
REFRESH MATERIALIZED VIEW Canada_MV;
```

Now run each of the two VIEWs once again (separately), after refreshing the materialized view the result is recent and it now matches the most recent data in the table.

```
SELECT * FROM Canada_V;
SELECT * FROM Canada_MV;
```

Convince yourself how each of the two view differ.

As an additional exercise, delete the newly added row from the city table using:

```
DELETE FROM City WHERE ID = 8000;
```

Observe the result set from the two views

```
SELECT * FROM Canada_V;
SELECT * FROM Canada_MV;
```

Once again, observe that the materialized view does not show the updated result set. What would you do to update the materialized view?

A materialized view has its advantages. Often the most recent data is not required, for example data from the previous day is acceptable. Queries that are demanding on the computers resources, CPU and storage, are candidates for materialized views. Users will be able to access the data without overloading the system. Periodically the materialized view is refreshed to update the data. Usually, developers and end users will work on views instead of tables. A data administrator will create views from subsets of tables filtering certain rows, columns or both rows and columns, that have sensitive data. Users are granted permission to use views but not tables. This method allows access control.

Review Questions on VIEW

1. What are the two types of views?

2. Write two advantages of a view.

6.2 GROUP BY

`GROUP BY` is used with aggregate functions, it provides a summary of rows. Examples of aggregate functions are `AVG`, `SUM` and `COUNT`.

Use `world` database to test the queries in this section.

```
SELECT SUM( Population )
FROM City;
```

Figure 6.1: Query to add population of all cities

The above query will give us one number which is the total of all cities in the City table. If we want to breakup this sum by each country, the query needs to be extended.

The following SQL statement provides the sum of the number of people living in cities in each of the countries:

```
SELECT CountryCode, SUM( Population )
FROM City
GROUP BY CountryCode;
```

Figure 6.2: Using `GROUP BY` clause

Run the query in figure 6.2 and observe the results.

Now, remove the `GROUP BY` clause and run the query again. Does the query work?

```
SELECT CountryCode, SUM( Population )
FROM City;
```

Figure 6.3: Using an Aggregate function without a `GROUP BY` clause

The query in figure 6.3 will give an error, `CountryCode` must be part of the `GROUP BY` clause. *Aside:* Versions of `MySQL` will run the query without error but will give an incorrect result. As an SQL user, always anticipate the expected result and then confirm with the result provided by the query.

6.2.1 GROUP BY and HAVING

`HAVING` clause qualifies groups. Expanding the query in figure 6.2, to limit countries with the sum of city populations less than 200000.

```
SELECT CountryCode, SUM( Population ) AS "Sum Population"
FROM city
GROUP BY CountryCode
HAVING SUM( Population ) < 200000;
```

Figure 6.4: `GROUP BY` qualified by `HAVING`

In figure 6.4 the `HAVING` clause qualifies the countries. Observe the same aggregate `SUM( Population )` in the expression list of the `SELECT` caluse and `HAVING` clause. `WHERE` clause does not allow aggregates, it qualifies rows in a table.

Caution: The following query although syntactically correct gives incorrect results. In the processing order in figure 2.1 on page 18 `HAVING` clause is processed before the `SELECT` clause. An alias cannot be used in the `HAVING` clause.

```
SELECT CountryCode, SUM( population ) as 'Sum Population'
FROM City
GROUP BY CountryCode
HAVING 'Sum Population' < 200000;
```

Figure 6.5: Syntactically correct query with incorrect results

Code in Figure 6.6 will list Countries with more than five cities.
Aside: Notice the alias after `AS` has the the name is double quotes

Exercise The code does not list the name of the country, just the `CountryCode` - modify the query to list the name of the country with `CountryCode`.

```
SELECT CountryCode, COUNT( CountryCode )  AS "Number of Cities"
FROM City
GROUP BY CountryCode
HAVING COUNT( CountryCode ) >= 5;
```

Figure 6.6: Using `GROUP BY` to list Countries With More Than 5 Cities

Exercise What is the result of the code in figure 6.7

```
SELECT CountryCode, SUM( Population ) AS SUM
FROM City
GROUP BY CountryCode
ORDER BY SUM DESC;
```

Figure 6.7: GROUP BY - Exercise

6.2.2 Examples of Some Invalid Queries

1. Aggregate functions cannot be used in a `WHERE` clause. Example,
   ```
   SELECT Name, Population
   FROM City
   WHERE Population = MAX( Population );
   ```

2. Subqueries cannot be used in aggregate functions. Example,
   ```
   SELECT AVG( SELECT Population FROM Country )...
   ```

3. Aggregate functions cannot be nested. Subqueries must be used to get the desired result. *Exception:* Oracle allows nested aggregate functions.
   Example,
   ```
   SELECT AVG( MAX( Population) )
   ```

6.2.3 Review Questions - `GROUP BY`

1. Which of the following can produce scalar and vector aggregates?
   (a) `ORDER BY`
   (b) `HAVING`
   (c) `GROUP BY`

2. An aggregate function, for example `MAX`, `SUM`, can be used in a `WHERE` clause.
   (a) True
   (b) False

3. An aggregate function, and a non aggregate function can be used in a `SELECT` clause only if a `GROUP BY` clause is used for the scalar value.
   (a) True
   (b) False

4. Aggregate functions cannot be nested, for example, `SELECT AVG( SUM( population ) )` is an invalid statement in SQL.
   (a) True
   (b) False

5. Study the following code and identify one correct statement.

```
SELECT Name, SurfaceArea
FROM Country
WHERE SurfaceArea = MAX( SurfaceArea );
```

   (a) The statement is correct, it will run.

   (b) The statement is incorrect, SurfaceArea cannot be repeated in the `SELECT` and `WHERE` clause.

   (c) The statement will not run, aggregate functions are not allowed in `WHERE` clause.

6.3 Exercise 7 - GROUP BY & HAVING

Objective

1. Differentiate between scalar and aggregate values.
2. Apply aggregate functions to a *all* rows in a table.
3. Apply aggregate functions to a set of rows using `WHERE` and `GROUP BY`.
4. Use `HAVING` clause to filter rows in conjunction with `GROUP BY`.
5. Write alternate `SQL` code to nested aggregate expressions.
6. The effect of `NULL` values on aggregate functions.
7. Identify limitations on use of aggregate functions in `WHERE` and `SELECT` clauses.

Reference

1. Section 6.2 page 78
2. Reference [4], Chapter 6, Page 169 ff. Chapter 6 from Fehily's book provides good reference material, and highlights the effect of `COALESCE()` on `NULL` values, and the use of nested aggregate functions in Oracle.

Submission

Complete the queries and the quiz

Background

Examples provided use the `PART_EXT_T` table, this table has additional data from the `Part_T` table.

Requirements

Create a database in postgres, call it `Treat`. Run the following four files, `Treat_DDL.sql`, `Treat_DML.sql`, `Part_EXT-DDL.sql`, `Part_EXT-DML.sql`. You now have two tables, `Part_EXT_T` and `Treat_T`. The example `SQL` queries can be used on `Part_Ext_T` table.

Type in each query and verify the results. Then write your own query using the `Treat_T` table. Note the queries and its results, they are required to answer the quiz questions. Read the section specified in the Reference above.

Procedure

Query 1. Listing the most expensive `Part` The following query will list each Part that is the highest cost, each type of part is listed. Type the query below and verify the result.

```
SELECT Part, MAX( Cost )
FROM Part_Ext_T
GROUP BY Part;
```

Aside: The following query will not work. Aggregates are not allowed in `WHERE` clause.

```
-- incorrect query
SELECT Part, Cost
FROM Part_Ext_T
WHERE Part = MAX( Cost );
```

Use the `Treat_T` table to write a query that will list the chocolate with the highest cost and its country of origin. Your result should be similar to the data in the table below.

Origin	**Highest Cost**
CHE	2.3
ITA	2.2
MEX	14.99

Table 6.1: Highest Cost of Item from Each Country

Query 2. List the number of Parts in each type of Material The following query lists the number of Parts in each size category.

```
SELECT Size, COUNT( SIZE )
FROM Part_Ext_T
GROUP BY Size;
```

Use the `Treat_T` table to write a query that will list the number of products from each country. Your result should be similar to the data in the table below.

Origin	**Number of Products**
CHE	3
ITA	1
MEX	4

Table 6.2: Count the Number of Products from Country Origin

Query 3. List the number of Parts in each type of `Material` with a condition. Modify Query 1 to list all Parts with Maximum cost greater than 2.4. Here is the query.

```
SELECT Material, MAX( Cost ),
FROM Part_Ext_T
GROUP BY Material
HAVING MAX( COST ) > 2.4;
```

Modify the above query to list the average cost of Parts in each Material, with the average cost greater than `1.32`. Your query should be similar to the data shown in table 6.3. *Tip:* Use `ROUND( expr, 2 )` and `AS` for alias.

Material	**Average Cost**
Steel	1.42
Aluminium	1.47

Table 6.3: Count the Number of Products from Country Origin

Use the `Treat_T` table to write a query to list the country and number of products, list only those countries which have the number of products greater than 1. Use aliases fro column title. Your query should be similar to data shown in table 6.4.

Country of Origin	**Number of Products**
CHE	3
MEX	4

Table 6.4: Number of Products from Country of Origin, with condition

6.4 Lab 7 - Retrieve Data from world database.

Objective

1. Retrieve Data from world database using sub-queries.
2. Specify range using BETWEEN
3. Set membership using IN
4. Use LIMIT and OFFSET
5. Use GROUP BY and HAVING
6. Simple joins – implicit and explicit

Reference

1. References in Lab02
2. Section 2.6 JOINs, PostgreSQL Documentation postgresql-9.5-US.pdf

Submission Complete the online quiz.

Procedure Type and run the sample queries, then modify them to get the required results. Do not cut and paste the queries from the PDF file into pgAdminIII.

Query 1. Query information schema There are several ways to display metadata. pgAdminIII provides a graphical interface. A command line version is shown below. Open an SQL editor for the world database and type in the following SQL statement. *Note:* The tablenames must be be in lowercase.

```
SELECT * FROM information_schema.columns WHERE table_name = 'city';
```

and

```
SELECT * FROM information_schema.columns WHERE table_name = 'country';
```

In each case columns_name is listed.

Query 2. Perform Implicit Join The statement

```
SELECT countrycode, name, population FROM city;
```

will list all countrycode, name and the city's population. The query can be made easier to read by replacing the country code with the name of the country. The name of the country is stored in the Country table. Write a SQL statememt that will join the two tables to display the country name. Your result should be similar to the sample shown below. Use an alias to change the name of each of the columns shown.

Hint: Use WHERE City.CountryCode = Country.Code to join City and Country tables.

```
Country       City        Population
-------       ----        ----------
Canada        Calgary     768082
Canada        Toronto     688275
Canada        North York  622632
```

Query 3. Explicit JOIN Use a `JOIN` clause to get the same result as in the above query.

Query 4. Using `BETWEEN` clause. Type in the statement below and observe the results. It lists all cities with a population between 5000 and 6000. The `BETWEEN` clause filters rows based on a range of values, it includes the boundary values. The `BETWEEN` clause can take Numbers, text and dates as data types. `BETWEEN` clause can be used with `IN` clause. Its inverse is performed by using `NOT BETWEEN`.

```
SELECT CountryCode AS "Country Code", Name AS "City", Population AS "City Population" FROM City
WHERE Population BETWEEN 5000 AND 6000;
```

List cities in Canada that have their population greater than or equal to 200000 and less than or equal to 300000. Your result should be similar to the sample data shown below.

```
Country       City         Population
-------       ----         ----------
Cananda       Brampton     296711
Canada        Windsor      207588
```

Query 5. Row and Array comparison Refer Section 9.23 in PostgreSQL documentation. The `SQL` statement below lists all cities in Cameroon, Cuba and Jamaica. Type it and check the results.

```
SELECT countrycode, city.name, population
FROM city
WHERE countrycode IN( 'CMR', 'CUB', 'JAM' );
```

The statement below will add the population of each of the cities.

SELECT CountryCode AS "Country Code", SUM(Population) FROM City WHERE CountryCode IN('CMR', 'CUB', 'JAM') GROUP BY CountryCode;

Modify the above statement, join the City and Country table to display the country name instead of the country code. List the sum of all population in the cities in Ecuador, France and Haiti their country codes are `ECU`, `HTI`, `FRA`.

Your result should be similar to the sample shown below, use alias for `Country` and `People Living in Cities`

```
Country         People Living in Cities
-------         -----------------------
Ecuador         5744142
France          9244494
```

```
Haiti           1517338
```

Query 6. Creating a `VIEW` Use a sub-query to list country, country population for countries that have their population less than the average population of all countries. Create a view for the above query. Call the view `Avg_V`. Test your view. Use this view to list the country in reverse alphabetical order. Refer Section 3.2 in postgreSQL documentation.

Query 7. Which country does the city **Zukovski** belong to? Write a SQL statement to display only one country name.

Query 8. Using `LIMIT` Refer section 7.6 in PostgreSQL documentation.

```
SELECT * FROM City;
```

Modify the above `SQL` statement to display the first 10 rows. Then use the `OFFSET` keyword, to display the next 10 rows, i.e. rows 11 to 20.

Query 9. `GROUP BY` and `HAVING` List the countries that have more than 200 cities in the database, use `GROUP BY` and `HAVING`

Your result should have the following countries.

```
BRA    250
CHN    363
JPN    248
IND    341
USA    274
```

Modify the statement to replace the Country Code with Country Name. Your result should be similar to

```
Brazil         250
India          341
Japan          248
United States  274
China          363
```

Query 10. List the names of countries where `Spanish` is spoken. Use an alias for the country name to appear as `Country`, sort the country name in ascending order.

7

JOIN

7.1 Introduction

This chapter addresses multi table queries, querying multiple tables include `joins` and `sub-queries`. Two other topics critical to web based development are embedded SQL and dynamic SQL. Sub-query is a versatile query technique in SQL, it is used to return values to another query.

7.2 Objective

- Build syntax for `JOIN`s - equijoin, natural join, outer join, inner join, self join
- Write sub-queries. Differentiate and compare sub-query with join
- Build equivalent queries using `JOIN` and sub-query
- Compare correlated and non-correlated sub-queries
- Combine queries using `UNION`
- `CREATE` and `DROP` Indexes
- Use conditional expressions

7.3 Working with more than one table

Table 7.1 lists types of joins with a brief explanation. Note, `CROSS JOIN` and `UNION` are not join operations.

Join	Result
NATURAL JOIN	All requested results, values of common columns returned only once. Duplicate column eliminated. If there are no common column names, `NATURAL JOIN` results into a `CARTESIAN PRODUCT` (`CROSS JOIN`)
INNER JOIN	Requires each record in two tables to have matching records. Used in many situations. May not be suitable in some conditions.
EQUI JOIN	Uses only field values in tables that are equal. Returns all requested results including values of common columns.
OUTER JOIN	Returns all values from one table even if match not found.
LEFT OUTER JOIN	Joins rows based on matched values. The result includes unmatched rows from the table on the **left** of the `JOIN` clause.
RIGHT OUTER JOIN	Joins rows based on matched values. The result includes unmatched rows from the table on the **right** of the `JOIN` clause.
FULL OUTER JOIN	Joins rows based on matched values and returns rows from both tables. Not supported in MySQL.
SELF JOIN	Joins the table to itself.

Table 7.1: Joins

Operation	Result
CROSS JOIN	Cartesian product of tables. Combines each row from the first table with each row from the second table.
UNION	Returns a table that includes all data from each table. Both tables (or views) must have the same number of columns and the data types of both columns must be same. Strictly, not a join.

Table 7.2: Other Operations

7.4 JOIN

Definition: A `JOIN` clause combines one or more tables based on a common data value. The tables are from the same database.

Data is stored in individual tables; it is the *relationship* between the tables that make the data meaningful; `JOIN`s perform that meaningful relationship. [4, Fehily, Page 193]. `SQL`'s strength is in `JOIN` operations.

NATURAL JOIN

Result of a `NATURAL JOIN` is a set of records from table 1 and table 2 that are equal in their common attribute.

A NATURAL JOIN is one of the eight operators. The following example is adopted from back inside cover of [1]. Given minimal metadata, perform a `NATURAL JOIN`.

a1	b1
a2	b1
a3	b2

Table 7.3: Table 1

b1	c1
b2	c2
b3	c3

Table 7.4: Table 2

Exercise - Natural Join

Perform a `NATURAL JOIN` on the following tables.

p1	x2
p2	x2
p3	x4

Table 7.5: Table A

x2	q1
x3	q2
x4	q3

Table 7.6: Table B

Using `JOIN`'s

Consider the following three tables `Team`, `Player` and `PlayerTeam`. Table `Team` has five Teams, `TeamID` is the prime-key. Table `Player` has six players. Players are assigned to teams, this is shown in table `PlayerTeam`. The ERD is shown in figure 7.1.

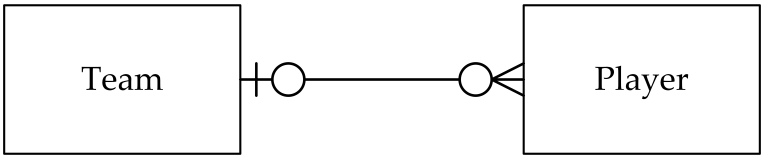


Figure 7.1: Logical ERD, Player Team

TeamID	Team
21	Woqag
22	Zumey
23	Gavop
24	Turoz
25	Nibeg

Table 7.7: Team

PlayerID	PlayerName
445	Bix
446	Lay
447	Vow
448	Cox
449	Sal
450	Kar

Table 7.8: Player

PlayerID	TeamID
445	21
446	23
447	23
448	24
449	24

Table 7.9: PlayerTeam

Player `Kar` has not been assigned a Team; Team `25,Nibeg` and Team `22,Zumey` has no players assigned.

Exercise

Use `PlayerID` and `TeamID` from Table 7.9 to write, in tabular form, Player Name and Team Name.

INNER JOIN

`INNER JOIN` is the most common type of join - although it is not suitable in all situations. It requires each record in two tables to have matching records. Here is an `INNER JOIN` that gives the lists Players and Team assigned to the player. The SQL statement in figure 7.2 joins three tables:

`INNER JOIN` has predictable results when referential integrity is enforced. Referential Integrity needs to be preserved if the results of `INNER JOIN` have to match the expected results. In sample tables, Player **Kar** does not appear in the result - he/she is not associated with a team. Also, team `25, Nibeg`, and team `22, Zumey` does not appear in the results.

```
SELECT PlayerName, TeamName
FROM Player
INNER JOIN
PlayerTeam USING( PlayerID )
INNER JOIN
Team USING( TeamID );
```

Figure 7.2: Inner Join

An `INNER JOIN` is suited for tables that do not have `NULL` values. Data with `NULL` values are omitted without error or warnings. `NULL` values in one table **do not** match any values in another table, not even `NULL` values.

LEFT OUTER JOIN

joins two tables; the result matches attributes from both tables *and* it includes unmatched rows from the table on the **left** of the `JOIN` clause.

