Introduction to Oracle 11g SQL Programming

Student Workbook

itcourseware

Training Materials For IT Professionals

INTRODUCTION TO ORACLE 11G SQL PROGRAMMING

Contributing Authors: Danielle Hopkins, John McAlister, and Rob Roselius

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editor: Jan Waleri

Editorial Assistant: Danielle North

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Elizabeth Boss, Denise Geller, Jennifer James, Julie Johnson, Roger Jones, Joe McGlynn, Jim McNally, and Kevin Smith, and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

Chapter 1 - Course Introduction 7

Course Objectives 8
Course Overview 10
Using the Workbook 11
Suggested References 12

Chapter 2 - Relational Database and SQL Overview 15

Review of Relational Database Terminology 16
Relational Database Management Systems 18
Introduction to SQL 20
Oracle Versioning and History 22
Logical and Physical Storage Structures 24
Connecting to a SQL Database 26
Datatypes 28
Sample Database 30

Chapter 3 - Using Oracle SQL*Plus 35

SQL*Plus 36
The SQL Buffer 38
Buffer Manipulation Commands 40
Running SQL*Plus Scripts 42
Tailoring Your SQL*Plus Environment 44
Viewing Table Characteristics 46
SQL*Plus Substitution Variables 48
Interactive SQL*Plus Scripts 50
SQL*Plus LOB Support 52
Graphical Clients 54
Labs 56

Chapter 4 - SQL Queries — The SELECT Statement 59

The SELECT Statement 60
The CASE...WHEN Expression 62
Choosing Rows with the WHERE Clause 64

NULL Values 66
Compound Expressions 68
IN and BETWEEN 70
Pattern Matching: LIKE and REGEXP_LIKE 72
Creating Some Order 74
Labs 76

Chapter 5 - Scalar Functions 79

SQL Functions 80
Using SQL Functions 82
String Functions 84
Regular Expression Functions 86
Numeric Functions 88
Date Functions 90
Date Formats 92
Conversion Functions 94
Literal Values 96
Intervals 98
Oracle Pseudocolumns 100
Labs 102

Chapter 6 - SQL Queries — Joins 105

Selecting from Multiple Tables 106
Joining Tables 108
Self Joins 110
Outer Joins 112
Labs 114

Chapter 7 - Aggregate Functions and Advanced Techniques 117

Subqueries 118
Correlated Subqueries 120
The EXISTS Operator 122
The Aggregate Functions 124
Nulls and DISTINCT 126
Grouping Rows 128
Combining SELECT Statements 130
Labs 132

Chapter 8 - Data Manipulation and Transactions 135

The INSERT Statement 136
The UPDATE Statement 138
The DELETE Statement 140
Transaction Management 142
Concurrency 144
Explicit Locking 146
Data Inconsistencies 148
Loading Tables From External Sources 150
Labs 152

Chapter 9 - Data Definition and Control Statements 155

Datatypes 156
Defining Tables 158
Virtual Columns 160
Constraints 162
Inline Constraints 164
Modifying Table Definitions 166
Deleting a Table Definition 168
Controlling Access to Your Tables 170
Labs 172

Chapter 10 - Other Database Objects 175

Views 176
Creating Views 178
Updatable Views 180
Sequences 182
Synonyms 184
Labs 186

Appendix A - The Data Dictionary 189

Introducing the Data Dictionary 190
DBA, ALL, and USER Data Dictionary Views 192
Some Useful Data Dictionary Queries 194

Solutions 197

Index 215

Chapter 1 - Course Introduction

Course Objectives

- Describe the features of a Relational Database.
- Interact with a Relational Database Management System.
- Use SQL*Plus to connect to an Oracle database and submit SQL statements.
- Write SQL queries.
- Use SQL functions.
- Use a query to join together data items from multiple tables.
- Write nested queries.
- Perform summary analysis of data in a query.
- Add, change, and remove data in a database.
- Manage database transactions.
- Work in a multi-user database environment.
- Create and manage tables and other database objects.
- Control access to data.