```
-- LEFT OUTER JOIN
-- Showing all players and their teams.
-- Showing player Kar does not belong to any team
SELECT Player.PlayerID, PlayerName, PlayerTeam.TeamID
FROM Player
LEFT JOIN PlayerTeam
ON PlayerTeam.PlayerID = Player.PlayerID;
```

Figure 7.3: Left Outer Join

```
-- LEFT OUTER JOIN
-- Player Kar does not belong to any team
SELECT Player.PlayerID, PlayerName, PlayerTeam.TeamID
FROM Player
LEFT JOIN PlayerTeam
ON PlayerTeam.PlayerID = Player.PlayerID WHERE PlayerTeam.PlayerID IS NULL;
```

Figure 7.4: Left Outer Join using `IS NULL`

```
-- LEFT OUTER JOIN
-- Showing team Zumey and team Nibeg do not have any players
SELECT Team.TeamID, TeamName, PlayerTeam.PlayerID
FROM Team
LEFT JOIN PlayerTeam
ON PlayerTeam.TeamID = Team.TeamID;
```

Figure 7.5: Left Outer Join

```
-- NOT IN clause
-- Note the use of a sub-query -- Showing team Zumey and team Nibeg do not have any players
SELECT Player.PlayerID, PlayerName
FROM Player
WHERE Player.PlayerID
NOT IN( SELECT PlayerID FROM PlayerTeam);
```

Figure 7.6: `NOT IN` clause

Aside: Equivalent SQL statement using `WHERE` clause to get the same result from query in figure 7.5. Type and run this query, you will notice the order of rows is different. Using and `ORDER BY` clause will not work directly, SQL needs both queries in the `UNION` clause to be ordered. This solution is a hack, it forces a `NULL` in the second statement.

```
SELECT Team.TeamID, TeamName, PlayerTeam.PlayerID
FROM
Team, PlayerTeam
WHERE Team.TeamID = PlayerTeam.TeamID
UNION
SELECT TeamID, TeamName, NULL
FROM Team
WHERE
TeamID NOT IN( SELECT TeamID FROM PlayerTeam );
```

RIGHT OUTER JOIN

joins two tables; the result matches attributes from both tables *and* it includes unmatched rows from the table that is on the **right** of the `JOIN` clause.

```
-- RIGHT OUTER JOIN
-- Showing team Zumey and team Nibeg do not belong to any players
SELECT Team.TeamID, TeamName, PlayerTeam.PlayerID
FROM PlayerTeam RIGHT JOIN
Team ON PlayerTeam.TeamID = Team.TeamID;
```

Figure 7.7: Right Outer Join

Tips on using `OUTER JOIN`

1. Equate data items that have common values. For example, referring to 7.7 using the phrase
`ON PlayerTeam.TeamID = Team.TeamName`
will not yield any results (or incorrect results.) `TeamID` and `TeamName` are different data items. *Note:* The data items must be same, not the field names. `City.CountryCode = Country.Code` will give results - `CountryCode` in the `city` table has matching values in the `country` table in the field name `Code`.

2. Make sure the list of attributes in the `SELECT` clause have meaningful attributes. The `JOIN` will give you the results but if the field names are not correctly chosen you will get invalid rows.

FULL OUTER JOIN

Joins rows based on matched values and returns rows from both tables. Not supported in MySQL; not often used. Same results are obtained by using `RIGHT OUTER JOIN`, `UNION` with `LEFT OUTER JOIN`.

SELF JOIN

A table joined with itself is a `SELF JOIN`. It happens in a unary relationship, when a table with a foreign key references the primary key in the same table. Think of a join on two tables which are same; each row of one table is combined with each row of the other table. There is no explicit statement for a `SELF JOIN`. Examples of `SELF JOIN`: Consider an employee table, with `EmployeeID`, `Name` and `ManagerID`. Most employees will have a manager whose ID will be associated with the employee record.
A drug will have contraindications with another drug. `DrugID`, `Name` and `ContradicationID` will be in the same record.

Exercise: Run the script `SELF JOIN Demo.sql` and `SELF JOIN Query.sql`

CROSS JOIN

A cross join does not apply any predicate or filter, it has limited practical use in data processing. Be careful when using a cross join, the result is often not what you expect. Mathematically, a `CROSS JOIN` is a `product`, (specifically a cartesian product) one among the *original eight* operators. A `NATURAL JOIN`, discussed later, is a join operator. Always examine the result of a join operation and verify with your expected outcome. An incorrectly specified `WHERE` clause results in a cross join

Aside: A predicate is an operator, or a function, that returns a `TRUE` or `FALSE`.

A cross join can be run in two ways:
`SELECT * FROM <tableA> CROSS JOIN <tableB>;`
or simply
`SELECT * FROM <tableA>, <tableB>;`

`SELECT * FROM Team, Player;` results in 30 rows. In this case it is not a useful result.

Question: Think of a use of a cartesian product, i.e. `CROSS JOIN`

Exercise: Run the script `deck.sql`.
Aside: Note the method of inserting rows using a single `INSERT` statement

Final Note

In MySQL, `JOIN` and `INNER JOIN` are syntactic equivalents, they can replace each other. In standard SQL, they are not equivalent. `INNER JOIN` is used with an `ON clause`

7.5 Review Questions

1. A join operation:
   (a) is used to combine indexing operations
   (b) joins two tables with a common attribute to form a single table or view, the common attribute must be a prime key in both tables
   (c) joins two tables with a common attribute to be combined into a single table or view
   (d) joins two disparate tables to be combined into a single table or view
2. A join operation is performed on two tables. The common field that is used for the join operation has a few `NULL` values in each of the two tables. The join operation will:
   (a) match `NULL` values from the first table but not from the second table
   (b) match `NULL` values from the second table but not from the first table
   (c) match `NULL` values from both tables, since it is a join operation
   (d) not match `NULL` records from any of the two tables
3. Identify the clause that is used to combine output from multiple queries into a single result table.
   (a) `COLLATE`
   (b) `INTERSECT`
   (c) `DIVIDE`
   (d) `UNION`
   (e) `SELF JOIN`

4. `SELECT * FROM Student, Course;` will result in a
   (a) `NATURAL JOIN`
   (b) `SELF JOIN`
   (c) `OUTER JOIN`
   (d) `CROSS JOIN`
   (e) `INNER JOIN`
5. A join in which rows that do not have matching values in common columns are still included in the result table is called a/an:
   (a) outer join
   (b) union join
   (c) equi-join
   (d) natural join
   (e) inner join
6. An operation to join a table to itself is called a/an:
   (a) self join
   (b) inner join
   (c) natural join
   (d) outer join
   (e) equi join
7. An alternative term used for `CROSS JOIN` is
   (a) `NATURAL JOIN`
   (b) `SELF JOIN`
   (c) `OUTER JOIN`
   (d) `CARTESIAN PRODUCT`
   (e) `INNER JOIN`

Additional Exercises

1. Is the result of the `LEFT OUTER JOIN` the same as the table on the *left* of the `JOIN` clause? Justify your answer.

2. Perform a `NATURAL JOIN` on the tables below.

omlette	salt
omlette	pepper
sandwich	ketchup
poutine	vinegar
poutine	salt

Table 7.10: Food A

salt	beans
ketchup	burger
mustard	burger
sugar	cake

Table 7.11: Food B

3. Perform a `NATURAL JOIN` on the tables below.

omlette	salt
omlette	pepper
sandwich	ketchup
poutine	`NULL`
poutine	salt

Table 7.12: Food C

salt	beans
ketchup	burger
`NULL`	burger
sugar	cake

Table 7.13: Food D

7.6 Additional Example on JOIN Operations

Abstract This exercise illustrates JOIN operations between two tables, music players and the instruments they play in Table 7.14 and the Instruments and the category in table 7.15. Various JOIN operations are shown below. In the data shown below, a player plays only one instrument.

Player	Instrument
Glen Gould	Piano
Young Joo Chang	Violin
Guan Pinglu	Guqin
Hari P.C.	Bansuri
Seamus Egan	Bodhran
Angela Hewitt	Piano
Alice Giles	Harp
Elina Karokhina	Balalaika

Table 7.14: Player_Instrument_T

Instrument	Category
Piano	String-Hybrid
Violin	String
Erhu	String
Bansuri	Wind
Guqin	String
Bagpipe	Wind
Bodhran	Percussion
Harp	String
Balalaika	String

Table 7.15: Instrument_T

7.6.1 DDL & DML statements

```
-- Filename: Player-Instrument-DDL-DML.sql
-- Author: S I'Aret
-- Created: 3 Feb 2019
-- Note: This is a shorter way to insert data using a single INSERT statement
-- Caution: It is harder to debug and detect errors using this method.
-- It can be used for very small tables, such as this
-- Do not use this method for your assginments.
-- The syntax is terse, hence its usage here.

DROP TABLE IF EXISTS Player_Instrument_T;
DROP TABLE IF EXISTS Instrument_T;

CREATE TABLE IF NOT EXISTS Player_Instrument_T(
   Player       VARCHAR( 25 ),
   Instrument   VARCHAR( 20 )
);
```

```
CREATE TABLE IF NOT EXISTS Instrument_T(
   Instrument   VARCHAR( 20 ),
   Category     VARCHAR( 15 )
);


INSERT INTO Player_Instrument_T( Player, Instrument ) VALUES
( 'Glenn Gould', 'Piano' ), ( 'Angela Hewitt', 'Piano' ),
( 'Young Joo Chang', 'Violin' ), ( 'Guan Pinglu', 'Guqin' ),
( 'Hari P. C.', 'Bansuri' ), ( 'Seamus Egan', 'Bodhran' ),
( 'Alice Giles', 'Harp' ), ( 'Elina Karokhina', 'Balalaika' );


INSERT INTO Instrument_T( Instrument, Category ) VALUES
( 'Piano', 'String-Hybrid' ), ( 'Violin', 'String' ),
( 'Erhu', 'String' ), ( 'Guqin', 'String' ),
( 'Bagpipe', 'Wind' ), ( 'Bodhran', 'Percussion' ),
( 'Bansuri', 'Wind' ), ( 'Harp', 'String' ),
( 'Balalaika', 'String' );


-- eof: Player-Instrument-DDL-DML.sql

INSERT INTO Player_Instrument_T( Player, Instrument ) VALUES
( 'Glenn Gould', 'Piano'), ( 'Angela Hewitt', 'Piano'),
( 'Young Joo Chang', 'Violin'), ( 'Guan Pinglu', 'Guqin'),
( 'Hari P. C.', 'Bansuri'), ( 'Seamus Egan', 'Bodhran'),
( 'Alice Giles', 'Harp'), ( 'Elina Karokhina', 'Balalaika');

INSERT INTO Instrument_T( Instrument, Category ) VALUES
( 'Piano', 'String-Hybrid'), ( 'Violin', 'String'),
( 'Erhu', 'String'), ( 'Guqin', 'String'),
( 'Bagpipe', 'Wind'), ( 'Bodhran', 'Percussion'),
( 'Bansuri', 'Wind'), ( 'Harp', 'String'),
( 'Balaika', 'String');

-- eof:  Player-Instrument-DDL-DML.sql
```

7.6.2 Queries

```
SELECT * FROM Player_Instrument_T;
SELECT * FROM Instrument_T;

-- All Players with their instruments.
-- JOIN, USING
-- All three attributes are selected from their tables without ambiguity
SELECT Player, Instrument, Category
FROM Player_Instrument_T
JOIN Instrument_T USING( Instrument );


-- All Players with their instruments.
-- JOIN, ON
-- Column Instrument must be explicitly defined as Player_Instrument_T.Instrument
-- result is the same as JOIN, USING
SELECT Player, Player_Instrument_T.Instrument, Category
FROM Player_Instrument_T
JOIN Instrument_T ON Player_Instrument_T.Instrument = Instrument_T.Instrument;


-- All Players with their instruments.
-- NATURAL JOIN
SELECT Player, Instrument, Category
FROM Player_Instrument_T
NATURAL JOIN Instrument_T;


-- All Players with their instruments.
-- INNER JOIN
SELECT Player, Instrument, Category
FROM Player_Instrument_T
INNER JOIN Instrument_T USING( Instrument );



-- implicit JOIN
-- This query does not use the JOIN keyword
-- It gives the same result with a WHERE keyword
SELECT Instrument_T.Instrument, Player, Category
FROM Instrument_T, Player_Instrument_T
WHERE Instrument_T.Instrument = Player_Instrument_T.Instrument;
```

```
-- Instruments with no players
-- Query using a LEFT JOIN
SELECT Player, Instrument, Category
FROM Instrument_T
LEFT JOIN Player_Instrument_T USING( Instrument )
WHERE Player IS NULL;


-- unmatched values in Instrument_T with Player_Instrument_T
-- Instruments without Players
SELECT Instrument_T.Instrument, Category
FROM Instrument_T
WHERE Instrument_T.Instrument NOT IN( SELECT Instrument FROM Player_Instrument_T );


-- Instruments with no players, and players with an instrument
SELECT Player, Instrument, Category
FROM Instrument_T
LEFT JOIN Player_Instrument_T USING( Instrument );


SELECT Player, Instrument, Category
FROM Instrument_T
FULL OUTER JOIN Player_Instrument_T USING( Instrument )
WHERE Player IS NULL;
```

7.7 JOIN statements: An Analysis

JOIN order, Implicit and Explicit JOINs are discussed in this section.

7.7.1 JOIN Order

The Order of the tables in a JOIN statement will make a difference, especially when `GROUP BY` is used. Consider the two queries from Author-Publisher Database.

```
SELECT BookID, BookTitle, COUNT( InventoryID )
FROM Book_T
JOIN Inventory_T USING( BookID )
GROUP BY BookID;
```

Figure 7.8: Explicit JOIN - Parent-Child

```
SELECT BookID, BookTitle, COUNT( InventoryID )
FROM Inventory_T
JOIN Book_T USING( BookID )
GROUP BY BookID;
```

Figure 7.9: Explicit JOIN - Child-Parent

Query is figure 7.8 runs without errors, but the second query in 7.9 does not run.

7.7.2 Implicit and Explicit `JOIN` statements

Consider the following SQL statement. The WHERE clause is parsed immediately after `FROM`, SQL now has all the attributes required to perform the `JOIN` and `GROUP BY` operations. It is preferable to use an explicit JOIN, it is predictable when handling NULL values.

```
SELECT Book_T.BookID, BookTitle, COUNT( InventoryID )
FROM Book_T, Inventory_T
WHERE Inventory_T.BookID = Book_T.BookID
GROUP BY Book_T.BookID;
```

Figure 7.10: Implicit `JOIN`

7.8 Exercise 8 - SQL `JOIN` statements

Objective:

1. Write a query using `INNER JOIN`
2. Write a query using `LEFT OUTER JOIN` and `RIGHT OUTER JOIN`

Submission: Complete the online quiz on `JOIN`s

Background: Read chapter 7 *JOIN* beginning at page 89 of these notes. Study the contents of the three tables, table 7.7 on page 92, table 7.8 on page 92 and table 7.9 on page 93. Study the `PRIMARY KEY` and `FOREIGN KEY` constraints.

Procedure: Start `postgreSQL`, run the file `Player-Team-DDL-DML.sql`, it will create three tables, `Team`, `Player` and `PlayerTeam` and add data to each of the three tables. Study the contents of the `.sql`. Study the logical ER diagram.

Open a new `.sql` file in `pgAdminIII` editor. *Type* the query 7.2 listed on page 94, do not cut and paste the queries from the PDF file. Observe the results.

Type and run queries 7.3, 7.4, 7.5 and 7.7 on page 94 onwards.
Exercise - `INNER JOIN`: Type the following code and observe the results.

```
SELECT Country.Code, Country.Name, City.Name
FROM Country, City
WHERE Country.Code = City.CountryCode;
```

Use an `INNER JOIN` to write a query that will give the same result as the above query's result. Refer to query 7.2 on page 94 for an example. Note: Query 7.2 joins three tables, in this exercise you are required to join two tables, `Country` and `City`.

Exercise - `LEFT JOIN`: `world` database has three tables, `Country`, `City` and `CountryLanguage`. In this exercise use two tables `Country` and `City`. Use query 7.4 as an example, write a query using `LEFT JOIN` to list all countries that do not have any cities in the city table. Your result should be similar to the figure 7.16.

Country Code	Country
ATA	Antarctica
BVT	Bouvet Island
IOT	British Indian Ocean Territory
SGS	South Georgia and the South Sandwich Islands
HMD	Heard Island and McDonald Islands
ATF	French Southern territories
UMI	United States Minor Outlying Islands

Table 7.16: List of Countries with no cities

Exercise - `RIGHT JOIN`: Type the following code and observe the results. It lists six countries from Country Table which do not have any language listed.

```
SELECT Country.Code, Country.Name
FROM Country
WHERE
Country.Code NOT IN( SELECT CountryCode FROM CountryLanguage );
```

Use `RIGHT JOIN` to join two tables `Country` and `CountryLanguage` to give the same results.

Additional Exercises: For each of the queries in queries 7.2, 7.3, 7.5 and 7.7 from page 94 ff, write a query using a `WHERE` clause that will give the same results.

7.9 Lab 8 - SQL `SELF JOIN`

Reference

1. Scripts provided

Objective

1. Run DDL & DML scripts and study the output

Submission

Show your work to the lab instructor during lab hours. Upload to LMS the SQL statements in selected questions.

Background

Review notes on `CARTESIAN PRODUCT`, correlated subquery and non-correlted subquery

Requirements

1. Run the script `Deck.sql`. Observe the output of a `CARTESIAN PRODUCT`, also known as `CROSS JOIN`.
2. Run the scripts `SelfJoinDDL.sql`, `SelfJoinDML.sql` and `Self Join Query.sql`. Observe how the manager's data, i.e. `ManagerID` needs to be inserted first and then the employee records. Alternatively `ManagerID` can be updated later. Observe the two ways in which a constraint can be implemented. First at the time of creating the table or by using `ALTER TABLE` clause. Draw the hierarchy of the employees and his/her manager. Insert the following Employee Records. Create EmployeeID's by using the first letter of the employees name with a next available digit for that letter.
   (a) `Corred` reports to `Mooq`
   (b) `Dijjov` reports to `Corred`
   (c) `Cifzuy` reports to `Corred`
   (d) `Dapdom` reports to `Dijjov`
   (e) `Xigmok` reports to `Dijjov`
   (f) `Luyban` reports to `Cifzuy`
   (g) `Qabrex` reports to `Cifzuy`
3. Run the SQL scripts in folder `Natural Join`. Study syntax for `NATURAL JOIN`, `JOIN`, `INNER JOIN`, `LEFT JOIN` and `RIGHT JOIN`.
4. Run the SQL script `non correlated and correlated sub query.sql`. Read the comments before each query. Write one non correlated and one correlated sub query of your choice.
5. Run the scripts `InvDDLDML.sql`, `InvQuery.sql` and `PVFCQuery.sql` answer question on page 29, Operator Precedence, of these notes.

7.10 Online Quiz

7.10.1 Hybrid Quizzes

No.	Title	Check	Section, Page
1	JOIN-1		
2	JOIN-2		

Table 7.17: List of JOIN Quizzes

7.11 Summary

Several SQL implementations do not differential between `NATURAL JOIN`, `INNER JOIN` and `EQUI JOIN`. A Join operation can be performed using the `WHERE` clause giving the same result set in most cases, in the industry the explicit `JOIN` operator is preferred; the `JOIN` operator make the results predictable with the data has `NULL` values.

8

Database Design

8.1 Objective

- Define a *relation*. Differentiate between a *table* and a *relation*. List properties if a *relation*.
- Define a *prime key*, *foreign key*, *composite key*, *surrogate key*.
- Maintain data integrity using *constraint*s.
- Define partial key dependency; identify and be able to remove it from a given relation.
- Define transitive dependency; identify and be able to remove it from a given relation.
- Define an anomaly with reference to a database.
- Identify the three anomalies in a database.
- Normalize data to third normal form (3NF) from unnormalized data.

8.2 Properties of a Relation

Six properties of a relation are:

1. Each relation in a database has a unique name.
2. An entry at the intersection of each row and column is atomic. Stated differently, each cell has a single value.
3. Each row is unique. Stated differently, no two rows are identical.
4. Each attribute within a relation has a unique name.
5. The sequence of attributes has no consequence.
6. The sequence of rows has no consequence.

Two additional properties are

- Data in a column is the same kind. Columns contain data on attributes in an entity.
- Rows contain data on an entity.

8.3 Terms

Prime Key An attribute or a combination of attributes that uniquely identifies each row in a relation.
Stated alternately,
A primary key is the ***minimum*** number of attributes required to uniquely identify a row in a table.

Composite Key A primary key that consists of more than one attribute.

Foreign Key An attribute, or a set of attributes, in a table that has a corresponding data value in another table. *Referential Integrity* is ensured when data in the second table has a matching value in the first table. Usually, a child table has a foreign key defined, this foreign key has a the same data value in the primary key of a parent table.

Surrogate Key A number assigned to uniquely identify a record. A surrogate key may serve as a prime key. It could be, a serial number with no significance to the data. A combination of letters and numbers such as a drivers licence number in Ontario, a health card number, a social insurance number (SIN) serves as a surrogate key. A key that is independent of user data is immune to changes in users data.

Constraint is a rule on data values, defined by a user, and enforced by the DBMS. A constraint maintains quality of data. Section 8.4 provides two examples of constraints.

Table Data represented in two dimensional format as rows and columns. As a convention, each column represents the same *type* (and same kind) of data.

A **Relation** is a two dimensional table of data that has the prime key defined, i.e. two rows of a relation cannot be identical. *Aside:* A relation is a special case of a table, all relations are tables, not all tables are relations.

Partial Functional Dependency A functional key in which one or more non-key attributes are functionally dependent on part, but not all, of the primary key.

Transitive Dependency In a relation, transitive dependency is present, when a non-key depends on another non-key.

Insertion Anomaly There are two types of Insertion Anomalies. First, data insertion in some attributes forces data into other attributes. Second, if data is not available for all attributes it forces a user to enter `NULL` values.

Deletion Anomaly other relevant data is possibly lost when a data element needs deletion.

Update Anomaly forces an update to all tuples when a change is made to one field in a tuple. Failure to update all tuples puts the database in an inconsistent state.

8.4 Constraint

Refer to `Inventory-DDL.sql` file, two examples of constraints are:

1. Primary key constraint, in the `Customer_T` table
   `CONSTRAINT Customer_PK PRIMARY KEY( Cust_Id )`
   This indicates that `Cust_ID` is a primary key. The DBMS will not allow a user to enter two customers with the same key. `Customer_PK` is a user defined object name. As a convention the suffix `PK_` is an aid to easily identify a prime key constraint.
2. In the `Invoice_T` table there are two constraints, the second constraint is

`CONSTRAINT Cust_ID_FK FOREIGN KEY( Cust_Id ) REFERENCES Customer_T( Cust_ID )`

The object name is `Cust_ID_FK`, it is a `FOREIGN KEY` constraint on `Cust_ID` in the `Invoice_T` table, each invoice will have a Customer ID (`Cust_ID`), the constraint will ensure that each entry in the `Invoice_T` table must (first) have an entry in the `Customer_T` table. It will not allow a user to add or change the customer id in the `Invoice_T` table unless it exists in the `Customer_T` table. Also, in the `Customer_T` table, an entry cannot be deleted if there is a record of the customer in the `Invoice_T` table.

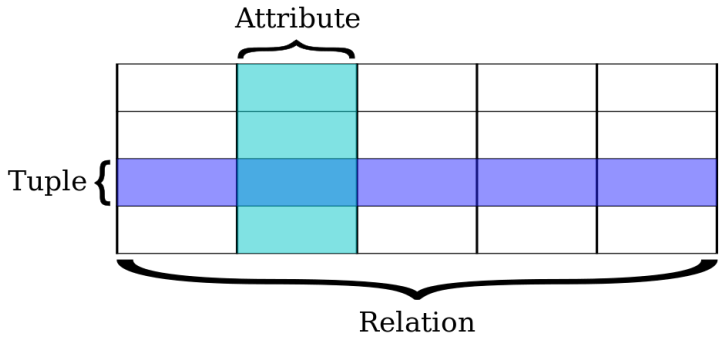


Figure 8.1: A Relation Ref: [6]

8.5 Normalization

Normalization is the process of organizing the fields and tables of a relational database to minimize redundancy. Normalization divides large tables into smaller, less redundant, tables and defines relationships between them. Normalization isolates data such that additions, deletions, and modifications to a field can be made in just one table and propagated through the rest of the database using the defined relationships. [5]

The process of Normalization is reversible. For example, data from 3NF can be denormalized to 2NF, data from 2NF can be denormalized to 1NF. No data is lost in the process; no data is added. A normalized database is free from *insertion*, *deletion* and *update* anomalies.

A relation is in **First Normal Form** (1NF) when the following two conditions are satisfied

1. Repeating groups have been resolved, i.e. there is only one data value in an attribute. A set of multiple values are not permitted.
2. The primary key has been defined

Note: The term *primary key* is singular, even though it can be a combination of more than one attribute.

A relation is in **Second Normal Form** (2NF) when the following two conditions are satisfied

1. The relation is in First Normal Form
2. Every non-key attribute is fully functionally dependent on the primary key. Stated differently, all non-key attributes are fully dependent on the ***entire*** prime key.

A relation is in **Third Normal Form** (3NF) when the following two conditions are satisfied

1. The relation is in Second Normal Form

2. It has no transitive dependencies

Note: A table in 3NF is usually free from the three anomalies.

8.6 Learning Activities

We shall identify anomalies and then normalize the tables.

8.6.1 First Normal Form

Exercise

Identify two disadvantages if data is represented as repeating groups.

Exercise

Study table 8.1. Identify the prime key. Give two reasons for your choice.

TeamID	Team
21	Woqag
23	Gavop
24	Turoz
25	Nibeg

Table 8.1: Team

Exercise

Table 8.2 lists the planets and its satellites. Some planets do not have any satellites, some have only one and others have more than one. Put the data in First Normal Form by resolving repeating groups and identifying the prime-key. Do not insert additional attributes or surrogate keys, such as `ID`. Identify any shortcomings in the tables. State any assumptions you have made.

Planet	Satellite
Mercury	NULL
Venus	NULL
Earth	Moon
Mars	Deimos, Phobos
Jupiter	Ganymede, Callisto, Io, Europa
Saturn	Titan, Enceladus
Uranus	Titania, Oberon, Umbriel, Ariel, Miranda
Neptune	Triton

Table 8.2: Planet-Satellite Table

Exercise

Study table 8.3, its data is taken from the `world` database. Identify the prime key. In what Normal Form is the table? Give reason(s) for your answer.

CountryCode	Country	CountryPopulation	CityID	City	CityPopulation
CAN	Canada	35540419	1814	Winnipeg	618477
CAN	Canada	35540419	1820	London	339917
CAN	Canada	35540419	1811	Calgary	767082
GBR	United Kingdom	63181775	456	London	7285000
GBR	United Kingdom	63181775	462	Manchester	430000
GBR	United Kingdom	63181775	464	Bristol	402000

Table 8.3: Country City Table

8.6.2 Second Normal Form

We study tables with composite keys and normalize the data to Second Normal Form.

Exercise

1. Confirm that table 8.4 is in First Normal Form.
2. Is the primary key a composite key?
3. The table is not in Second Normal Form. Give a reason.
4. Normalize the data to Second Normal Form.

StudentID	**CourseID**	**StudentName**	**CourseName**	**FinalGrade**
704	8110	Suxot	Programming	A+
736	8110	Bimiq	Programming	A-
695	8001	Tigol	Math	B
695	8215	Tigol	Database	A-
798	8215	Jaceq	Database	B+
142	8001	Zarun	Math	A+
798	8001	Jaceq	Math	B+
142	8215	Zarun	Database	A
408	8110	Widaw	Programming	A

Table 8.4: Student-Course Table

Exercise

Study table 8.5 and answer the following questions.
Assume: A driver can drive more than one route.

1. Confirm that the relation is in First Normal Form.
2. Is the primary key a composite key?
3. Give a reason why the table is not in Second Normal Form.
4. What would happen to the data if DriveID: **653**, DriverName: **Binut**, is promoted to a manager and does not drive route **98**?
5. Normalize the data to Second Normal Form.
6. After Normalizing the data is the deletion anomaly resolved?

Route	**DriverID**	**DriverName**
95	485	Vizeq
95	658	Tutem
95	825	Semok
94	754	Fewup
94	658	Tutem
96	412	Sahab
96	825	Semok
97	157	Sonoq
97	570	Ziyir
98	653	Binut
99	773	Gitiv

Table 8.5: Route-Driver Table for OCTranspo

8.6.3 Third Normal Form

Study table 8.3. Will it be in First Normal Form after you have identified the prime key? Is it in Second Normal Form? Indicate why the table is not in Third Normal Form. Normalize the table to Third Normal Form.

8.6.4 Normalization Process

Figure 8.2 illustrates the process of bringing data from UNF to 3NF. The circle indicates a process, the rectangle represents data.

Figure 8.2: Normalization Process

8.6.5 Boyce-Codd Normal Form (BCNF)

BCNF is a more strict form of 3NF. In rare cases a relation in 3NF may show some anomalies, BCNF addresses these rare cases.

8.7 Guidelines on Normalization

Some general recommendations to follow in the normalization process.

General Recommendation

1. Read the Business Rules carefully.
2. Work on only one step at a time.
3. Read the Terms in section 8.3, page 112
4. Start with a clean sheet of paper, draw a logical ER diagram from rules provided.
5. Do not add any data.
6. Do not remove any data.
7. Do not add any attributes.
8. Do not eliminate any attributes.
9. The process should be reversible, i.e. it should be possible for data in 3NF to be brought back to 1NF, using SQL `JOIN` operations.

First Normal Form

1. Write the unnormalized data in a table.
2. Identify the prime key; the table is now a relation.
3. You may have to make simple assumptions. Write your assumptions clearly. Your assumptions should not contradict any business rules given to you.
4. Look for attributes which indicate ID's (Identification) such as CustomerID, `VIN`, `SIN`. These attributes are likely to be prime keys.

Second Normal Form

1. Look for partial key dependencies.
2. How may attributes do you have from the first normal form? A single attribute as a prime key indicates that the table is already in 2NF. Stated differently, if the prime key is not composite, a relation in 1NF is also in 2NF. A single attribute as a prime key cannot have any *partial-functional* dependencies.
3. If you have more than one attribute as a prime key, you need to identify the attribute that depend on *part* of the key.
4. Descriptive attributes *usually* depend on it's corresponding surrogate key. For example, `Student_Name` will depend on `StudentID`, `Course` will depend on `CourseID`. Check other non-keys with the key, they may not be fully dependent even if the name of the attribute suggests it.
5. Form tables in 2NF from 1NF.
6. Your task is not complete, the newly formed tables will still have anomalies.

Third Normal Form

1. Work only on identifying non-keys that are dependent on other non keys
2. Break-up tables from 2NF to smaller tables, identify prime keys.
3. Verify that all dependencies are removed. Refine tables.

8.8 Lookup Table

A design technique that uses a simple table to store an *informal* code and a description. Consider the table `Part_T` table in lab on page 35. The `Material` attribute stores 9 characters for `Aluminium`, 5 characters each for `Brass` and `Steel`. A design decision could replace this column with one character, to store (say), `A`, `B` and `S`, for each of the materials use. An additional table then describes each of this characters. Table 8.6 illustrates such a table. For reporting purposes a simple `JOIN` will translate the material code into its description. It is easy to see the savings in storage space for (say) 100 or 1000 products. Using a lookup table is a design decision that is taken much later in the systems development life cycle. It is unlikely to show in the conceptual or logical design phase, it is also unlikely to show in business rules.

MaterialCode	Material
A	Aluminium
B	Brass
S	Steel

Table 8.6: Lookup Table. `MaterialCode_T`

Aside: What normal form is the lookup table in?

8.8.1 Normalization Exercise - Garage Shop

A vehicle repair-shop wants to keep records on vehicle repairs based on rules listed below. Normalize the database to 3NF.

Customer

1. A customer can have more than one vehicle
2. A customer brings in one vehicle at one time
3. Customer address is not recorded

Vehicle

4. Vehicle may be taken for a test drive; odometer reading may be higher when the vehicle is returned than when it is brought in
5. A vehicle can be serviced or maintained for more than one problem

Mechanic

6. A list of mechanics that work for the Garage-Shop is maintained
7. A vehicle can be worked on by only one mechanic during a repair incident
8. A mechanic can work on more than one vehicle in a day
9. A different mechanic can work on the same vehicle if it has more than one problem

Invoicing

10. Each instance of repair is recorded, with `DateTimeIn`, `DateTimeOut`
11. Cost of repair is not recorded (to keep simplicity in this exercise)
12. A vehicle taken in for repair is identified by `InvoiceID`
13. Each problem for an invoice has a `ProblemID`
14. `ProblemID` identifies the problem within an invoice. The count is reset for a new invoice
15. `InvoiceID` is unique for the company
16. Time when vehicle is brought in and returned is stored. Time is stored in 24 hour format

`CustomerID`, `CustomerName`, `CustomerPhone`, InvoiceID, `VIN, Make, Model, OdometerIn, OdometerOut, DateTimeIn, DateTimeOut, {ProblemID, MechanicID, MechanicName, Problem}`

`357, Varoq, 613-734-8712, IV-5291, RQ-3861, Toyota, Corola, 439876, 439880, 29-Oct-2014 11:29, 30-Oct-2014 15:30, {PR-01, S-211, Siv, Brake squeaking | PR-02, G-465, Gab, Noise in rear passenger wheel}`

`361, Tevif, 819-425-1329, IV-5292, LX-2478, Honda, Civic, 198176, 198176, 1-Nov-2014 9:27, 4-Nov-2014 12:30, {PR-01, X-191, Xad, Oil Change | PR-02, Y-532, Yob, Differential noise at low speed | PR-03, G-465, Gab, Change headlight bulb}`

Supplementary Exercises

1. In each table specify the number of rows the table will have, based on the sample data provided.
2. Write an SQL statement using `JOIN` clauses to simulate the results from 3NF to appear similar to your table in 1NF.

8.8.2 Normalization Exercise - Hotel Booking

A family owned hotel keeps records of its guests, employees and rooms. They take in guest bookings. Guests come repeatedly to their hotel. A small group of employees maintain and clean the room after the guests have left.

Draw a logical ER diagram and then a physical ER diagram from the information given. Show all cardinality. Chose appropriate names for entities and tables.

1. A guest can make more than one booking
2. A room can be occupied by a guest on many occasions
3. An employee maintains several rooms
4. A room can be maintained or cleaned by any employee

Hotel Rules:

1. Same `RoomCost` applies to all guests.
2. `BookingID` is unique for the hotel. Each room occupancy has a unique `BookingID`.
3. Sample data shows bookings and maintenance for two rooms.
4. Every time a guest leaves the room undergoes maintenance.
5. One maintenance is done per `BookingID`.
6. A new `BookingID` is given to a guest who books more than one room at the same time.
7. Only one employee cleans a particular room.

Aside: For simplicity, a convention if used for names, numbers and ID's

1. Guest names are chosen as five letters.
2. `GuestID` is coded as X-999, X is the first letter of the guests first name.
3. Employee names are four letters.
4. `RoomNumber` is three numerals; a room number is unique to the hotel.
5. `EmployeeID` is coded as X-99, X is the first letter of an employees first name.
6. `Checkin`, `Checkout` includes date and time. The format is yyyy-mm-dd, hh:mm, 24 hour time.

Normalize the database to 3NF. Write the column names for each normal form; do not show data elements.

```
RoomNumber, RoomType, RoomCost, { GuestID, GuestName, BookingID, CheckIn, CheckOut, EmployeeID,
EmployeeName, MaintenanceDate }
```

Sample Data:

```
235, Single BR, 112.00, { F517, Fiwix, 4, 2014-10-30 07:30:00, 2014-11-02 09:30:00, K35, Kiox,
2014-11-02 11:00 |
 V385, Vitec, 7, 2014-06-21 01:30:00, 2014-06-25 14:00:00, P29, Peec, 2014-06-25 14:30:00  }
342, Suite, 180.00, { Q624, Qusar, 6, 2014-11-07 18:30:00, 2014-11-11 12:30:00, G59, Gaav,
2014-11-12 08:00:00 | M331, Meqit, 5, 2014-06-29 18:30:00, 2014-07-01 19:30:00, G59, Gaav,
2014-07-02 08:00:00 }
108, Double BR, 140.00, { F517, Fiwix, 12, 2015-01-07 12:30:00, 2015-01-09 15:00:00,
P29, Peec, 2015-01-10 08:00:00 }
```

8.9 Review Questions

1. The entity integrity rule states that:
   (a) no primary key attribute can be null
   (b) referential integrity must be maintained across all entities
   (c) a primary key must have only one attribute
   (d) each entity must have a foreign key
2. A foreign key must match a primary key value in another relation or the foreign key value must be `NULL`. This rule is called
   (a) Entity Integrity Rule
   (b) Referential Integrity Rule
   (c) Anomaly
   (d) Normalization
   (e) Functional Dependency
3. What value will you assign to an attribute when you do not know it's value or when the value is unknown
   (a) Use the same value from the previous record
   (b) Use the same value from the next record
   (c) Take the average values from the table and use it
   (d) Assign a `NULL` value
   (e) Remove the attribute from the table
4. An inconsistency may occur when attempting to update a table that is not normalized. What is the term given to such a situation.
   (a) functional dependency
   (b) transitive dependency
   (c) anomaly
   (d) domain constraint
   (e) recursive foreign key
5. A database is in First Normal Form when the following two conditions are satisfied
   (a) Repeating groups have been resolved and every non-key attribute is fully functionally dependent on the primary key
   (b) Repeating groups have been resolved and primary key has been defined
   (c) Repeating groups have been resolved and it has no transitive dependencies
   (d) Partial dependencies and transitive dependencies have been removed
6. A database is in Second Normal Form when Database is in First Normal Form and
   (a) repeating groups have been resolved
   (b) every non-key attribute is fully functionally dependent on the primary key
   (c) primary key has been defined
   (d) it has no transitive dependencies

7. A database is in Third Normal Form when Database is in Second Normal Form and
   (a) every non-key attribute is fully functionally dependent on the primary key
   (b) repeating groups have been resolved
   (c) primary key has been defined
   (d) it has no transitive dependencies

8.10 Supplementary Questions

Answer True or False

1. Normalization is done after Logical Database Design is complete.
2. Normalization takes into consideration how data is displayed, how it is used in reports and how a database is queried.
3. One of the two conditions for a relation to be in First Normal Form is: absence of repeating groups.
4. When inserting a row in a *normalized* database, i.e. 3NF, there may be additional insertions that result in duplication of data.
5. When deleting a row in a *normalized* database, i.e. 3NF, there may be loss of data.
6. When modifying a single row in a *normalized* database, there may be changes required to other rows.
7. The process of Normalization is reversible, i.e. it is possible to put data in 1NF from 3NF.
8. In the Normalization process, it is possible that some columns can be ignored.
9. A `NULL` value is the same as a space or a zero.
10. The entity integrity rule states that a primary key attribute can be null.
11. Referential integrity is satisfied when a value of one column of a table exists as a value of another column in a different, or same, table.
12. A foreign key can have a `NULL` value.
13. The columns of a relation can be rearranged without changing the meaning or use of the relation.
14. Rows of a relation must not be interchanged and must be stored in a certain sequence.
15. A primary key uniquely identifies each row in a relation.
16. A composite key consists of only one attribute.

Select three statements:

The *objective* of Normalization is to

A. derive relations that are free of anomalies
B. simplify the enforcement of referential integrity constraints
C. make it easier to print reports, build queries and display data
D. make it easier to maintain data
E. pay attention to processing efficiency
F. determine a method to permanently save data on physical storage

8.11 Prime Key Identification - Exercise

1. Identify the prime key in table 8.7 shown below.
   Is the key composite?

A	B	Cups
Carrot	Milk	1
Sugar	Sugar	3
Carrot	Honey	1
Pepper	Sugar	3

Table 8.7: Ingredients & Measurements

2. Identify the prime key in table 8.8 shown below.
   Is the key composite?

Animal	Diet	Weight
Horse	Hay	900
Horse	Oats	900
Grizzly	Deer	250
Grizzly	Berries	250
Elk	Vegetation	475
Moose	Vegetation	475

Table 8.8: Animal, Diet & Weight

3. Identify the prime key in table 8.9 shown below.
   Is the key composite?

Day	**Patient**	**Mood**
Mon	Waor	Happy
Mon	Yiur	Sad
Mon	Roup	Happy
Tue	Qaam	Grumpy
Tue	Wail	Tired
Tue	Waor	Happy
Wed	Waor	Joyful
Wed	Wail	Excited

Table 8.9: Day, Patient & Mood

4. The table below lists household pets and the food they eat. Identify the prime key, if it exists, in table 8.10 shown below.

Pet	**Food**
Dog	Bone
Cat	Fish
Fish	Worm
Dog	Kibble
Horse	Hay
Dog	Bone
Rabbit	Cabbage
Horse	Carrot
Rabbit	Carrot

Table 8.10: Pet & Food

5. Identify the prime key in table 8.11 shown below.
   Is the key composite?

A	B	C	D
3	5	7	9
3	1	4	6
3	5	2	9
8	5	2	7
7	9	4	6

Table 8.11: Numbers

8.12 Further Normalization

Terms:

key: A combination of one or more attributes that uniquely identifies rows in a relation.

Candidate Key A key that determines all the other columns in a relation. Stated differently, a candidate key uniquely identifies a row in a relation. *Note:* A key although used in a singular noun, can have more than one attribute. *Aside:* It is possible to have more than one candidate key in a relation; i.e. it is possible to have more than one set of attributes that qualify as a candidate key. A key K_2 cannot be a candidate key if key K_1 is a proper subset of K_2.

Functional Dependency A value of one or more attributes determines the value of another attribute. Functional dependency is represented by A $\rightarrow$ B, which implies B is functionally dependent on A. Also, A determines B. Each value of A is associated with exactly one value of B. *Note:* A and B may consist of more than one attribute.

For a given value of A only one value of B can be found. When two tuples have the same value of A they must also have the same value of B. For a given value of **B** there may be more than one value of **A**. It is intuitive to define A and B in the direction of the arrow - A functionally determines B.

Determinant In the functional dependency is representation A $\rightarrow$ B, A is the determinant. Loosely, the attribute on the left hand side of the representation is a determinant.
A determinant is an attribute, or a set of attributes, that other attributes are fully functionally dependent.

Boyce-Codd Normal Form (BCNF) A strict form of 3NF. To bring a relation to BCNF data values, in a relation that is in 3NF, need to be examined. A relation in BCNF is also in 3NF; a relation in 3NF may not be in BCNF. A relation is in BCNF *iff* every determinant is a candidate key.
For a table to be in 3NF it is sufficient to remove partial functional dependencies from the prime key and then remove transitive dependencies considering the prime key. For a relation to be in BCNF partial functional dependencies must be removed from all candidate keys. 3NF can be achieved by an understanding of the attributes from business rules, to bring a relation to BCNF individual data values need to be analyzed. A relation can be further normalized to BCNF only if there are more than one candidate keys; i.e. a table that does not have more than one candidate key is guaranteed to be in BCNF.

Superkey A relation will always have at least one key. For example, a relation with three attributes `A`, `B`, `C`, will always have at least a key defined as `ABC`. This is called a trivial superkey. This property exists because in a relation, no two rows are the same.

Equivalent Terms: Table, Relation, File. Column, Attribute, Field. Row, Tuple, Record. Reference: Database Processing.

Exercise 1: Compare the definitions of **key**, **candidate key** and **prime key**. Note the differences. Can a table have

1. more than one candidate key?
2. more than one prime key?

Exercise 2: A table has `n` attributes. How many possible keys can it have?

Exercise 3: A relation has at least one candidate key. Is the statement true?

8.13 Exercise 9 - Author-Publisher

Objective

1. Normalize the given data to 3NF. Show each step in the normalization process.
2. After the normalization process, write a `JOIN` statement to bring the 3NF back to 1NF, confirming that the process is reversible.

Submission: Submit Logical and Physical ER diagrams on paper. Answer the 5 questions. *Aside:* You should not spend more than 1 hour on this exercise.

Abstract: A smaller section of the library database is represented. The library wants to maintain a record of borrowers who have borrowed a copy of a book, with a few additional details.

Rules: A book has one or more copies. A borrower may borrow a copy of a book. A copy of a book may be borrowed by more than one borrower; only one borrower may borrow a copy at any one time.

Book

`BookID` is the unique for each book title.
The library maintains a copy number for each book of the same title; each copy of a book is identified by `CopyID`.
The same book title can be lent to more than one borrower on another occasion.
Copy number may begin with `1` for a title, but may have a number other than `1`.
It is possible that a copy of a book has not been borrowed.
Each time a book is borrowed a unique `TransactionID` is generated by the system.

Borrower

A borrower is identified by a `BorrowerID`. A borrower can borrow more than one book at a time.
A borrower may borrow the same copy of the book on another occasion.
A borrower may return all books such that he does not have any books on loan.
If borrower returns all borrowed books, he is still considered a borrower; he may borrow in future.
Date of book borrowed is stored.
Date of book returned is stored.
The systems should ensure that date borrowed is not after the date returned. This constraint should ensure quality of data.

Un-normalized Data:

```
BookID, BookTitle { CopyID, { TransactionID, BorrowerID, BorrowerName,
DateBorrowed, DateReturned } }
```

```
BRKA, The Brothers Karamazov { 1, { 1001, R01, Reg Redfuh, 21-Nov-2017, 30-Nov-2017 | 1002,
Y01, Yon Yencir, 13-Dec-2017, 30-Nov-2018 }, 2 { 1003, G01, Gip Gebpel, 2-Feb-2018, 21-Mar-2018
} }
```

```
DVDCOP, David Copperfield, { 1, { 1004, V01, Vit Vetrup, 17-May-2018, 15-Jun-2018 | 1005, T04,
Tud Tidmeg, 2-Jul-2018, 5-Jul-2018 }, 3 { 1006, M05, Miy Mihvol, 17-Aug-2018, 1-Sept-2018 |
```

```
1007, R01, Reg Redfuh, 21-Nov-2017, 30-Nov-2017 } }
```

Question 1. **[4 marks]** Draw an logical ER diagram and then a physical ER diagram from the information given. Show all cardinality. Choose appropriate names for entities and relations.

Question 2. **[4 Marks]** Normalize the database to 1NF from the unnormalized data. Write the column names for each normal form; show data elements.

Question 3. **[4 Marks]** Normalize the database to 2NF. Write only the attributes; do not show data elements.

Question 4. **[4 Marks]** Normalize the database to 3NF. Write only the attributes; do not show data elements. In each of the tables indicate the number of rows the table will have, based on the given un-normalized data.

Question 5. **[4 Marks]** Use `JOIN` clauses to simulate the results from 3NF to appear similar to your table in 1NF.

8.14 Exercise 10 - Normalization

Objective

1. Bring table 8.3 page 115 to 3NF.
2. Draw a logical ERD
3. Write DDL and DML statements for the 3NF. Test and run the statements
4. Write `INSERT` statements for the data in table 8.3 page 115. Test and run the statements.
5. Write `JOIN` statements for the new tables to show data in 1NF
6. Write Views
7. Reverse Engineer your ERD

Requirements

1. Verify the number of rows in each table. Show your results to the lab instructor to obtain marks for the lab.
2. DDL should have all prime key and foreign constraints. Use DROP statements at the beginning of the file.
3. DML statements should run. Use DELETE statements at the beginning of the .sql file
4. Write views for the JOIN statements
5. Reverse Engineer your ERD from the DDL, compare it with the hand drawn ERD.

8.15 Lab 9 - Author-Publisher

Reference

Section 8.5 page 113, Section 8.7 page 119.

Objective

1. Normalize the given data to 3NF. Show each step in the normalization process.
2. Indicate the number of rows in the each table during each stage of the normalization process.
3. Create and populate database.
4. Enforce constraints to maintain referential integrity.
5. Create and use `INDEX` and `VIEW`.
6. After the normalization process, write a `JOIN` statement to bring the 3NF back to 1NF, confirming that the process is reversible.

Submission

Submit Logical and Physical ER diagrams; the Normalization Process, Database in 3NF..

Requirements

Normalize the data to 3NF, show each step. Create tables in 3NF, reverse engineer, save the ER diagram to a `.pdf` file, each entity should be related to atleast one entity.

Abstract

A small city library wants to store its inventory of books. Data for publishers, authors, books and borrowers are maintained. The sample data represents record of people borrowing books from a library.

Rules:

Author and Publisher

An author writes at least one book.
A publisher publishes at least one book.
A book title is published by only one publisher.
A book can be associated with only one author, the primary author of the book
A book having more than one author can have only one authors name, the primary author.
The same book title cannot be published by more than one publisher

Book

BookID is the same for the each title.
BookID is different for each title by the same author.

Library

The library maintains a copy number for each book of the same title.

The same book title can be lent to more than one borrower.

The library maintains a copy number for each book of the same title.

Copy number begins with 1 for a title.

Copy number increments by 1 for each copy of the same title.

The copy number is reset for each title.

It is possible that all copies of a book are in the library.

It is possible that no copy of a book has been borrowed.

A person is a borrower even if he has not borrowed a book; he may borrow in future.

If borrower returns all borrowed books, he is still considered a borrower.

A borrowers historical data is not maintained.

Borrower

A borrower can borrow more than one book.

A borrower may return all books such that he does not have any books on loan.

```
BookID, BookTitle, AuthorID, AuthorName, PublisherID, PublisherName,
 { InventoryID, BorrowerID, BorrowerName }

ALCHE, The Alchemist, COELHOP, Paulo Coelho, HARP, Harper,
 { 1, MORZ, Zev Moriv | 2, GATD, Dof Gatum | 3, TOSF, Gin Tosig }


ANNAK, Anna Karenina, TOLSTOL, Leo Tolstoy, ROSI, Rosiya Press,
 { 1, FELT, Taz Felor }


TFA, Things Fall Apart, ACHEBEC, Chinua Achebe, SELF, Self Press,
 { 1, FIJJ, Juv Fijoy | 2, GATD, Dof Gatum }


TKAMB, To Kill A Mockingbird, LEEH, Harper Lee, WILE, Wiley,
 { 1, LUHK, Kit Luhuh }


WNP, War and Peace, TOLSTOL, Leo Tolstoy, ROSI, Rosiya Press,
 { 1, LUHK, Kit Luhuh }
```

Requirements for Hybrid Component

1. Submit an initial, i.e. logical, ERD on paper using a pencil. You may scan the paper, take a picture and upload or submit the paper itself.
2. Write and test DDL statements to create tables with constraints using the 3NF relations.
3. Reverse Engineer the ERD from the DDL statements.
4. Write and test DML statements to enter the data in the tables.
5. Write `INSERT` statements to add two borrowers, these two borrowers have not borrowered any books from the library. The two borrower names are *Heyia Suls* and *Jileau Kurc*. Use your own BorrowerID.
6. Write `UPDATE` statements to change the name from *Heyia Suls* to *Hayia Suls*, use BorowerID to identify the row.
7. Write a join statement to make the 3NF relation appear as in 1NF
8. Write a `LEFT` or `RIGHT OUTER JOIN` statement to identify the two borrowers who have not borrowed any books from the library.

9

SQL - Transaction Management & Other Topics

9.1 Introduction

Several topics on are covered in this Lesson. Subqueries, including co-related and non-corelated subqueries, User Defined Data Type (UDT), Derived Attributes, Functions & Stored Procedures, Triggers and Transaction Management.

9.2 Objective

- Recognize the need for Transaction Management
- Determine the need for embedded SQL and dynamic SQL in web based applications

9.3 Sub-query

A query nested inside another query is known as a `subquery`. The nested (inner) query executes first, its result is equated to the *outer* query.

Figure 9.1 illustrates a subquery. It lists cities in Canada whose population is higher than the average of all cities in Canada. The inner most query
`( SELECT AVG( population ) FROM City )`
is run first, its result is then compared to the `WHERE` condition in the outer query. Type and run this query in the `world` database. *Aside:* Change the greater than sign to a less than sign, observe the result. A query illustrated in figure 9.1 is also called *non corelated subquery*, the inner query is independent from the outer query. It is run only once, its results are then passed on to the outer query.

```
SELECT ID, Name, Population
FROM City
WHERE Population > ( SELECT AVG( Population ) FROM City )
AND CountryCode = 'CAN';
```

Figure 9.1: Sub Query

A `correlated` subquery, also known as a synchronized subquery, uses values from the outer query. The subquery is evaluated once for each row processed by the outer query; an incorrectly used corelated subquery can slow down the results. Figure 9.2 and 9.3 are examples of correlated subqueries. In the first example the query lists all cities which have their population less than all cities in that country. The subquery (i.e. inner query) is evaluated each time the outer query is evaluated. This query is not efficient, neither it is optimized. Note the use of an alias in the table name. Recall that an alias is not permitted in the `WHERE` clause.

In the second example the query is part of the `SELECT` clause. Using the `ROUND` function the result is less cluttered. For example,

```
( SELECT ROUND( ( AVG( Population ) ), 0 )
```

```
SELECT ID, Name, CountryCode, population AS "City Population"
FROM City AS "Avg City Population"
WHERE Population < ( SELECT AVG( population )
FROM City
WHERE CountryCode = "Avg City Population".CountryCode );
```

Figure 9.2: Correlated Sub Query

```
SELECT CountryCode, Name, Population,
( SELECT ( AVG( population ) ) AS "Country Average"
FROM City WHERE CountryCode = City_T.CountryCode )
FROM City AS City_T;
```

Figure 9.3: Correlated Sub Query. Query in the `SELECT` Clause

Study the following two queries. In figure 9.4 the expression in the outer `SELECT` is a subquery; it is evaluated once for each row in the result set. Figure 9.5 gives similar results, but it is a non-corealted query.

```
SELECT AuthorID, AuthorName,
( SELECT COUNT( AuthorID ) AS "Popular"
FROM Inventory_T
NATURAL JOIN Book_T
NATURAL JOIN Author_T
WHERE A.AuthorID = Author_T.AuthorID
GROUP BY AuthorID )

FROM Author_T AS A
GROUP BY AuthorID;
```

Figure 9.4: Corelated query, Lists the number of books borrowed for each Author

```
SELECT AuthorID, AuthorName, COUNT( AuthorID ) AS "Popular"
FROM Author_T
NATURAL JOIN Book_T
NATURAL JOIN Inventory_T
GROUP BY AuthorID;
```

Figure 9.5: Non corelated, Lists the number of books borrowed for each Author

9.4 Review Questions

1. In a non correlated sub query which query runs first
   (a) inner query
   (b) outer query
2. A type of subquery where processing the inner query depends on data from the outer query.
   (a) correlated
   (b) non-correlated
   (c) inner
   (d) outer
   (e) embedded
3. A subquery that is executed once for the entire outer query is
   (a) embedded
   (b) correlated
   (c) non-correlated
   (d) outer
   (e) dynamic

4. Which one of the sub-queries does not depend on data from the outer query
   (a) inner
   (b) embedded
   (c) correlated
   (d) non-correlated
   (e) outer

9.5 Learning Activities

1. Use the `world` database to write correlated subqueries. Identify the *inner* and *outer* queries

9.6 `EXISTS` operator

The `EXISTS` operator tests if a sub-query results in one or more rows; a `TRUE` value is returned if one or more rows exists. `EXISTS` will look for atleast one row and return a `TRUE` value, it will not process all rows in the subquery. The return value can be negated by using `NOT EXISTS`.

For example, the query in figure 9.6 lists customers that have an invoice in the `Invoice_T` table, and the result is reversed in query in figure 9.7. Try replacing the sub-query with `LIMIT 0` as in
`SELECT Cust_ID FROM Invoice_T WHERE Customer_T.Cust_ID = Invoice_T.Cust_ID LIMIT 0`
in each case, and observe the results.

```
SELECT Cust_ID FROM Customer_T
WHERE EXISTS
( SELECT Cust_ID FROM Invoice_T WHERE Customer_T.Cust_ID = Invoice_T.Cust_ID );
```

Figure 9.6: Example of an `EXISTS` operator

```
SELECT Cust_ID FROM Customer_T
WHERE NOT EXISTS
( SELECT Cust_ID FROM Invoice_T WHERE Customer_T.Cust_ID = Invoice_T.Cust_ID );
```

Figure 9.7: Example of an `NOT EXISTS`

Aside: In this case, similar results can be obtained using the `IN` keyword.

```
SELECT Cust_ID FROM Customer_T
WHERE Cust_ID IN( SELECT Cust_ID FROM Invoice_T );

SELECT Cust_ID FROM Customer_T
WHERE Cust_ID NOT IN( SELECT Cust_ID FROM Invoice_T );
```

9.7 User Defined DataType - UDT

Maintaining and enforcing a consistent naming scheme for attributes is a challenge in a database system. Think of a large organization with many departments - manufacturing, sales, marketing, design among others. How would all departments use the same attribute name for `postal code` and they all enforced a six character length with the combination of letters and numerals in the Canadian postal system. A User Defined Datatype (`UDT`) is the answer. PostgreSQL `DOMAIN` is an object to implement a `UDT` with (or without) constraints.

The example in 9.8 creates a datatype called `Canada_PostCode_D` with a CHECK constraint of `A9A9A9`, character-numeral combination. This is implemented using a regular expression, the tilde ẽnforces case sensitivity.