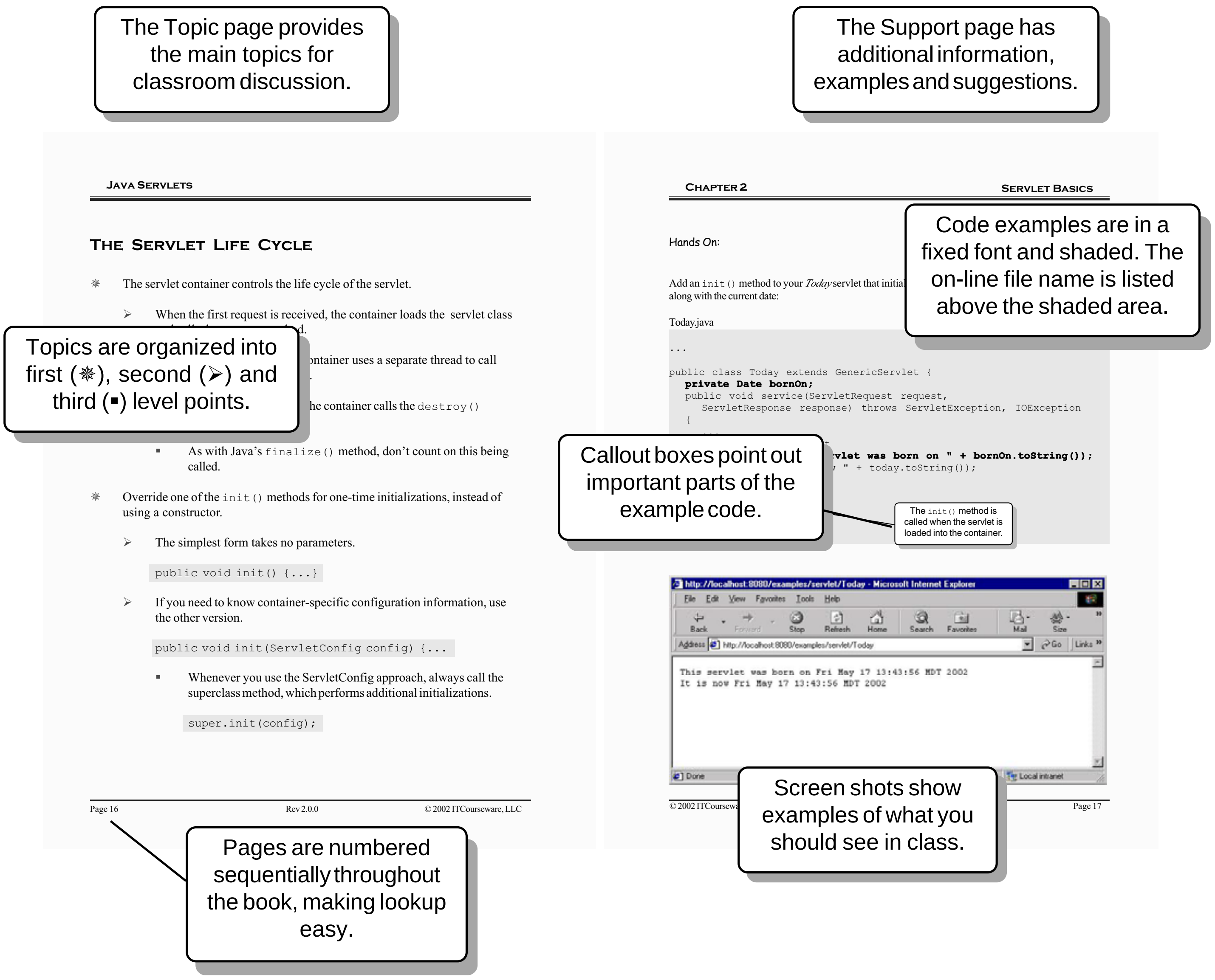
COURSE OVERVIEW

- **Audience:** This course is designed for database application developers.
- **Prerequisites:** Familiarity with relational database concepts.
- **Student Materials:**
  - Student workbook
- **Classroom Environment:**
  - One workstation per student
  - Oracle 11*g*

Using the Workbook

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. **Hands On** sections provide opportunities for practical application of key concepts. **Try It** and **Investigate** sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.



The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

Java Servlets

The Servlet Life Cycle

✵ The servlet container controls the life cycle of the servlet.

➢ When the first request is received, the container loads the servlet class

Topics are organized into first (✵), second (➢) and third (▪) level points.

ontainer uses a separate thread to call

he container calls the `destroy()`

▪ As with Java's `finalize()` method, don't count on this being called.

✵ Override one of the `init()` methods for one-time initializations, instead of using a constructor.

➢ The simplest form takes no parameters.

```
public void init() {...}
```

➢ If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) {...
```

▪ Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

Pages are numbered sequentially throughout the book, making lookup easy.

Chapter 2 Servlet Basics

Hands On:

Add an `init()` method to your *Today* servlet that initial along with the current date:

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Today.java

```
...

public class Today extends GenericServlet {
  private Date bornOn;
  public void service(ServletRequest request,
     ServletResponse response) throws ServletException, IOException
  {
```

```
vlet was born on " + bornOn.toString());
 " + today.toString());
```

Callout boxes point out important parts of the example code.

The `init()` method is called when the servlet is loaded into the container.

http://localhost:8080/examples/servlet/Today - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Mail Size
Address http://localhost:8080/examples/servlet/Today
Go Links
Done
Local intranet