```
DROP TABLE IF EXISTS Canada_Post_T;
DROP DOMAIN IF EXISTS Canada_PostCode_D;


-- domain will allow valid string of
-- characters and numerals as postcode used in Canada
CREATE DOMAIN Canada_PostCode_D AS TEXT
CHECK( VALUE ~'[A-Z][0-9][A-Z][0-9][A-Z][0-9]');

CREATE TABLE Canada_Post_T(
Address_ID SERIAL PRIMARY KEY,
Street1 TEXT NOT NULL,
Street2 TEXT,
City VARCHAR( 25 ) NOT NULL,
PostCode Canada_PostCode_D NOT NULL);
```

Figure 9.8: User Defined Datatype - UDT

Try the following insert statements to confirm the constraint.

```
DELETE FROM Canada_Post_T;

-- Successful INSERT operation
INSERT INTO Canada_Post_T( Street1, City, PostCode )
VALUES( '1385 Woodroffe Avenue', 'Ottawa', 'K2G1V8' );
```

```
-- Unsuccessful INSERT operations
INSERT INTO Canada_Post_T( Street1, City, PostCode )
VALUES( '1385 Woodroffe Avenue', 'Ottawa', 'K2g1V8' );

INSERT INTO Canada_Post_T( Street1, City, PostCode )
VALUES( '1385 Woodroffe Avenue', 'Ottawa', 'K21V8' );

INSERT INTO Canada_Post_T( Street1, City, PostCode )
VALUES( '1385 Woodroffe Avenue', 'Ottawa', '2K31V8' );
```

Thought Question: What would be the advantage of implementing such a constraint at the DDL level compared to implementing at the programming level or user level?

9.8 Derived Attribute

The name suggests that it is an attribute expressed by existing values in the database. For example in the Inventory database in figure 5.6, page 70 `Line_Price` can be derived from `Prod_Price` in the `Product_T` table and `Line_Units` in `Invoice_Line_T` table. As a data administrator you need to decide if this values needs to be stored, even though it can be derived each time a report is printed.

The age of a person is derived from date of birth and the current date, taken from the operating system. Age of a living person changes, it is usually derived and rarely stored in the database.

A derived attribute is represented in square brackets in a ERD. Not all ERD diagramming tools have the provision to represent a derived attribute. The code sample in figure 9.9 shows how the `Line_Price` is calculated in the `Invoice_Line_T` table.

```
-- Query to update a column using data from a parent table
-- Run these SQL statements on the Inventory Database after
-- running the DDL and DML files.

SELECT * FROM Invoice_Line_T;

-- first set Line_Price to zero in all records in Invoice_Line_T
UPDATE Invoice_Line_T SET Line_Price = 0;

-- UPDATE the column Line_Price
UPDATE Invoice_Line_T SET Line_Price = ( Line_Unit *
( SELECT Prod_Price FROM Product_T WHERE Invoice_Line_T.Prod_Code = Product_T.Prod_Code ) );

-- Check to see the update results.
-- Compare the last two columns
SELECT Product_T.Prod_Code, Product_T.Prod_Price, Line_Unit, Line_Price,
( Line_Unit * Product_T.Prod_Price ) AS "Check"
FROM Invoice_Line_T
JOIN Product_T USING( Prod_Code );
```

Figure 9.9: Update column `Line_Price` in Inventory Database

9.9 Transaction Management

A transaction is a unit of work that changes the state of a database. Examples of transactions are `INSERT`, `UPDATE` and `DELETE` statements, among others. A `SELECT` statement does not alter the database, it is not a transaction. Often several SQL statements need to run as a single unit. For example, if we had one record insertion in the `Invoice_T` table and 2 records inserted in the `Invoice_Line_T` table, a total of 3 statements in two tables as shown in Figure 9.10.

```
INSERT INTO Invoice_T( Invoice_Number, Cust_Id, Invoice_Date )
VALUES( 'I23008', 'C006', '2011-12-15');

INSERT INTO Invoice_Line_T( Invoice_Number, Invoice_Line, Prod_Code, Line_Unit, Line_Price )
VALUES( 'I23008', 1, 'P2016', 3, 689.00 );
INSERT INTO Invoice_Line_T( Invoice_Number, Invoice_Line, Prod_Code, Line_Unit, Line_Price )
VALUES( 'I23008', 2, 'P2017', 3, 35.99 );
```

Figure 9.10: Transactions in Inventory Database

These three statements must be completed together or not at all. If only the first or the first two statements ran successfully and the remaining statements were not run, the data would be inaccurate and even inconsistent.

Transaction Management allows to run all statements together or none at all. To do this the statements are wrapped around `BEGIN TRANSACTION` and `END TRANSACTION`, if there is an error the statements are nullified using the `ROLLBACK` command.

The exercise below illustrates how `ROLLBACK` will affect your commands.

1. start `psql` There are several ways to do this.
2. List all databases on server: `\l`
3. The command to connect to a database is: `\c <database>`. Connect to Inventory: `\c Inventory`
4. List all tables in the database: `\dt`
5. By default AUTOCOMMIT is ON, i.e all commands will automatically will be permanent.
   Turn AUTOCOMMIT OFF, type in. `\set AUTOCOMMIT off`. Now commands can be reversed.
6. Display records in `Invoice_Line_T;`
   `SELECT * FROM Invoice_Line_T;`
7. Delete records from table. `DELETE FROM Invoice_Line_T;`
8. Check records, `SELECT * FROM Invoice_Line_T;`
   Do you see a listing?
9. Type in: `ROLLBACK;` Your commands are now reversed.
10. `SELECT * FROM Invoice_Line_T;`

Transaction Management - Review Questions

1. The SQL statement

   ```
   SELECT * FROM <table>;
   ```

   is a transaction

   (a) True

   (b) False

2. A unit of work that changes the state of a database is a/an

   (a) trigger

   (b) stored procedure

   (c) embedded query

   (d) transaction

   (e) dynamic SQL query

3. Which statement undoes a transaction

   (a) `COMMIT`

   (b) `UNDO`

   (c) `REDO`

   (d) `ROLLBACK`

4. Which statement make changes to a database permanent

   (a) `COMMIT`

   (b) `UNDO`

   (c) `REDO`

   (d) `ROLLBACK`

   (e) `SAVE`

5. Transaction management is needed when:

   (a) a transaction consists of just one SQL command

   (b) there is a security risk

   (c) multiple SQL commands must be run as a single transaction

   (d) there is more than one database administrator

   (e) triggers are used

9.10 Review Questions

Advanced SQL

1. A type of query that is placed within a `WHERE` or `HAVING` clause of another query is called a/an:
   (a) PL/SQL
   (b) subquery
   (c) embedded SQL
   (d) dynamic SQL
   (e) trigger
2. Identify the operator that takes a value of `TRUE` if a subquery returns one or more rows in an intermediate results table.
   (a) `IN`
   (b) `HAVING`
   (c) `EXISTS`
   (d) `WHERE`
3. Identifying specific attributes in the `SELECT` clause, instead of using `SELECT *` will help reduce network traffic
   (a) True
   (b) False
4. SQL statements in a program written in another language such as C or Java are called
   (a) dynamic SQL
   (b) embedded SQL
   (c) sub-query
   (d) join
   (e) union
5. Embedded SQL statements can create a flexible and accessible interface for the user
   (a) True
   (b) False
6. Embedded SQL statements help enforce security by granting permission to required applications instead of granting permission to users
   (a) True
   (b) False
7. SQL statements are built by DBMS at the time a user or procedure requests data from a DBMS or a transaction is performed. The name given to such a concept is
   (a) Dynamic view
   (b) Material view
   (c) Dynamic SQL
   (d) Triggers
   (e) PL/SQL

9.11 Summary

This chapter has a few intermediate and advanced SQL topics that are covered in a later course in greater detail, they include triggers, stored procedures, embedded SQL, dynamic SQL, range constraints and user defined data types. In a sub-query, results of one or more queries are evaluated first, these results then serve as a parameter to another query. Correlated queries and non-correlated sub-queries are differ in their execution. Care must be taken when writing correlated sub-queries, they are CPU intensive when compared to non correlated sub-queries.

9.12 Exercise 11 - UPDATE with a sub query

Objective

1. Use a subquery to update a table.
2. Use `ALTER TABLE`

Reference

1. Lab 4, Section 5.9 page 68
2. Exercise 5, Section 3.6 page 48
3. Physical ERD, figure 5.6, page 70

Submission Run each example query, complete the exercises. Complete the online quiz for this lab.

Pre-requisites Complete the lab and exercise listed in the Reference section before attempting this lab.

Background This exercise augments Lab 4. Three tables are modified by adding columns using `ALTER TABLE`, these newly created columns are then updated using the `UPDATE` command.

First, add a column `Prod_Price`to the `Invoice_Line_T` table. This table does not have the price of the product at the time of creating the Invoice. The following statement creates the column.

```
ALTER TABLE Invoice_Line_T ADD COLUMN Prod_Price NUMERIC( 5, 2 );
```

Aside: Use
`ALTER TABLE Invoice_Line_T DROP COLUMN IF EXISTS Prod_Price;`
if you wish to run the .sql file while testing the code.

Now, update this newly created field with `Prod_Price` from the `Product_T` table. The following statement will perform the update.

```
UPDATE Invoice_Line_T SET Prod_Price = (                SELECT Prod_Price FROM Product_T WHERE
Product_T.Prod_Code = Invoice_Line_T.Prod_Code );
```

Next we attempt to update only a subset of records, this is what would happen in reality.

Visualize a scenario when the Product Price has reduced and we want the new price to take effect for all new invoices. This product has been sold twice, in Invoice Number I23001 and I23004. For this example, we want to change only one of the two invoices, I23004.

Change the Product Price `Prod_Price` for `Prod_Code` `P2014` to `149.99`, using

```
UPDATE Product_T SET Prod_Price = 149.99 WHERE Prod_Code = 'P2014';
```

Finally update the new price from the Product table for a single invoice, an additional condition in the `WHERE` clause is added.

```
UPDATE Invoice_Line_T SET Prod_Price = (
SELECT Prod_Price FROM Product_T WHERE Product_T.Prod_Code = Invoice_Line_T.Prod_Code
AND Invoice_Line_T.Invoice_Number = 'I23004' );
```

Question 1. Write an SQL statement to multiply the `Prod_Price` with `Line_Unit` in the `Invoice_Line_T` table. Update the `Line_Price` with this value.

Question 2. Update the above query to update records from a single invoice of your choice.

Question 3. Alter the `Invoice_T` table, add a column, call it `Invoice_Amount`, with data type `NUMERIC( 9, 2 )`.

Question 4. Write an UPDATE statement to add all the Line_Price for each invoice and store this amount in the `Invoice_T` table in the newly created `Invoice_Amount` column.

Question 5. Write an update statement to replace `Cust_Balance` in the `Customer_T` table with the sum of all `Invoice_Amount` for each customer.

9.13 Exercise 12 - Tutor

Objective

Use the following techniques in `postgreSQL`

1. Restore and backup
2. Lookup table
3. `ALTER TABLE` to add a column
4. Use of `BETWEEN` clause
5. Use of `SERIAL` keyword
6. `ENUM` keyword
7. `RANGE`
8. `DEFAULT`

Reference

1. Section 8.7 Enumerated Types from `postgresql-9.5-US.pdf` page 147 ff
2. Section 5.3.1 Check Constraints from `postgresql-9.5-US.pdf` page 57 ff
3. Section 8.17 Range Types from `postgresql-9.5-US.pdf` page 181 ff

Abstract

The College Tutor program wants to maintain a data to track students in need of tutors. The system needs to register tutors and the courses they can tutor. Pairing a student with a tutor is currently done manually. This system shall suggest a possible tutor for a student. Each session with a student and tutor is recorded. The system shall provide reports on hours taught to a student, using range of dates among other parameters.

Rules

1. A list of courses that are available for tutoring are maintained by the system.
2. A student can place a request for a tutor for one or more than one course.
3. A tutor can place a request to tutor, tutors name is maintained by the system with the date of request.
4. A tutor may put his request on `HOLD`.
5. A tutor is listed in the system, but if she has not tutored for a predetermined time the status can be `INACTIVE`. An tutor with status `INACTIVE` may be tutored in the past.
6. Each session with a tutor and student is recorded; date, time and a short comment are also recorded.
7. Student ID starts with `S`.
8. Tutor ID starts with `T`.

Procedure

Instructions on restoring data from a backup are listed in section A.2 on page 172.

Download file `Tutor.backup`. Create a database called `Tutor`. After restoring, write the given queries. Reverse engineer the database, study the ER Diagram, get familiar with the entities, relationships and attributes.

Queries

1. Tutors who have not tutored as yet. *Note:* Refer rule 5.

2. Are there any students who have not been paired with a tutor?

3. Number of enrolled students, use `Student_T` Table.

4. Number of students, who have registered for at least one course.

5. Number of students, who have registered for more than one course.

6. Number of hours each tutor has tutored.

7. Total of Tutor-Student Hours in the entire system.

8. Tutor with the highest number of hours.

9. Count the number of unique courses in the database.

10. Number of Students in the database.

11. Number of tutors who have had tutoring sessions.

10

Modeling Data

10.1 Introduction

This lesson builds the foundation for Relational Databases. You are introduced to several new terms - they are used throughout the book. Modeling includes: Entities and Attributes, and their Relationships. Toward the end of the chapter Time Dependent Data Modeling is covered; this topic has relavance because of compliance regulations in US by Sarbanes-Oxley and Basel II and in Canada, Bill 198.

Considering the number of new terms and concepts introduced, we will revisit sections of this chapter several times. To focus our understanding on critical topics we will not introduce examples for certain concepts such as *degree of relationship*.

10.2 Objective

- Draw a Data Model from a give set of rules using entity relation diagram
- Business Rules
- List the types of attributes used in a DBMS
- Identify types of entities and relationships
- Define and differentiate *cardinality* and *degree*

10.3 Business Rules

Business Rules are the foundation of a data model, they represent the language and structure of an organization. They are derived from business policies, procedures, events, functions and other business objects. Business constraints are specified in business rules. Business rules provide a formal way to understand the organization by stakeholders. Some of the stakeholders are: owners of the organization, employees, information system designers in the organization. Business rules determine data in creation, storage, retrieval and data administration. Constraints are represented in Entity-Relationship Diagrams (ERD). Refer Page 54, [3].

10.4 Learning Activitiy

Use web resources to identify two *early* models of database that were used *before* relational model.

Is a `NULL` value different from blanks? What are the differences in queries when using `NULL` values

10.5 Relationships

In each of the ER diagrams fill in the entity of your choice that is appropriate to the the cardinality, label the relationship and enter a prime key and atleast one other attribute.

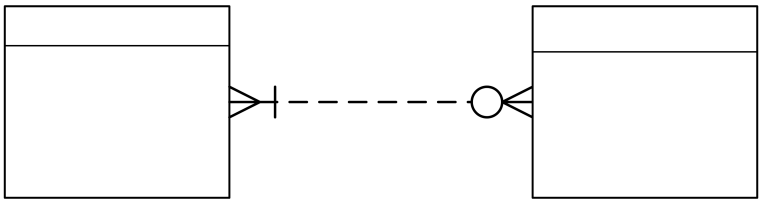


Figure 10.1: Many to Many

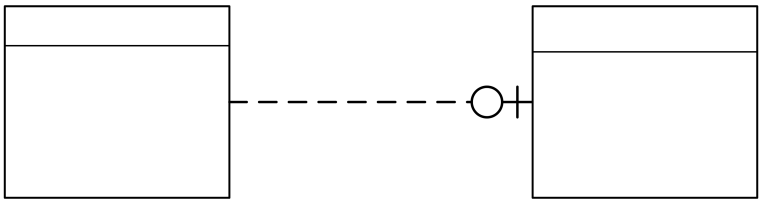


Figure 10.2: One to One

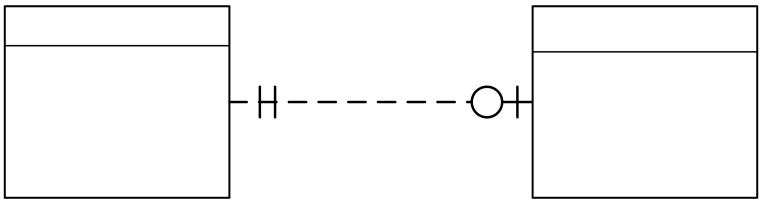


Figure 10.3: One to Mandatory One

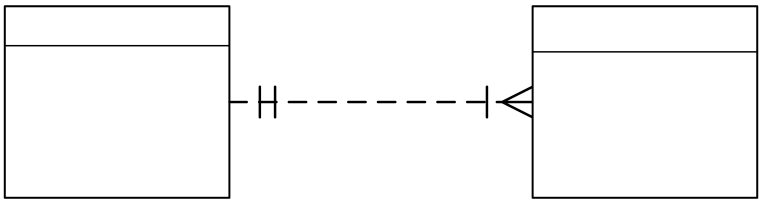


Figure 10.4: One to Many

10.6 Review Questions

Modeling the Rules of the Organization

1. A Business Rule must be
   (a) Stored at multiple locations in a central repository but expressed only once
   (b) Stored only once in a central repository then shared throughout the organization
   (c) Stored at multiple locations in a central repository and shared throughout the organization
   (d) Stored only once in a central repository but never shared within the organization
2. A business rule can have many interpretations. Each stakeholder can have his own interpretation.
   (a) True
   (b) False
3. Data models in an organization change less frequently than business rules.
   (a) True
   (b) False
4. A word or phrase that has a specific meaning for the business
   (a) words
   (b) entity
   (c) term
   (d) relationship
   (e) business rule
5. A *fact* is an association between two or more:
   (a) words
   (b) entities
   (c) terms
   (d) relationships
   (e) business rules
6. `Supplier`, `Student`, `Book`, `Course` are examples of
   (a) Relationship
   (b) Business Rule
   (c) Entity
   (d) Degree of Relationship
   (e) Fact

Modeling Entities and Attributes

7. A strong entity is an entity that exists independent of other entity types
   (a) True
   (b) False
8. Identify the entity whose existence depends on another entity
   (a) Identifying Owner
   (b) Identifying Relationship
   (c) Entity Instance
   (d) Strong Entity
   (e) Weak Entity
9. An attribute that must have a value for every entity (relation) instance is a/an
   (a) Multivalued Attribute
   (b) Atomic Attribute
   (c) Composite Attribute
   (d) Required Attribute
   (e) Optional Attribute
10. An attribute that cannot be broken into smaller components is called a/an
    (a) Multivalued Attribute
    (b) Atomic Attribute
    (c) Composite Attribute
    (d) Required Attribute
    (e) Optional Attribute
11. An attribute whose value can be calculated from other attributes is called
    (a) Multivalued Attribute
    (b) Atomic Attribute
    (c) Composite Attribute
    (d) Derived Attribute
    (e) Optional Attribute
12. A Relational Database represents data as a collection of
    (a) Bits
    (b) Bytes
    (c) Attributes
    (d) Tables
    (e) Tuples
13. A time value that is associated with a data value, indicating when the data value was updated
    (a) Composite Value
    (b) Effective Date

(c) Effective Time

(d) Time Stamp

(e) Compliance Regulation

14. The term `update` implies

(a) Insertion of a data value

(b) Deletion of a data value

(c) Change in data value

(d) Deletion or change in a data value

(e) Insertion, deletion or change in a data value

15. Consider an invoice. `Line_Item_Amount` is an attribute that is the product of `Quantity` and `Unit_Price`, `Line_Total` is the sum of all `Line_Item_Amount` in an invoice. The data analyst decides not to store `Line_Item_Amount` and `Line_Total`. How will you describe this attribute?

(a) Multivalued Attribute

(b) Atomic Attribute

(c) Composite Attribute

(d) Derived Attribute

(e) Optional Attribute

Questions of General Interest

16. E-R Diagrams were conceptualized by

(a) C J Date

(b) Edgar F Codd

(c) Peter Chen, 陳品山

(d) C J Date and Edgar F Codd

(e) Donald D Chamberlin and Raymond F Boyce

17. Relational Database *theory* was developed by

(a) C J Date

(b) Edgar F Codd

(c) Peter Chen

(d) C J Date and Edgar F Codd

(e) Donald D Chamberlin and Raymond F Boyce

18. Relational *Model* for database management was developed by

(a) C J Date

(b) Edgar F Codd

(c) Peter Chen

(d) Donald D Chamberlin and Raymond F Boyce

(e) Charles Bachman

19. `Structured Query Language (SQL)` was developed by
    (a) C J Date
    (b) Edgar F Codd
    (c) Peter Chen
    (d) C J Date and Edgar F Codd
    (e) Donald D Chamberlin and Raymond F Boyce

Written Answer

1. Why is it important to store a business rule only once in a central repository?

2. Give an example of a weak entity and its corresponding entity. Write the necessary business rule(s) and assumption(s) to support your answer.

3. Define a table, or a set of tables, such that you can give an example of a *required attribute*, corresponding *optional attribute*, an *atomic attribute* and a *derived attribute*. Write the necessary business rule(s) or assumptions to support your answer.

4. How is a composite attribute different from a multivalued attribute.
    . Give an example of each.
    . What is the benefit of using a multivalued attribute? What could be possible disadvantages?
    . What notation is used to identify a multivalued attribute?

5. In relation to compliance regulations how is *time-dependent data* associated with sustainability. Which aspect of sustainability, can you relate to this type of data?

ER Modeling

1. Table 10.1 below has a multi-valued attribute `Hobby`, the prime key is `RegID`. It is difficult to search a person's hobby, and to count the number of people who pursue a hobby. Split the table into two tables to make search and count easier. Identify the prime key of each table.

RegID	**FirstName**	**LastName**	**Hobby**
8421	Tiwah	Xuyac	Ice Hockey, Badminton
1721	Kozev	Colec	Travel, Reading
9523	Vemox	Lewor	Orchids, Bonsai
1556	Lucim	Caros	Badminton, Reading, Travel
1038	Feres	Kohul	Ice Hockey, Yoga
1152	Nucom	Kowom	Yoga, Skiing

Table 10.1: Name-Hobby

10.7 Summary

Data modeling is an important step in the development process. A sound data model supports data integrity. Writing clear, concise and unambiguous business rules is required for designing a database that is relevant to the organization. Although business rules change over time data models do not change at the same rate. This fact makes it imperative to put thought into developing a thorough data model.
There are several variations in ER diagram notations, we have adopted one variation - Crow's Feet.

10.8 Lab 10 - Create a Physical Data Model

Objective

1. Use a data modeling tool such as `pgmodeler` or Toad Data Modeler to create a Physical Data Model
2. Annotate and document the model
3. Export the data model to a `.png` or `.pdf` format

Submission

Upload the data model to LMS. A link is provided for upload.

Requirements

Use the model shown below as a reference when creating your model.

Procedure

Using the modeling software is simple. After knowing the features you need, you will be able to navigate your way in the button tile. Your first diagram will take a bit longer, subsequent ones will be quicker. Do not create the associative entity, choosing the `m:n` relationship and then converting this relationship will automatically create it. Use the `stamp` object to document the model.

Rubric - Physical ERD Winter 2021

Sr No	Requirement	Maximum	Earned
1	Text Box Inserted. Correct details added.	3	
2	Associative Entity correctly resolved from a M:M relationship.	3	
3	Attributes in Student and Course Entities defined.	3	
	Total	9	

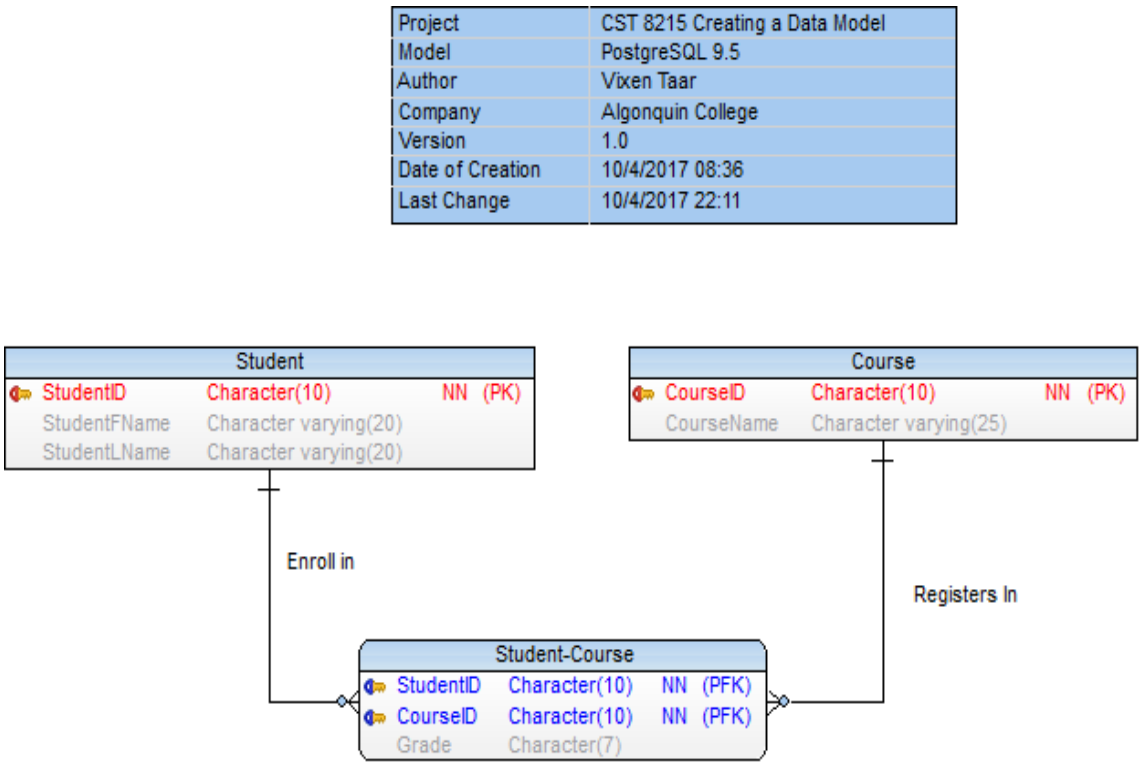


Figure 10.5: Physical ERD

11

SQL - Stored Procedure & Trigger

11.1 Introduction

This chapter covers Functions, Stored Procedures & Triggers.

11.2 Objective

- Determine the use and location of triggers and stored procedures. Differentiate stored procedures and triggers
- Differentiate between procedures and functions

11.3 Function

It is easy to write a function in `SQL`. Although they are restrictive, i.e. `SQL` functions cannot have decision statements or loops, `SQL` functions can be used within `SELECT` statements. They cannot call `UPDATE`, `DELETE` or `INSERT` statements. Functions can call other functions. They are stored a database objects.

Figure 11.1 illustrates a simple function titled `add_one`, it accepts a number adds one and returns the result. Note the single quotes before `BEGIN` and after `END;`. You do not have name the passed variable as `InValue`, the parameter is then referred as $1 instead of `InValue`.