```
This servlet was born on Fri May 17 13:43:56 MDT 2002
It is now Fri May 17 13:43:56 MDT 2002
```

Screen shots show examples of what you should see in class.

© 2002 ITCoursew Page 17

SUGGESTED REFERENCES

Beaulieu, Alan. 2009. ***Learning SQL***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596520823

Celko, Joe. 2010. ***Joe Celko's SQL for Smarties, Fourth Edition: Advanced SQL Programming***. Academic Press/ Morgan Kaufman, San Francisco, CA. ISBN 0123820227

Celko, Joe. 2006. ***Joe Celko's SQL Puzzles and Answers***. Morgan Kaufmann, San Francisco,CA. ISBN 0123735963

Churcher, Clare. 2008. ***Beginning SQL Queries: From Novice to Professional***. Apress, Inc., Berkeley, CA. ISBN 9781590599433

Date, C.J. and Hugh Darwen. 1996. ***A Guide to The SQL Standard, Fourth Edition***. Addison-Wesley, Reading, MA. ISBN 0201964260

Date, C. J.. 2011. ***SQL and Relational Theory, Second Edition: How to Write Accurate SQL Code***. O'Reilly Media, Sebastopol, CA. ISBN 1449316409

Gennick, Jonathan. 2004. ***Oracle Sql*Plus Pocket Reference, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596008856

Gruber, Martin. 2000. ***SQL Instant Reference, Second Edition***. SYBEX, Alameda, CA. ISBN 0782125395

Kline, Kevin. 2009. ***SQL in a Nutshell, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596518847

Kreines, David. 2003. ***Oracle Data Dictionary Pocket Reference***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005172

Loney, Kevin. 2008. ***Oracle Database 11g: The Complete Reference***. McGraw-Hill Osborne Media, ISBN 0071598758

Mishra, Sanjay. 2004. ***Mastering Oracle SQL, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596006322

http://tahiti.oracle.com
http://www.dbasupport.com
http://www.hot-oracle.com
http://www.oracle.com
http://www.searchdatabase.com
http://www.toadworld.com

Your single most important reference is the SQL Reference book, which is part of the Oracle Database Online Documentation. You may have received this on CD-ROM with your Oracle distribution. If not, you can access it online at Oracle's web site. This is the official, complete description of Oracle's implementation of SQL. It includes many examples and discussions.

An easy way to find it is to go to:

http://tahiti.oracle.com/

Find the documentation for your version of Oracle. Locate the SQL Reference, and open the HTML table of contents.

If you have web access in the classroom, open a browser now and find the SQL Reference. Set a browser bookmark, and have the SQL Reference at hand throughout this class.

CHAPTER 2 - RELATIONAL DATABASE AND SQL OVERVIEW

OBJECTIVES

- ❈ Describe the features of a Relational Database.
- ❈ Describe the features of a Relational Database Management System.
- ❈ Work with the standard Oracle datatypes.
- ❈ Review Oracle history and versions.
- ❈ Distinguish between a database server program and a client application program.
- ❈ Connect to and disconnect from a database.

REVIEW OF RELATIONAL DATABASE TERMINOLOGY

- Relational Databases:
  - A *Relational Database* consists of *tables*, each with a specific name.
  - A table is organized in *columns*, each with a specific name and each capable of storing a specific *datatype*.
  - A *row* is a distinct set of values, one value for each column (although a column value might be empty (*null*) for a particular row).
  - Each table can have a *primary key*, consisting of one or more columns.
    - The set of values in the primary key column or columns must be unique and not null for each row.
  - One table might contain a column or columns that correspond to the primary key or unique key of another table; this is called a *foreign key*.

A *Relational Database* (RDB) is a database which conforms to Foundation Rules defined by Dr. E. F. Codd. It is a particular method of organizing information.

A *Relational Database Management System* (RDBMS) is a software system that allows you to create and manage a Relational Database. Minimum requirements for such a system are defined by both ANSI and ISO. The most recent standard is named *SQL2*, since most of the standard simply defines the language (SQL) used to create and manage such a database and its data. Some people use the term *SQL Database* as a synonym for *Relational Database*.

Each row (sometimes called a *record*) represents a single entity in the real world. Each column (sometimes called a *field*) in that row represents an attribute of that entity.

Entity Relationship Modeling is the process of deciding which attributes of which entities you will store in your database, and how different entities are related to one another.

The formal word for row is *tuple*; that is, each row in a table that has three columns might be called a *triple* (a set of three attribute values); five columns, a *quintuple*; eight columns, an *octuple*; or, in general, however many attributes describe an entity of some sort, the set of column values in a row that represents one such entity is a tuple. The formal word for column is *attribute*.

RELATIONAL DATABASE MANAGEMENT SYSTEMS

- A *Relational Database Management System* (RDBMS) provides for users.
  - Each *user* is identified by an account name.
    - A user can access data and create database objects based on privileges granted by the database administrator.
  - Users own the tables they create; the set of tables (and other database objects) owned by a user is called a *schema*.
    - Users can grant *privileges* so that other users can access the schema.
- A *session* starts when you connect to the system.
- Once you connect to the database system, all your changes are considered a single *transaction* until you either *commit* or *rollback* your work.
- *SQL* is a standard language for querying, manipulating data, and creating and managing objects in your schema.
- The *Data Dictionary* (also called a *System Catalog*) is a set of ordinary tables, maintained by the system, whose rows describe the tables in your schema.
  - You can query a system catalog table just like any other table.

You can use the Oracle Enterprise Manager to graphically display database schemas, users, and object details, as well as to perform a variety of administrative tasks. You may also use SQL*Plus to perform many of the same tasks. For example, you can use SQL*Plus to query the Data Dictionary:

dictionary.sql

```
SELECT *
  FROM dictionary
 WHERE table_name LIKE 'USER%';
```

user_tables.sql

```
SELECT table_name FROM user_tables;
```

INTRODUCTION TO SQL

- SQL is the abbreviation for *Structured Query Language*.
    - It is often pronounced as "sequel."
- SQL was first developed by IBM in the mid-1970s.
- SQL is the international standard language for relational database management systems.
    - SQL is considered a fourth-generation language.
    - It is English-like and intuitive.
    - SQL is robust enough to be used by:
        - Users with non-technical backgrounds.
        - Professional developers.
        - Database administrators.
- SQL is a non-procedural language that emphasizes what to get, but not how to get it.
- Each vendor has its own implementation of SQL; most large vendors comply with SQL-99 or SQL:2003 and have added extensions for greater functionality.

SQL statements can be placed in two main categories:

Data Manipulation Language (DML):

Query:	**SELECT**

Data Manipulation:	**INSERT**
	UPDATE
	DELETE

Transaction Control:	**COMMIT** **ROLLBACK**

Data Definition Language (DDL):

Data Definition:	**CREATE** **ALTER** **DROP**
Data Control:	**GRANT** **REVOKE**

SQL is actually an easy language to learn (many users pick up the basics with no additional instruction). SQL statements look more like natural language than many other programming languages. We can parse them into "verbs," "clauses," and "predicates." Additionally, SQL is a compact language, making it easy to learn and remember. Users and programmers spend most of their time working with only four simple keywords (the Query and DML verbs in the list above). Of course, as we'll learn in this class, you can use them in sophisticated ways.

ORACLE VERSIONING AND HISTORY

- The original "Oracle," named Software Development Laboratories, was founded in 1977, which then changed its name to Relational Software in 1979.
  - There was no Version 1 for marketing purposes.
  - Version 2 supported just basic SQL functionality.
- The *i* in Oracle versions stands for 'internet.'
  - The database has features that make it more accessible over the World Wide Web.
  - A Java Virtual Machine was embedded into the database.
- The *g* in Oracle versions stands for 'grid.'
  - Databases can be managed remotely by a web accessible tool, the Oracle Enterprise Manager (OEM).
    - The OEM comes in Database Control and Grid Control versions, depending on whether you are managing a single database, or a grid of systems.

Version	Release Year	New Features