```
DROP FUNCTION IF EXISTS add_one( InValue FLOAT8 );

CREATE OR REPLACE FUNCTION add_one( InValue FLOAT8 ) RETURNS FLOAT8
AS '
BEGIN
RETURN( InValue + 1 );
END; '
LANGUAGE 'plpgsql';
```

Figure 11.1: SQL Function

The statements below illustrate its usage:

```
SELECT add_one( 7 );

SELECT add_one( 4.8 + 5.3 );

SELECT * FROM Employee_T, add_one( Bonus_Percent );
```

`Bonus_Percent` is an attribute in table `Employee_T`.

11.4 Stored Procedure

Similar to a function, a stored procedure is an object that is stored in a database and is used by all other database objects. Stored Procedures includes functions. Unlike SQL queries that need to be interpreted each time it is run, a stored procedure is (*stored*) in the database at the server side and its code is optimized by the compiler. In addition, the concept of using stored procedures allows the administrator to permit access to authorized users to run these procedures Ref: [15] Chapter 10. Stored Procedures reside on the server side, not on the client side, adding to access control. Invoked from the client side, only the results are passed on to the caller, this reduces network traffic. Several applications can use a single stored procedure, standardizing processing rules.

Stored Procedures extend the capabilities of a DBMS. Associating a stored procedure to a trigger enables automatic execution of code when `INSERT`, `UPDATE` or `DELETE` operations are performed. The language used in postgres to write stored procedures is `plpgsql`, in addition, Java, C and Python are used to write stored procedures.

`plpgsql` is a block structured language. Variables declared in a block of code are visible only in the block. It is a strongly typed language, all variables are declared before they are used.

11.5 Trigger & Trigger Function

A trigger is an object associated with a table that runs a function when an `UPDATE`, `INSERT` or `DELETE` operation is performed on the table; i.e. trigger can be invoked when a data is changed. The function associated to a trigger must be written specifically for the trigger; it can call a general purpose function. A trigger must be associated to a table. A trigger function has access to data values *before* and *after* the data change event. These data values are used as `NEW.<attribute>` and `OLD.<attribute>`. `OLD` and `NEW` can be compared to determine if a change has taken place. A trigger function (although called a function) does not take any parameters nor does it return any parameters.

Production databases will have several triggers associated to tables. Some of the tasks they can perform are, data validation, audit, refresh `MATERIALIZED VIEW`s, send an email to a database administrator and other database maintenance tasks. A trigger can be invoked just once for all rows that have changed or for each row that has changed. Postgres allows only one trigger function to be associated to a trigger, if more than one function needs to run then multiple triggers must be created. Triggers associated to an event will run in alphabetical order. Data that has changed by an earlier trigger in the run order is available to subsequent triggers. A `ROLLBACK` on a later trigger will affect all previous triggers, they will also `ROLLBACK`. A trigger can be written in several languages, including `plpgSQL`, `Python` or `Perl`. *Caution:* Triggers can be used to enforce referential integrity constraints but this practice is not recommended, spurious use of triggers can degrade performance.

A trigger cannot be invoked from a `SELECT` statement; only from a data change operation, but a trigger can run a `SELECT` statement. Functions, stored procedures and trigger functions are stored in the database. Trigger functions are associated to triggers which in turn are associated to tables and are available only to tables even though they are stored in the database. Regular functions (i.e. non trigger functions) and stored procedures are also stored in the database but are not associated to a table, they are available to all objects in the database.

11.6 Comparison - Trigger, Function and Stored Procedure

The table below has been adapted from Database Processing, 40 Anniversary Edition, Kroenke et al., Page 378, Figure 7-33. It compares the three objects, Functions, Triggers and Stored Procedures.

	Function	**Trigger**	**Stored Procedure**
Accepts Parameters	Yes	No	Yes
Returns Result	Yes	No	Yes
Used in `SELECT` statement	Yes	No	No
Can use `SELECT` statememt	Yes	Yes	Yes
Uses `INSERT`, `UPDATE` and `DELETE` Statements	No	Yes	Yes
Can call a function	Yes	Yes	Yes
Can invoke a Trigger	No	Not Directly	Not Directly
Can invoke a stored procedure	No	Yes	Yes
Scope	as database object	specific to a table	as database object

Table 11.1: Comparison - Function, Trigger and Stored Procedure

11.7 Review Questions

Trigger & Stored Procedure

1. A named set of SQL statements that are executed when a data modification occurs are called:
   (a) Sub-queries
   (b) Stored Procedures
   (c) Triggers
   (d) PL/SQL

2. Identify the commands that need to be called explicitly, i.e. they cannot run automatically when a transaction has taken place
   (a) Sub-queries
   (b) Stored Procedures
   (c) Triggers
   (d) PL/SQL

3. Which one of the following returns values and take input parameters
   (a) procedures
   (b) functions

11.8 Summary

This chapter covered creation of simple functions in `SQL` and an overview of Stored Procedures and Triggers

11.9 Lab 11 - Trigger

Objective

1. Definition of a trigger
2. Purpose of a trigger
3. Creating and observing a trigger in action
4. Use of `SERIAL` as datatype

Reference

1. Read section 11.5 Trigger on page 165
2. `postgresql-9.5-us.pdf` - Section 9.27, Trigger Functions
3. http://www.postgresqltutorial.com/postgresql-triggers/ visit each of the links below and read the contents. Note: `TRUNCATE` trigger is specific to PostgreSQL.
   - Introduction to PostgreSQL trigger
   - Creating your first PostgreSQL trigger
   - Managing PostgreSQL triggers
4. `postgresql-9.5-us.pdf` - Section 9.27, 8.1.4 `SERIAL` Data Type

Submission

1. Complete the modifications to the trigger, submit the modified versions of `LearnTrigger-DDL.sql`, `LearnTrigger-DML.sql` and `LearnTrigger-Query.sql` file
2. Complete online quiz

Background

Read the explanation for trigger from the references given above. The trigger given the `LearnTrig-DDL.sql` file mainatins a log of all updates to the `Last_Name` in the `Employee` table. Study the DDL file and understand the structure of the code.

Observe the `DROP TRIGGER` and `DROP FUNCTION` statements at the beginning. A trigger is `DROPPED` before a `FUNCTION` is dropped.

The lab uses `SERIAL` data type in the employee table. You do not have to add an employee ID in the `INSERT` statement, postgreSQL automatically determines the last ID and updates the records.

Requirements

1. Create a database called `LearnTrigger`
2. Run the DDL file. It will create the tables and trigger.
3. Open the DML file and add about 10 more rows with employee names. Run the DML file. Correct and errors in your DML statements.

4. Run the statements one by one from `LearnTrigger-Query.sql` file. Convince yourself on the reason why some statements will not insert an entry in the `Employee_Audit` Table.
5. Modify the trigger in the same `DDL` file such that it adds a record to the `Employee_Audit` table each time the `First_Name` also changes. You will need to made appropriate changes to the Employee Audit table to store the `First_Name` and `Last_Name`
6. Write additional statements in the query file to test your conditions in the modified trigger. A sample file `LearnTrig-Query.sql` is provided to test the conditions.

Appendix A

Labs

A.1 Lab Submission Guidelines & Best Practices

Note: All items may not apply to every lab.

1. Save all `SQL` queries in a single, separate `.sql` file. `DDL` and `DML` statements should be in separate files.
2. Ensure commands in both files can be executed sequentially and independently. `DDL` statements should be executed first,the the `DML` statements.
   Aside: Keep the `.sql` file for revision (or study) during tests and exam.
3. Submit reverse, or forward, engineered ERD in `.png`, `.jpg` or `.pdf` format.
4. `CREATE VIEW` is a `DDL` statement, it should be in a `DDL` file. When you write a statement that will *use* a view, it is a `DML` statement. For example, `SELECT` statements that use a view should be in a query file, put these statements with the other queries you have written.
5. Business Rules and Abstract should be in `.docx`, `.doc` or `.rtf` format.
6. You have three attempts to upload to Brightspace, the last upload will be evaluated. If you upload a revised version, submit *all* files, not just the revised files.
7. Complete a quiz on the lab, if any.
8. Naming Convention: Name all files with eight character Algonquin email address, example `meaz0083-Inventory-DDL.sql`, for queries, `meaz0083-Inventory-ERD.pdf` for ERD, `meaz0083-Business Rules.sql` for Business Rules.
9. `JOIN` and `UNION` statemens should be with your queries, they are `DML` statements.
10. All explanations should be written in a separate `.txt` or `.rtf` file.

Best Practices

1. Files required for the course are available on `One Drive`, the link is posted in `Week 1` of LMS. Familiarize yourself with the folder organization.
2. Reserve your computer exclusively for school work. Do not use this computer for gaming.

3. Uninstall all games from your computer, many games interfere with the client server connection. Games have several processes in the background that affects the performance of your computer.
4. Uninstall all antivirus software from your computer, even if you have a paid version. Leave only Windows Defender running. `Kasperkey`, `McAfee`, `Norton Antivirus` will interfere with postgres installation.
5. You will need Win 10 on your computer to run `postgres` efficiently. Do not install postgres on `Windows 8`.
6. Do not click on a `.sql` file to open it. `postgres` is a client server application. `.sql` files do not behave the same way as a `.docx`, or `.xlsx` file.
7. Do not edit `.sql` files in another editor such as `Notepad++` or `Sublime`. Editing, testing and debugging `SQL` queries is easier in `pgAdmin`, it will save constant copy-paste from an external editor to `pgAdmin`.
8. Get familiar with Lynda.com.
9. After you install postgres do not move the directory from its installed location. For example, do not move it to the desktop or any other directory. Windows registry keeps track of the home directory, if the installed software directory is moved the association is broken.

A.2 Restoring Data from a Backup

A `.backup` file contains all constraints, data objects and data from a database. Data objects include `view`, `UDT`, `trigger` and others.

Procedure Restoring Data First create a database in `postgres`. Next, in `pgAdmin`, right click on the Database leaf, choose `Restore`. Select the file with extension `.backup`. This procedure creates tables with constraints and populates them with data.

Appendix B

Assignment

Assignment 1

Weight - 5%

Reference

1. Lab 4, page 68
2. Lab 6, page 73
3. Class Notes
4. Read the section titled Tips & Hints, written at the end of this assignment

Objective

1. You will need to combine Lab Inventory and Lab Country-City Database into a new database.
2. Add additional mandatory data to this new database.
3. Enforce constraints to maintain referential integrity.
4. Write queries using `JOIN`, subquery.
5. Write `VIEW`s.

Team Rules

One model per team. You may work alone or in a group of two students; a group of more than two students is not permitted. Select a team member from the *same* lab section.

Submission

Use time available during lab hours for clarification. Upload to LMS by due date. You have three attempts to upload to LMS, the last upload will be evaluated. If you upload a revised version, submit *all* files, not just the revised files. Submit a reverse engineered ERD diagram of the `InventoryII` database in `.jpeg` file or `.png` format. Save all `SQL` statements in a `.sql` file. Separate `DDL` and `DML` statements in two `.sql` files. Ensure commands in

both files can be executed sequentially and independently. During lab hours you need to give a demonstration to your instructor, the demonstration is worth marks. Do not submit zip files, upload individual files with its correct extension.

Note: `CREATE VIEW` is a DDL statement. All `SELECT` statements should be in a separate DML file without the `CREATE VIEW` statements.

Refer to the rubric to verify that you have met the requirements. For each query that you write mention your name and date of writing the statement. Each student in a team must submit the assignment with the required naming convention. Name all files with eight character Algonquin email address, example `meaz0083-zaem0794`, for first student, `zaem0794-meaz0083` for second student (if done in a team). Complete the quiz on the assignment.

Background

In Lab 4 you have modified the `DML` statement to populate the data. In this assignment you will modify the `DDL` statements, write the constraints, test and verify database. You will also reverse engineer the database.

Requirement

The trading company sells its products to customers in other countries, it wants to identify its customers by country. The company also purchases products from different countries and wants to keep track of country of origin of the product.

Modify the Database You need to modify the database and implement additional requirements. The new database is titled `InventoryII`. You are required to modify the two scripts (`DDL` and `DML`), run and test them. The DDL scripts should the tables. Write appropriate DML statements to add data. After you test the DDL statements, reverse engineer using a modeling tool such as `pgmodeler`. Rearrange the entities in the ER Diagram if required, marks are awarded for format and presentation. Add a textbox with your Name, Student Number, Section Number, Course Number and Semester.

Country Table Add a country table, call is `COUNTRY_T`. This has been implemented in Lab 6, page 73

City Table Use the same table as created in Lab 6. Retain the foreign key constraint on the `Cntry_Code` attribute. Reference it to `Cntry_Code` in the `COUNTRY_T` table.

Modify `Customer_T` table Add a column to the `Customer_T` table. Call it `Cust_Country`, use an appropriate type to accommodate the Country Code (`Cntry_Code`) from the `Country_T` table. Add a foreign key constraint to the `Customer_T` table, `Cust_Country` references `Cntry_Code` in the `Country_T` table. Use the same naming conventions for this constraint.

Modify `Product_T` table Add a column to the `Product_T` table, call it `Cntry_Origin`. This attribute indicates the country of origin of the product. Add a foreign key constraint to the `Product_T` table to reference `CntryCode` in the `Country_T` table.

Populate City and Country table Add about 10 countries and 20 cities. You may use the existing data from the world database. Add at least the following six countries in the `COUNTRY_T` table. The sample code should work in the table you create. All `INSERT` statements you write should have the names of attributes in it, as shown in the sample below. Modify all `INSERT` statements in the `Inventory-DML.sql` file.