Relational Software - Version 2	1979	Basic SQL functionality for queries and joins. No support for transactions. Operated on VAX/VMS systems only.
Oracle 3	1983	Rollback and commit transactions supported. UNIX operating system supported.
Oracle 4	1984	Read consistency.
Oracle 5	1985	Client-Server model. Supported networking, distributed queries.
Oracle 6	1988	Oracle Financials released. PL/SQL, hot backups, row-level locking.
Oracle 7	1992	Triggers, integrity constraints, stored procedures.
Oracle 8	1997	Object-oriented development, multi-media applications.
Oracle 8*i*	1999	Support for the internet. Contained native Java Virtual Machine.
Oracle 9*i*	2001	Over 400 new features — flashback query, multiple block sizes, etc. Ability to manipulate XML documents.
Oracle 10*g*	2003	Grid computing-ready features — clustering servers together to act as one large computer. Regular expressions, PHP support, data pumping (replaces import/export), and SQL Model clause, plus many more features.
Oracle 11*g*	2007	Improved grid administration.

The American National Standards Institute (ANSI) published the accepted standard for a database language, SQL, in 1986 (X3.135-1986). This standard was updated in 1989 (also called SQL89 or SQL1) and included referential integrity and column constraints (X3.135-1989). The 1992 standard (also called SQL2) offers a larger and more detailed definition of the SQL-89 standard. SQL-92 is almost 600 pages in length, while SQL-89 is about 115 pages. SQL-92 adds additional support capabilities, extended error handling facilities, better security features, and a more complete definition of the SQL language overall.

A new standard was published in 1999. Some critical features of SQL:1999 include a computationally-complete (that is, procedural) language and support for object-oriented data. Oracle 10*g* adheres to SQL:1999 standards.

Standards continue to evolve, with SQL:2003 as the successor to SQL:1999.

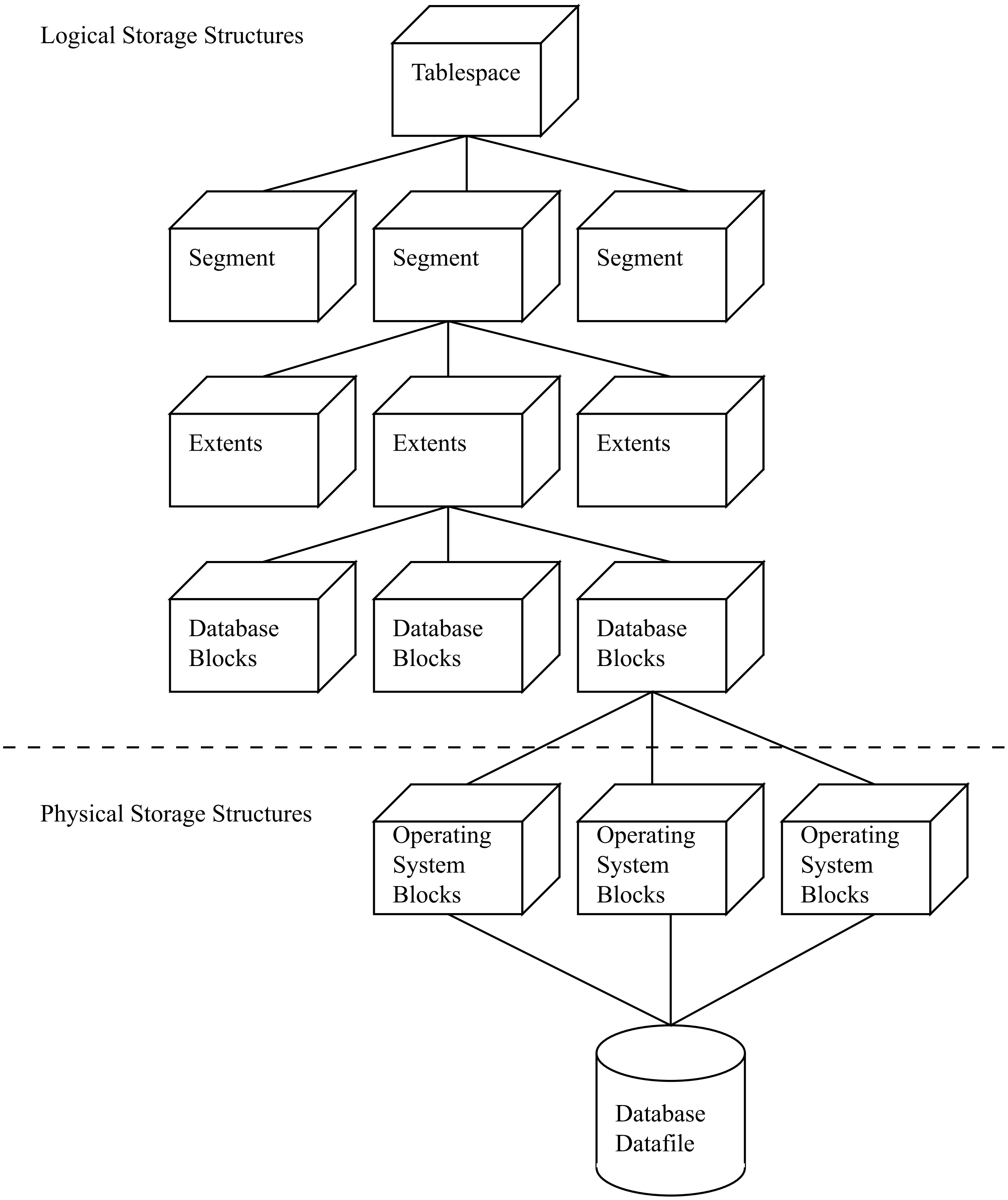
LOGICAL AND PHYSICAL STORAGE STRUCTURES

Logical Storage Structures:

- A *tablespace* stores the tables, views, indexes, and other schema objects.
  - It is the primary logical storage structure of an Oracle database.
- Each tablespace can contain one or more *segments*, which are made up of *extents*, which consist of *database blocks*.
- Each *database object* has its own segment storage.
  - When an object runs out of space, an extent is issued to the object.
- Each extent is made up of database blocks, which are the smallest logical storage unit.

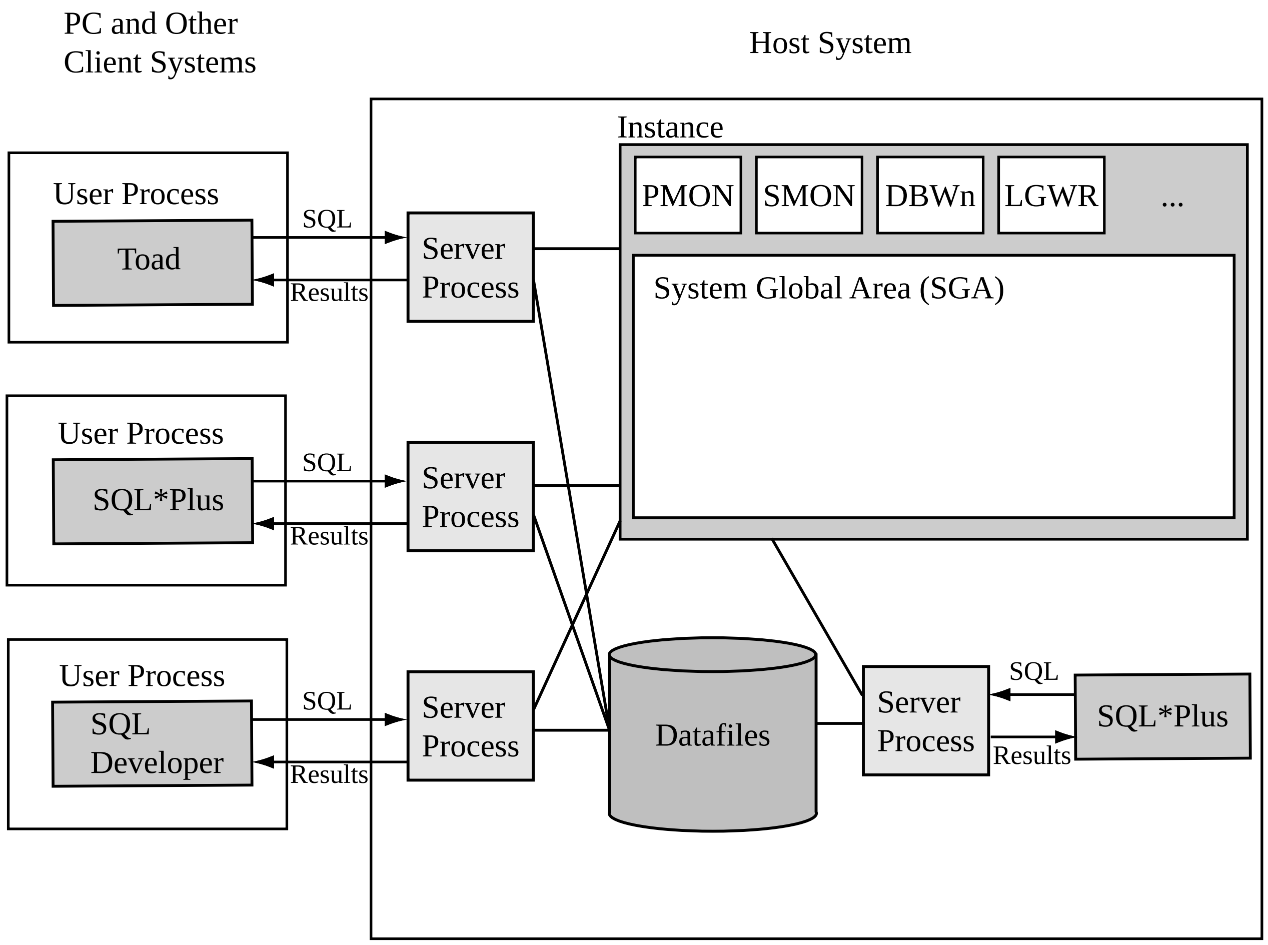
Physical Storage Structures:

- A tablespace is stored in one or more datafiles.
  - A *datafile* is a binary file whose name usually ends in *.dbf*.
- Datafiles consist of *operating system blocks*, which are not the same as database blocks.
  - A database block may consist of many operating system blocks.
- A *database* is the set of datafiles, as well as files containing configuration and management information, necessary to run an RDBMS.
  - An Oracle database includes a **SYSTEM** tablespace, as well as others.
  - The **SYSTEM** tablespace contains the Data Dictionary.


Logical Storage Structures
Tablespace
Segment
Segment
Segment
Extents
Extents
Extents
Database Blocks
Database Blocks
Database Blocks
Physical Storage Structures
Operating System Blocks
Operating System Blocks
Operating System Blocks
Database Datafile

Connecting to a SQL Database

- An *Oracle instance* is a set of processes and memory which coordinate efficient access to a database.
  - An instance must be running in order for you to access the database.
- A *server process* is a program that uses both the instance and the datafiles to give you access to the database.
  - All access to the data is performed by the server process.
  - Together, we sometimes refer to the instance and server process as the *database server* or *engine*.
- A *user process*, or client, is the program you run that uses database data.
  - The user process sends requests to the server in the form of SQL statements, and gets the resulting data in return.
  - The client program and the server program might be running on the same physical machine.
  - The client may be on a different machine, communicating with the server over a network.
- To begin using the database, your client program must connect to the server, thus starting a *session*.
  - To connect, you must provide a valid database account name (and usually a password).


PC and Other Client Systems
Host System
Instance
User Process
Toad
SQL
Results
Server Process
PMON
SMON
DBWn
LGWR
...
System Global Area (SGA)
User Process
SQL*Plus
SQL
Results
Server Process
User Process
SQL Developer
SQL
Results
Server Process
Datafiles
Server Process
SQL
Results
SQL*Plus

DATATYPES

- Each column in a table has a specific, predefined datatype.
    - Only values of the correct type can be stored in the column.
    - Many datatype definitions also include a limit on the size of the values that are allowed.