```
INSERT INTO Country_T( Cntry_Code, Cntry_Name, Cntry_Population ) VALUES( 'RUS', 'Russian Federation', 144192450 );
INSERT INTO Country_T( Cntry_Code, Cntry_Name, Cntry_Population ) VALUES( 'MEX', 'Mexico', 119530753 );
INSERT INTO Country_T( Cntry_Code, Cntry_Name, Cntry_Population ) VALUES( 'CAN', 'Canada', 36155487 );
INSERT INTO Country_T( Cntry_Code, Cntry_Name, Cntry_Population ) VALUES( 'DZA', 'Algeria', 40400000 );
INSERT INTO Country_T( Cntry_Code, Cntry_Name, Cntry_Population ) VALUES( 'CHN', 'China', 1376049000 );
INSERT INTO Country_T( Cntry_Code, Cntry_Name, Cntry_Population ) VALUES( 'CHL', 'Chile', 18006407 );
```

Figure B.1: Data for `Country_T` table

Populate Customer and Product tables Modify the insert statements from Lab Inventory on page 68 to accommodate the new fields. Add at least one customer from Canada. Add atleast one product from Russian Federation and one product from Chile.

Add `City_T` and `Country_T` to the `DROP TABLE` list You should be able to run the DDL file multiple times. `Country_T` table is the last one to be dropped. `City_T` table should be dropped before the `Country_T` table. Change the order in the `DROP TABLE` list, observe the error message, determine the cause of the message. *Aside:* Tables need to be created in a certain order, foreign key constraints determine this. If the order of creation is changed, use `ALTER TABLE` to add constraints. Your database should have six tables, four tables from the Inventory Database and two tables from the `world` database.

Views Write two views of your choice. One dynamic view and one materialized view. Use the naming convention as ⟨ViewName⟩`_V`. `CREATE VIEW` is a `DDL` statement, it should be in the file which has DDL statements. Use `DROP VIEW IF EXISTS` in the DDL file *before* the `DROP TABLE` statements. Tables cannot be dropped if there are views that depend on them.

Mandatory Data Your DML file should contain the following statements. Place them in the appropriate sections of your code. Type in the code, **DO NOT** cut and paste from the PDF file.

```
-- mandatory data for Customer table
INSERT INTO Customer_T( Cust_ID, Cust_FName, Cust_LName, Cust_Phone, Cust_Address, Cust_City, Cust_Prov, Cust_PostCode, Cust_Country,
Cust_Balance ) VALUES( 'C097', 'Aze', 'Balai', '7-731-707-7243', 'Ulitsa Aleutskaya','Valdivostok', 'VL','VL7SK4', 'RUS', 0 );

-- mandatory data for Product table
INSERT INTO Product_T( Prod_Code, Prod_Description, Prod_InDate, Prod_QOH, Prod_Min, Cntry_Origin, Prod_Price, Prod_Discount )
VALUES( 'P2119', 'Organic Chard', '2018-01-14', 240, 100, 'RUS', 15.00, 5 );
INSERT INTO Product_T( Prod_Code, Prod_Description, Prod_InDate, Prod_QOH, Prod_Min, Cntry_Origin, Prod_Price, Prod_Discount )
VALUES( 'P2020', 'Organic Collard Green', '2018-01-14', 140, 110, 'RUS', 11.00, 5 );

-- mandatory data for Invoice table
INSERT INTO Invoice_T( Invoice_Number, Cust_Id, Invoice_Date ) VALUES( 'I88001', 'C097', '2018-01-15');


-- mandatory data for Invoice_Line Table

INSERT INTO Invoice_Line_T( Invoice_Number, Invoice_Line, Prod_Code, Line_Unit, Line_Price ) VALUES( 'I88001', 1, 'P2119', 3, 15.00 );

INSERT INTO Invoice_Line_T( Invoice_Number, Invoice_Line, Prod_Code, Line_Unit, Line_Price ) VALUES( 'I88001', 2, 'P2020', 3, 11.00 );
```

Figure B.2: Data for other tables

DELETE Statements After you have populated the `Customer_T` and `Product_T` tables, run the following `DELETE` statements. If there is an error, indicate the error number and error message. Explain in your own words the reason for failure.

```
DELETE FROM Customer_T WHERE Cust_Country = 'CHL';
DELETE FROM Product_T WHERE Cntry_Origin = 'RUS';
```

JOINS Write two joins. You may use `RIGHT JOIN` and/or `LEFT JOIN`

1. Write a `SQL` statement to list countries that do not have any Customers.
2. Write an `SQL` statement to list countries from which no products are bought by the company.
   Hint: `JOIN` `Product_T` table and `Country_T` table.

Refer page 96 on using a `JOIN` statement.

Tips & Hints

1. **Object Names Cannot be Duplicated** Ensure the constraint names for foreign keys are unique.
2. **Duplicate Values in Prime Keys are Not Permitted**

   Do not run the DML without first running the DDL. After adding insert statements to the DML file you will need to first run the DDL and then run the entire DML file. You can execute a single DML statement, if the prime key has not been entered before.
3. **Create prime key constraints before creating foreign key constraints**

   Foreign key constraints cannot be created unless a prime key is defined in the parent table.

Suggested Schedule

1 hour : Identify entities, relationships and attributes on paper. Prepare draft ERD on paper. Create Tables, write constraints, reverse engineer.

1 hour : Write queries and views. Test database, test queries.

1 hour : Refine and Submit.

Rubric - Assignment 01 Winter 2021

Sr No	Requirement	Maximum	Earned
1	Formatting & Presentation. Code formatted. Header and footer written in all `.sql` files.	3	
2	Tables first dropped and then created. DDL file can run without errors.	3	
3	Primary & Foreign Keys defined. ERD reverse engineered and exported to `.png` file. Textbox is created. Refer to sample ERD in figure B on page 178.	3	
4	Delete statements tried and explanation provided.	3	
5	`INSERT` and `SELECT` statements tested and functional.	3	
6	`VIEW`'s Created.	3	
7	`JOIN` statements tested and functional.	3	
8	Demonstration given.	3	
9	Bonus - Early submission.	3	
	(Maximum possible marks 24) **Total**	**24 (+3)**	

Table B.1: Assignment 1 Rubric

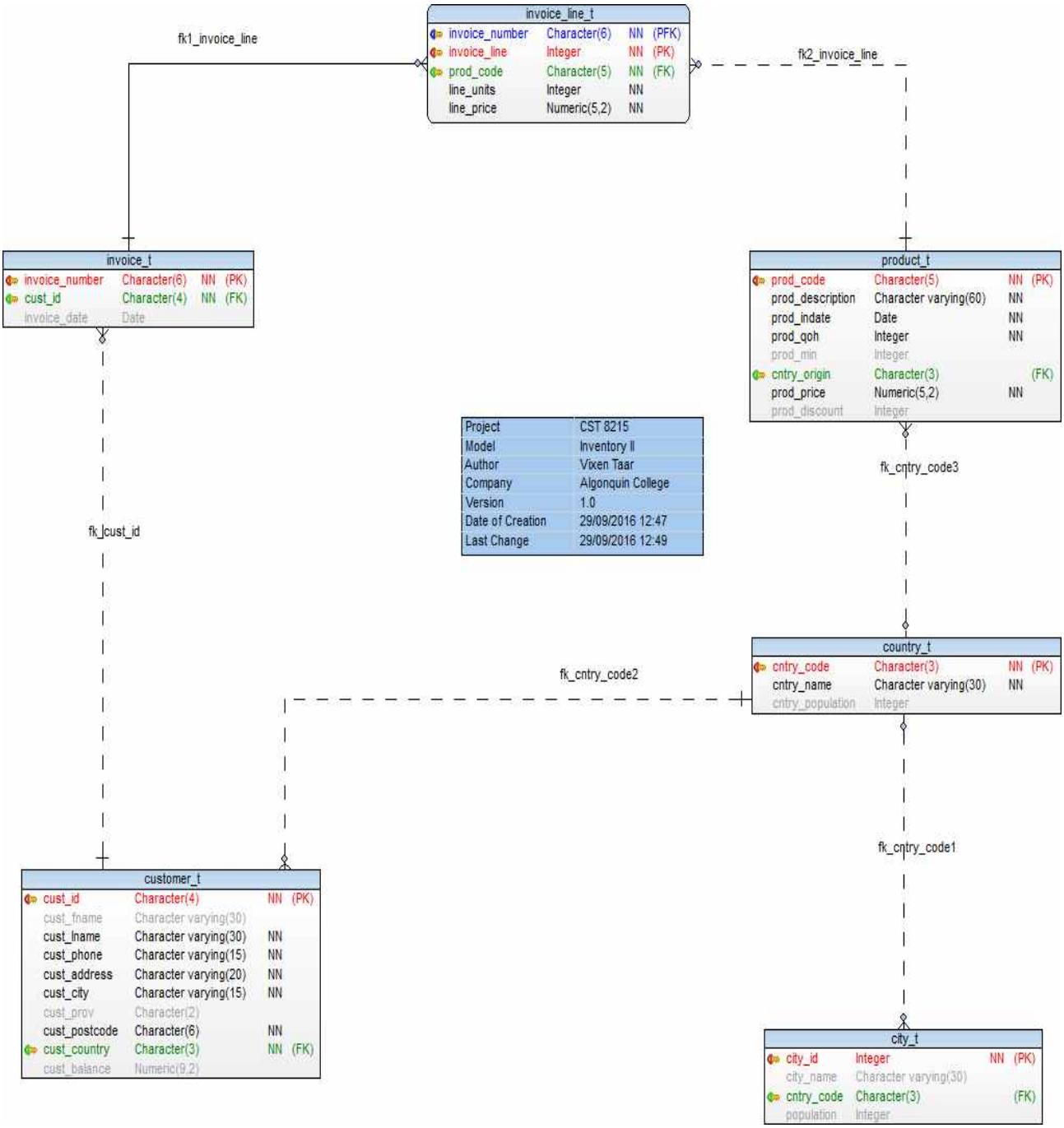


Figure B.3: Sample ERD with Textbox

Assignment 2

Objective

1. Model a database using ER diagram using available tools.
2. Design a database. Refine the model and implement it in PostgreSQL.
3. Reverse Engineer the database
4. Write business rules for your model.

Weight: Refer table on page **??**

Team Rules

You may work alone or in a group of two students; a group of more than two students is not permitted. Use time available during lab hours to work on the assignment. Indicate your team members to the instructor before the end of Week 8. Team members may be assigned by the lab instructor. The lab instructor may change your team member to facilitate progress in the assignment. Team members must be from the same lab section.

Submission

Each student must upload the assignment individually, even if done in a group. Upload `.dbm` file, `.png` file of Reverse Engineered ERD, `.sql` files with queries and `.doc` with business rules and metadata. You have unlimited attempts to upload, the last upload will be retained, all previous uploads will be overwritten. Refer page 171 for additional information on submission.

You are encouraged to consult with your lab instructor to validate your design and ER Model. Refer to the rubric to verify that you have met the requirements. Use file naming conventions as in Assignment 1. Write `DDL` and `DML` statements in separate files.

Requirements

Application Chose an application of your liking, a topic that interests you. Refer section 1.5 on page 3 for examples. You will need sufficient data to populate your database, therefore it is important that you choose an appropriate application.

Pencil first, computer next Draw an ER diagram using a clean sheet of paper, a pencil and eraser. Write (business) rules as you plan your database; aim to write at least 12 rules, upto a maximum of 20 rules. Create at least 5 entities, not including associative entities. Verify the ERD and business rules with your lab instructor on week 8, refer rubric. *Do not* draw the ER diagram using a diagramming tool on your computer and forward engineer.

One of the objectives of the assignment is to model a database that resembles to your experience in future work situations. Modelling game situations are discouraged; it is unlikely you will implement these types of databases in

your work place. Scenarios from PC games which include nouns such as *bomb*, *weapons* and verbs like *kill*, *destroy*, *death* or similar violent scenarios are **not** acceptable.

Code Documentation For each query, index or block of DML statement that you write mention your name and date of writing the statement. Do not write your name for each DML statement, only for a block of statements. If you are doing the assignment on your own you do not need to write your name for individual code blocks. Each `.sql` file should have a header and footer; refer example provided.

Clearly identify

1. Rules. Refer section D.2 on page 189.
2. Entities
3. Attributes for each entity and their types. Choose correct data types for each attribute such as `CHAR`, `INT`, `DATE`, `VARCHAR()`, `FLOAT`
4. Primary keys and foreign keys
5. Relationships and cardinality, `1:1`, `1:M`, `mandatory one`, `optional one`, `mandatory many`, `optional many`. Refer page 58 Ref[2]. You do not need to have all the above mentioned cardinalities in your model. The cardinality of `1:1` needs to be carefully examined, it should be collapsed into one entity for the purpose of this assignment. Use only one relationship between two entities.
6. Metadata. Tip: You may use the SQL command
   `SELECT * from information_schema.columns WHERE table_name = '<tablename>';`
   to get the metadata, then add additional details such as description. Refer table 1.1 for an example and page 3 of these notes for a short explanation on metadata.
7. Write at least 5 `INSERT` statements for each entity in your database. Write simple `SQL` statements using `SELECT` and `WHERE` for at least three entities. Do not write more than 20 INSERT statements for any one entity.
8. All entities should be related to each other by foreign keys. No entity should be isolated.
9. Write one `LEFT OUTER JOIN` and one `RIGHT OUTER JOIN` query of your choice. Document and justify your queries. The JOIN statements may include subqueries.
10. Write one `UNION` query.
11. `VIEW` - Write two views of your choice.
12. Reverse Engineer the database after creating the tables. Save to `.dbm`, export to `.pdf` or `.png`. Submit both files.

Procedure to build metadata Querying the `INFORMATION_SCHEMA` provides information about tables, views, columns and procedures in a database. The `SELECT` statement to query the `INFORMATION_SCHEMA` is shown in figure B.4.

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_NAME = 'author_t' OR
TABLE_NAME = 'publisher_t' OR
TABLE_NAME = 'borrower_t' OR
TABLE_NAME = 'inventory_t' OR
TABLE_NAME = 'book_t';
```

Figure B.4: `SELECT` statement to query the `INFORMATION_SCHEMA`

Note: Table names are case sensitive when querying the schema; use lower case for table names. Replace the asterisk `*` with `column_name, data_type, character_maximum_length` to get a three columns, get the result, and then add the two columns `Description` and `Source`.

Postgres gives you the details for all the attributes in the Author-Publisher schema. Export this result to the `.csv` (comma separated value) file. Use `File->Export`, choose column separator as a comma and quote char as double quote. This file can now be exported in a spreadsheet such as LibreOffice or Microsoft Excel. Remove all columns that are not required and format the spreadsheet data as shown in the example on page 4.

Suggested Schedule & Strategy:

1.5 hours: Identify entities, relationships and attributes on paper. Prepare draft ERD. Write business rules. Create tables with constraints. Reverse Engineer. Verify Abstract, Business Rules and ERD with Instructor, seek approval.

2 hours: Write DDL statements with PRIME KEY and FOREIGN KEY constraints. Reverse Engineer your model - verify. Do not write DML statement until you are convinced that your physical model agrees with the logical model and business rules. Write DML statements. Write queries and views. Test and refine database.

2 hours: Refine ERD, Submit.

Rubric

Sr No	Requirement	Maximum	Earned
1	Milestone reached. Business Rules agree with model. Verified with lab instructor.	3	
2	Milestone reached. Abstract written and verified with lab instructor.	3	
3	Milestone reached. ER Diagram drawn on paper. Verified with lab instructor.	3	
4	Formatting & Presentation. Header and Footer placed in each file	3	
5	Primary & Foreign Keys defined. Minimum 5 Entities defined.	3	
6	Relationships/Associative Entities and Cardinality defined.	3	
7	Metadata created.	3	
8	`INSERT` and `SELECT` with `WHERE` statements tested and functional.	3	
9	`LEFT` and `RIGHT OUTER JOINS` written and tested.	3	
10	Minimum one RANGE or CHECK constraint defined. Views created and tested.	3	
11	Subqueries written and tested.	3	
12	Reverese Engineered diagram in the diagramming tools' file format and `.png`. ERD should have a textbox with the correct details.	3	
13	Demonstration Given.	3	
14	Early submission bonus.	(3)	
	Total (Maximum)	**39**	

Projects for Assignment 2

As part of the sustainability initiative by the provincial government you are encouraged to think about sustainability - social, economic and environmental. The projects on *Civilization* and *UNESCO* address social and environment sustainability. A few projects are listed here to start help you select a project. You are encouraged to seek your own projects. Do not select projects from samples used in this workbook.

Civilization Build a database on worlds civilizations. The following link https://en.wikipedia.org/wiki/Civilization could be a starting point to for your research.

World Heritage Sites United Nations Educational, Scientific and Cultural Organization (UNESCO), as the name suggests, this UN branch coordinates international cooperation in education, science and culture. Read about the organization at https://en.unesco.org/. Canada has several world heritage sites, you may chose to build a database on these sites or sites from several other countries. Tip: explore latitude and longitude data types in Postgres if you wish to plot the locations of these sites. Alternatively chose other UNESCO initiatives that interest you.