- The details of how data values are stored vary among database vendors.
    - Most database vendors provide similar sets of datatypes, though.
- The most important datatypes in Oracle include:
    - **VARCHAR2** — Text values whose length can vary, up to a predefined maximum length for each column.
    - **NUMBER** — Numeric values, possibly with predefined precision and scale.
    - **DATE** — A special value representing a moment in time, with one-second precision.
        - When you retrieve a **DATE** value from the database, Oracle normally converts it to a string representation of the date value, in a readable format.
    - **CHAR** — Text values of a specific predefined length; if a shorter value is inserted, Oracle automatically pads it to the correct length with spaces.

VARCHAR2

When you define a **VARCHAR2** column, you specify the maximum number of characters allowed for a value in the column. For example, for U.S. state names, you might define a column as **VARCHAR2(14)**, but for a product description you might define a column as **VARCHAR2(200)**. A **VARCHAR2** column can contain no more than 4000 bytes. **NVARCHAR2** supports Unicode.

CHAR

Use **CHAR** columns for values that are always the same length — state abbreviations, area codes, phone numbers, etc. It is inconvenient to store varying-length values in a **CHAR**, because you have to account for space-padding at the end of shorter values. **NCHAR** supports Unicode.

DATE vs. TIMESTAMP

The **DATE** datatype stores the century, year, month, day, hours, minutes, and seconds of a date. The **TIMESTAMP** datatype contains that same information, plus milliseconds. Calculating the interval between two dates is much easier if you use timestamps.

NUMBER (*precision*, *scale*)

Precision refers to the total number of digits, and scale is the number of digits to the right of the decimal point. If precision and scale are not specified, the max values are assumed. If a value exceeds the precision, Oracle returns an error. If the value exceeds the scale, it will be rounded:

Definition	Data	Stored Data
NUMBER	12345.678	12345.678
NUMBER(3)	1234	error
NUMBER(5,2)	123.45	123.45
NUMBER(5,2)	123.45678	123.46
NUMBER(7,-3)	123432.54	123000
NUMBER(5,2)	1234.56	error

Sample Database

Our company is a hardware/software retailer with stores in several cities.

We keep track of each person's name, address, and phone. In addition, if a person is an employee, we must record the store in which he or she works, the supervisor's ID, the employees's title, pay amount, and compensation type ("Hourly," "Salaried," etc.)