Arts Build a database on a list of famous (music) composers and their works, select composers of at least three continents. *Suggestion:* Categorize music works into its types, for example in western music - sonata, concerto, symphony. Alternatively, build a database on famous sculptures, paintings or other fine arts of your choice. Add data from several different cultures.

Employee-Department Build a database of Employee, Department and projects undertaken by employees and departments. You may chose to include location of project sites, department location, project budget.

Do not use the following models for your assignment

1. Author-Publisher-Borrower-Book
2. Mechanic-Garage-Vehicle
3. Game development on different platforms
4. Course-Student-Professor
5. Product-Supplier-Customer-Invoice
6. Hotel Booking Database

Appendix C

Practical Exam - Winter 2021

Abstract A dental clinic maintains records of patient appointments, visits and the dental procedure carried out during the visit. After the visit the system bills the patient. A patient may have insurance coverage, which can be from more than one insurance company. A patient's insurance may cover all or a portion of the cost. A dental procedure may be carried out by a dentist, dental hygienist or a surgeon.

Business Rules

- **Patient**

1. A patient may be an individual. A patients' family members may also be patients in the same clinic.
2. A patients home address, phone number, email address and employer are maintained, among other details.

- **Dental Service**

3. A list of standard procedures is maintained in the database.
4. Dental Procedures are taken from the document titled `Ontario Schedule of Dental Services.pdf` pages 26ff. Dental Services are divided into Diagnostic, Preventive, Restorative, Endodontic (treatment of dental pulp), Periodontal (supporting structures of teeth), Prosthetics Removable (replacing missing tooth), Prosthodontic Fixed.
5. Some Preventive services have time units associated with them. For example, for cleaning one unit of time is 15 minutes.
6. Each tooth is numbered, this system follows the Canadian tooth numbering system.

- **Appointment**

7. A patient may book one or several appointments. More than one appointment may be booked for a procedure that takes several visits.
8. It is possible that a patient does not have any current appointments on record.
9. An appointment has patient details and the primary dentist attending the patient at that time.

- **Insurance**

10. A list of insurance companies is maintained.
11. A patient may not have any insurance, in this case the patient pays the entire amount on his own.
12. A patient may have one or more insurance companies that pays for the dental service.

13. A patient may be covered fully or partially by the insurance company.
14. The term co-payment is used when part of the payment is made by the insurance company and the balance is paid by the patient.

- **Visit**

15. After an appointment is confirmed, a patient will visit the dental office, the date, patient id and dentist id are recorded for the visit.
16. Each procedure performed on the patient during the visit is recorded
17. Cost of each procedure is the standard cost from the Procedure database.

- **Procedure**

18. Each procedure is specified by the Ontario Dental Association.

- **Billing**

19. After each visit the payment for the service is determined.
20. After the insurance coverage, if any, is determined the balance, if any, is debited to the patient.
21. The balance is collected from the patient account.
22. An invoicing system is maintained.
23. No tax is collected for dental services.

Exam Rules and Instructions

1. Open book exam. You may use your notes, textbook and online resources.
2. You must not share any material with your neighbour or talk to any other student.
3. You must not use a cell phone or your computer to email, to text or talk to another person during the test.
4. Do not leave your cell phone on the desk, do not use your cell phone to keep time.
5. Write your Last Name, First Name, Student Number and section number on this exam question paper. You name should be in the same order as in the college register and attendance sheet. Your section number is on the first page of the attendance sheet.

Preparation The following URL provides some background on the dentists office.
https://www.youroralhealth.ca/dental-procedures98/common-dental-procedures
https://www.cda-adc.ca/en/oral_health/talk/.

Create a database titled `DentalHealth` in postgres. Restore the data from the backup file `Dentist.backup`. Instructions on restoring data from a `.backup` file are given in section A.2 on page 172. The database has (atleast) 8 tables. Reverse engineer the database, study the ER diagram and familiarize yourself with the tables and relationships. In the Practical Exam you will answer a set of (about) 20 questions. You may use Postgres to test and verify the statements before submitting them to LMS. Questions in the test bank will have different values for each group of lab students.

Submission

1. Complete the online exam on LMS.
2. Write the answers to written questions on paper and submit the paper to your instructor.
3. Write the `INSERT`, `DELETE` and `UPDATE` statements, if any, in a file titled with your login id and `-DDL.sql`, as you have done in your assignment.

Requirement For the exam you will need

- Your laptop with charger
- Postgres and pgmodeler installed
- Sharpened pencils, eraser and sharpener
- Pen
- ERD for the database

Sample Questions & Review Questions

1. Write a query that will give the average amount charged for visits.
2. Write a query to list the procedure or procedures that have the highest cost.
3. Write a query to list the procedure or procedures that have the lowest cost.
4. Practice using a `LEFT JOIN` and `RIGHT JOIN`. List procedures that have not been performed till date.
5. List the dentists that are in the database, but have not seen a patient. Filter the rows using a date, say, dentists that have not seen a patient in the past 30 days or one month.
6. Practice adding data to each of the tables.
7. Practice using the `UPDATE` statement to change data in the tables.
8. You may be given SQL statements that have an error introduced in it, you should be able to interpret the postgres error message correct the error and run the statement.
9. Practice recognizing statement that will violate referential integrity constraints.
10. Identify the relation that has a unary relationship (if any).
11. Use `ALTER TABLE` to existing tables.
12. Insert Primary key and foreign key constraints to new tables to the existing database.
13. Add data to existing tables.

Appendix D

Database Files

D.1 File List

Study and run these files, they will consolidate your understanding of database topics.

1. `NATURAL JOIN`
2. `DECK` - demonstrates `CARTESIAN` product. Also (incorrectly) called `CROSS JOIN`
3. `SELF JOIN`
4. corelated and non corelated queries
5. Planet Statellite Database

D.2 Abstract

Assignment 2 requires you to write an *abstract*. What is an Abstract? An abstract tells the reader what you plan to do for your project, what the document (project) is about, what can the reader expect from the document. It is about 1/3 to 1/2 page long, Times Roman 12 point (or similar proportional spacing font with serif), single line spacing. Mention any tools you intend to use, the problem statement and your proposed solution. Remember, this is a draft and may change (albeit slightly) before you submit your document. Do not use casual, spoken phrases (or language) in the abstract - your approach should be formal. An abstract is often called an Executive Summary. After you submit your abstract, you and the instructor will agree on the scope of the project.

D.3 Business Rules

In addition to an abstract you are required to write business rules for assignment 2. To model data an analyst needs to document rules and policies of an organization. These rules state how data will be created, modified (updated) and deleted (removed). Rules are not universal, each organization defines its own rules. As an example, consider the number of wireless service providers in the city, the way each company makes its offer to its customers, the different usage plans, payment options and contracts; these differ albeit slightly from one company to another.

Rules change over time. As a data analyst your task is to identify and understand rules (and policies) that govern the organizations data.

Business Rules are independent of the tools (technology) used to implement the system. For instance, you may be using PostgreSQL and a data modeling tool (eg. pgmodeler) to implement the system, these tools do not influence the narration of business rules. Your focus should be on defining the policies and constraints of the business. Refer [12]. Business rules are stored in a *Central Repository* and shared throughout the organization.

Atomicity, consistency, clarity and non-redudancy of Business rules are written, validated and enforced using software. Business Rule Management System (BRMS) is the category of software; an example of BRMS is a *ILOG* a company owned by IBM.

Characteristics of a Business Rule are listed below. Refer [3] Page 61.

Characteristic	**Description**
Atomic	A rule should specify only one idea in a statement. The term *Atomic* implies that the item cannot be divided.
Consistent	A rule cannot contradict another rule. It cannot contain conflicting statements.
Clear	A rule must be stated such that it cannot be misinterpreted; a rule must have only one interpretation.
Distinct	A business rule cannot be redundant, i.e. it must be stated only once. A business rule may refer to *Terms of Reference* and *Definitions* with the document.
Business Oriented	A rule is stated in common business language that is understood by *all* stakeholders.

Table D.1:

Appendix E

Navigating PostgreSQL

This page has been initiated by database students of Fall 2016. The following students have contributed to this page - Steven Adema,

E.1 Keyboard Shortcut

E.1.1 Edit

1. Comment a block of code `Ctrl-K`. Insert two dashes - as the first two characters on the line.
2. Uncomment a block of code `Ctrl-Shift-K`
3. Delete a line `Ctrl-L`
4. Duplicate a line `Ctrl-D`

E.1.2 Query

1. CTRL-E to open a query window. Or, click on the `SQL` icon
2. Run (Execute) a query `F5`

E.2 Metadata

1. Show all information of a table
```
SELECT *
FROM Information_Schema.Columns
WHERE Table_Name = 'country';
```
2. Show the column names of a table
```
SELECT Column_Name
FROM Information_Schema.Columns
WHERE Table_Name = 'country';
```

3. Show all objects in the database
   ```
   SELECT *
   FROM Information_Schema.Columns
   WHERE Table_Schema = 'public';
   ```

E.3 Misc

1. to insert an apostrophe as a literal enclose the data item in double dollar signs. For example to add `St John's` as city name, use `$$St John's$$`.

E.4 psql

`psql` can be invoked through `pgAdminIII` or from the Windows Command Prompt, i.e. by first running `cmd`. From `pgAdminIII` select a database, next from the Plugins drop down menu chose `PSQL Console`, you will get a windows shell. Alternatively run the Command Prompt and run the following commands. `C\` is the default postgres installation directory.

- Change the directory, `cd "C:\Program Files\PostgreSQL\9.5\bin"`.
- for additional commands use `psql --help`
- `psql -h localhost -p 5432 -U postgres -d Inventory`.
  `Inventory` is the database,
  `localhost` is the host,
  `5432` is the port number.
- To run a SQL script use:
  `psql -f "c:\Test.sql" -d Inventory -h localhost -p 5432 -U postgres -a`
  `Test.sql` is the `SQL` script.

A few useful psql commands are shown below: `\?` + ENTER KEY (backslash-question mark + ENTER)- to get a list of commands in `psql` `\c` ⟨dbname⟩ dbname [username]; connect to database `\l` list all databases on server `\dt` list all tables in the current database `SHOW DATA_DIRECTORY`;

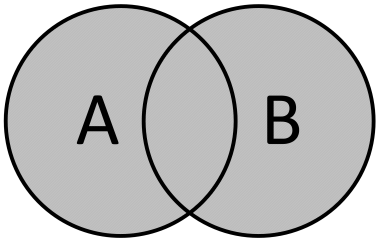
E.4.1 pg_dump

`pg_dump` is a utility to extract data into a file. Refer documentation for details. Here is one example, it writes the contents of the `city` table from the `world` database to a file titled `d:\INSERT_CITY.sql`. Without the `-inserts` option it will use the `COPY` option.

```
pg_dump -h localhost -U postgres -p 5432 -d world -t city --inserts -f d:\INSERT_CITY.sql
```

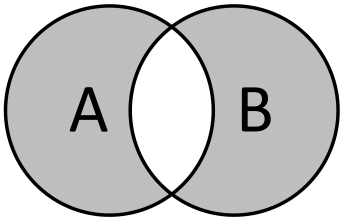
E.5 Venn Diagram

Ian Glas has prepared these diagrams.



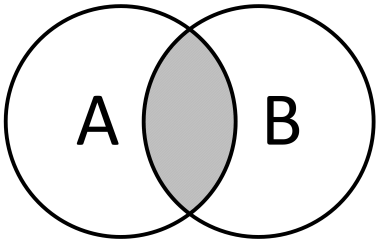

```
SELECT <list>
FROM tableA A
FULL OUTER tableB B
ON A.key = B.key
```

Figure E.1: Full Outer Join



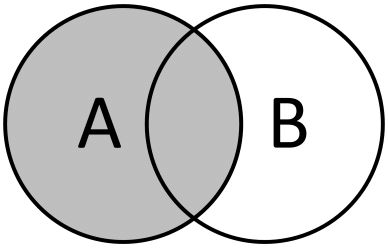

```
SELECT <list>
FROM tableA A
FULL OUTER JOIN tableB B
ON A.key = B.key
```

Figure E.2: Full Outer Join using `NULL`



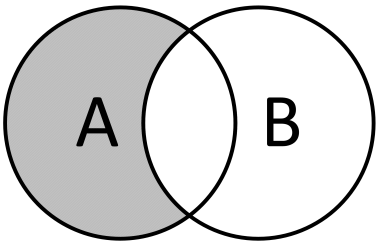

```
SELECT <list>
FROM tableA A
INNER JOIN tableB B
ON A.key = B.key
```

Figure E.3: Inner Join




```
SELECT <list>
FROM tableA A
LEFT JOIN tableB B
ON A.key = B.key
```

Figure E.4: Left Join




```
SELECT <list>
FROM tableA A
LEFT JOIN tableB B
ON A.key = B.key
WHERE B.key IS NULL
```

Figure E.5: Left Join using `NULL`

E.6 Link

A good resource for PostgreSQL tutorial http://www.postgresqltutorial.com/ https://www.techonthenet.com/postgresql/index.php

E.7 Tips and Traps

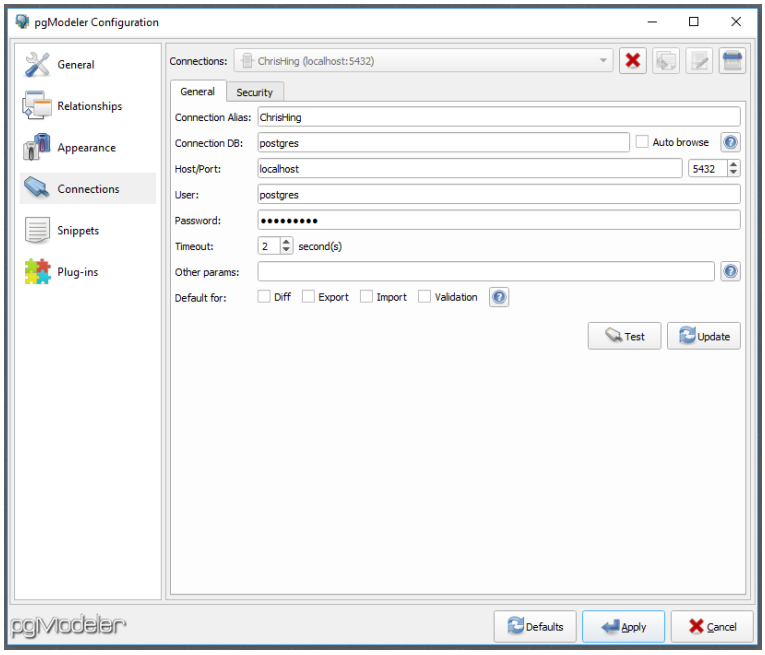
`CREATE OR REPLACE TABLE` is not permitted. You have to first `DROP` the table and then `CREATE` it. Similarly `CREATE OR REPLACE TRIGGER` is not permitted. There is no method to accurately identify all dependencies of a `TABLE`.

Appendix F

pgmodeler Configuration

F.1 Establishing a connection

pgmodeler is a client that connects to Postgres server. To establish the connection use the parameters below. Go to Settings->Connections. Click on the New Connection icon and enter the parameters below. Use the same password that you used when installing Postgres. Test the connection. After a successful connection, select Add. Now you have the parameters saved.



F.2 Naming a database

Database names must be contiguous, i.e. no space between words,
i.e. Use DentalClinic instead of Dental Clinic.
Although Postgres will accept the name, pgmodeler will not accept this naming convention when importing file.

Bibliography

[1] C J Date, An Introduction to Database Systems, 8e, Pearson Education Inc., 2004, 978-0-3211-9784-9

[2] Hoffer J H, Ramesh V, Keikki T, Modern Database Management, 11e, Pearson Education Inc., 2004, 978-0-13-266225-3

[3] Hoffer J H, Ramesh V, Keikki T, Modern Database Management, 12e, Pearson Education Inc., 2016, 978-0-13-354461-9

[4] Chris Fehily, Visual Quickstart Guide, SQL, 3e, Pearson Education Inc., 2008, 978-0-321-55357-3

[5] http://en.wikipedia.org/wiki/Database_normalization

[6] http://en.wikipedia.org/wiki/Relational_database

[7] Paul DuBois, MySQL, 5e, Addison Wesley, 2013, 978-0-321-83387-7

[8] T Connolly, C Begg, Database Systems, 6e, Pearson Education Inc., 2010, 978-0-13-294326-0

[9] Oracle Database, SQL Language Reference, 11g Release 1 (11.1) August 2010, Primary Author: Diana Lorentz

[10] C J Date, An Introduction to Database Systems, 5e, Addison Wesley Pub. Co., 1990, 0-201-52878-9

[11] D M Kroenke. et al, Database Processing, 14e, Pearson, 2016, 978-0-13-387670-3

[12] https://adtmag.com/Articles/2001/06/05/Turning-rules-into-requirements.aspx?Page=1

[13] Regina Obe & Leo Hsu, PostgreSQL Up & Running, 3e, O'Reilly, 2018, 978-1-491-96341-8

[14] Young, Sue F. & Wilson, Robert J., The ICE Approach, Portage & Main Press, 2000, 978-1-894-11064-8

[15] Matthew, Neil & Stones Richard, Beginning Databases with PostgreSQL, 2e, APress, 2005, 978-1-59059-478-0

[16] van der Lans Rick F., Introduction to SQL, 4e, Addison Wesley, 2007 20th Anniversary Edition, 978-0-321-30596-1

Glossary

abstraction A technique that hides details of an underlying system, but still allows development based on the amount of information provided. Abstraction is done to control complexity of software systems. 4

anomaly In database context, an anomaly is an inconsistency. A transaction that will make the data inaccurate, unreliable and not dependable. 111

architecture Art and science of designing and managing a system (a database system). A specific model. 10

concatenate Attach together, link together. Attach two character strings together. 71

constraint Restriction. A method, command or phrase that maintains database integrity. 4

cumbersome slow, complicated, inefficient, difficult to use. 75

default A value chosen by the system when a user does not provide it. For example, if no date is entered by the user the *default* is today's date. Default is also used in a setting, for example during OS startup or program launch. *Aside:* This term has other meanings in finance and law. We shall restrict the meaning to computing. 20

expression a code fragment that returns a value. 31

LMS Learning Management System, currently Brightspace at Algonquin College. 16

mandatory Required, obligatory, not optional, compulsory. 60

operand a value that is used by an operator. For example, in the arithmetic operation `2 + 3`, 2 and 3 are operands, `+` is the operator. 25

operator A symbol or term that performs an arithmetic or logical operation. For example, `+`, `-` perform arithmetic operations. `NOT`, `AND`, `OR`, `XOR` are logical operators. The operation is performed on operands. *Usually* there are two operands. The logical `NOT` operator has only one operand. 22, 25

parse To resolve (split) a statement into its smaller components with the intention to analyse, interpret or describe it. This term is used in grammer, SQL is a (computer) language, this word works well for our purpose. 17

platform An operating system or environment such as a database, a computer or microprocessor, used to describe an environment for running other software, or for defining a software or hardware environment. For example (i) This SQL script runs on PostgreSQL platform. (ii) This program compiles on Mac OSX platform. 183

syntax rules used to construct expressions and statements in a language. Aside: `SQL` is a computer language, hence a syntax. 30

tuple In relational database, a tuple is a row. Often the term record is used interchangably with row. The term *tuple* is used formally during database design. 112

Index

Notes