Sometimes a customer will fill out an order, which requires an invoice number. Each invoice lists the store and the customer's ID. We record the quantity of each item on the invoice and any discount for that item. We also keep track of how much the customer has paid on the order.

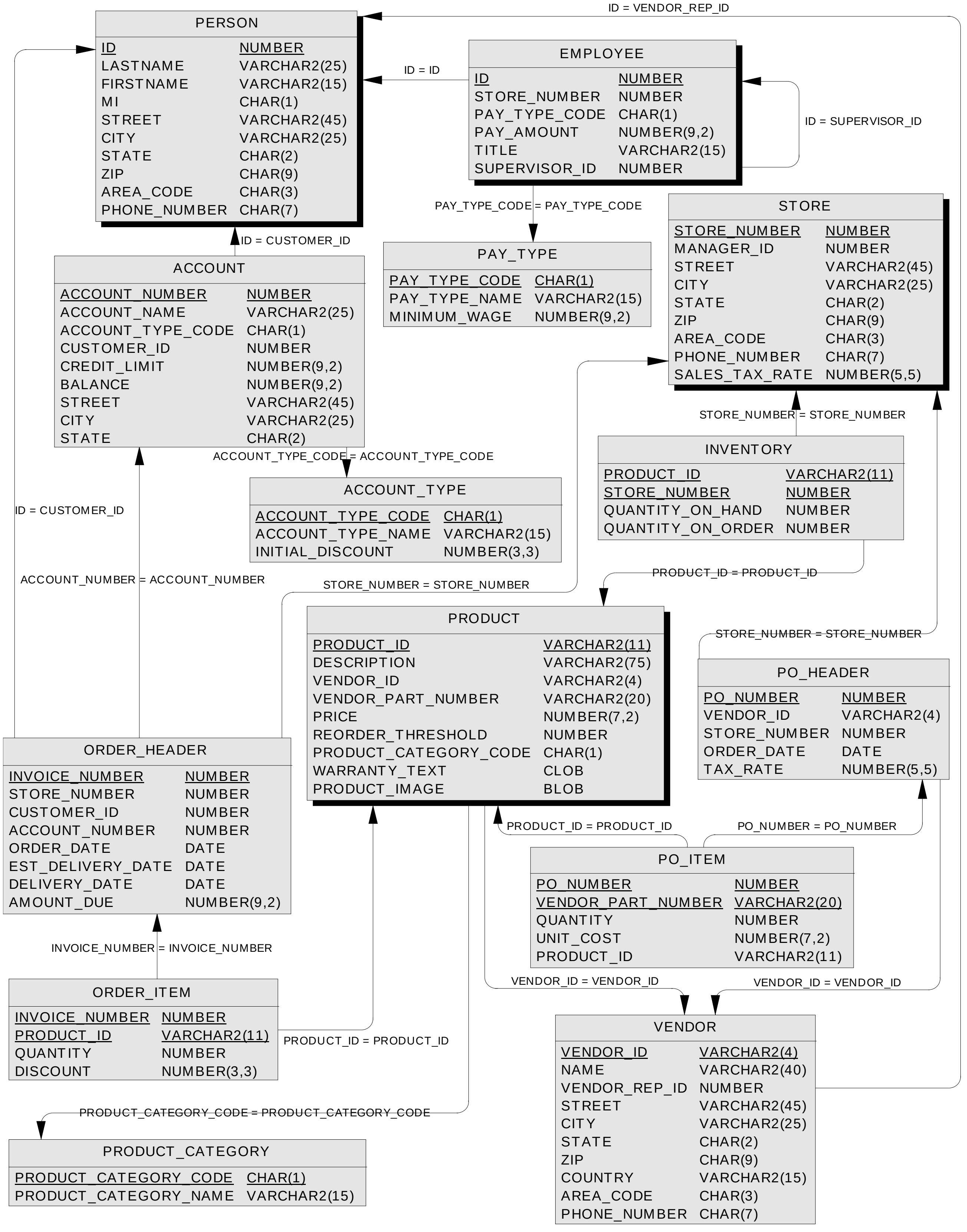
When a credit account is used for an order, the balance for the account must be updated to reflect the total amount of the invoice (including tax). The salesperson verifies (with a phone call) that the person is a valid user of the account.

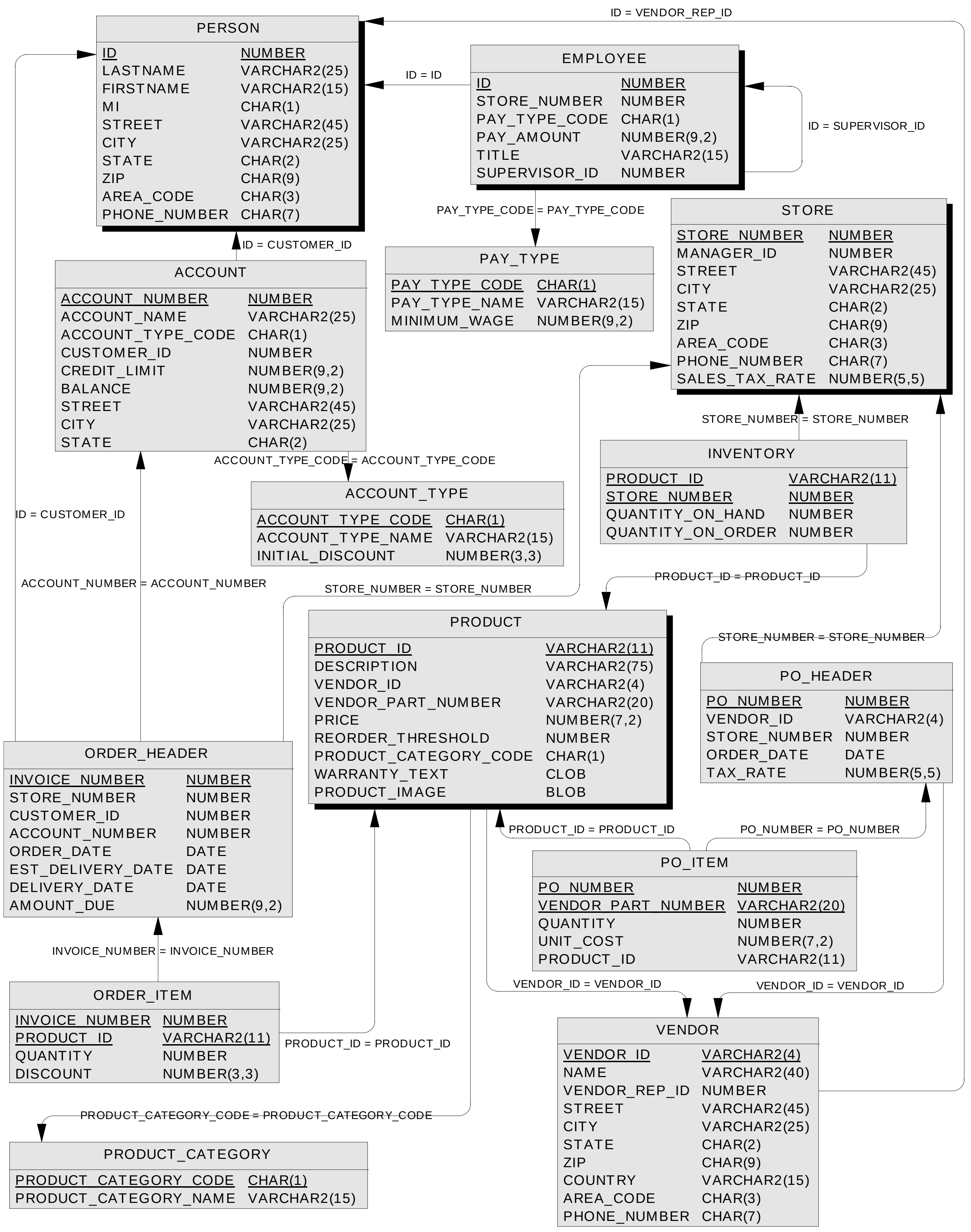
Each product we sell has a product ID and description. In addition, we keep track of the vendor from whom we purchase that product, the ID for that product, and the category ("Software," "Peripheral," or "Service"). We also store the address, phone, and sales rep ID for each individual vendor.

We keep track of how many items are on hand at each store and how many each store has on order. When an item is sold, the inventory is updated.

We maintain the address, phone number, and manager ID for each store. Each store has a unique store number. We record the sales tax rate for that store.

When a store runs low on a product, we create a purchase order. Each purchase order in our company has a unique PO number. A PO is sent to a single vendor, from which we might be ordering several items, each at a specific unit cost. The inventory reflects the sale or order of any item in a store.


PERSON
ID NUMBER
LASTNAME VARCHAR2(25)
FIRSTNAME VARCHAR2(15)
MI CHAR(1)
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
ID = VENDOR_REP_ID
ID = ID
EMPLOYEE
ID NUMBER
STORE_NUMBER NUMBER
PAY_TYPE_CODE CHAR(1)
PAY_AMOUNT NUMBER(9,2)
TITLE VARCHAR2(15)
SUPERVISOR_ID NUMBER
ID = SUPERVISOR_ID
PAY_TYPE_CODE = PAY_TYPE_CODE
ID = CUSTOMER_ID
ACCOUNT
ACCOUNT_NUMBER NUMBER
ACCOUNT_NAME VARCHAR2(25)
ACCOUNT_TYPE_CODE CHAR(1)
CUSTOMER_ID NUMBER
CREDIT_LIMIT NUMBER(9,2)
BALANCE NUMBER(9,2)
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
PAY_TYPE
PAY_TYPE_CODE CHAR(1)
PAY_TYPE_NAME VARCHAR2(15)
MINIMUM_WAGE NUMBER(9,2)
STORE
STORE_NUMBER NUMBER
MANAGER_ID NUMBER
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
SALES_TAX_RATE NUMBER(5,5)
STORE_NUMBER = STORE_NUMBER
ACCOUNT_TYPE_CODE = ACCOUNT_TYPE_CODE
ACCOUNT_TYPE
ACCOUNT_TYPE_CODE CHAR(1)
ACCOUNT_TYPE_NAME VARCHAR2(15)
INITIAL_DISCOUNT NUMBER(3,3)
INVENTORY
PRODUCT_ID VARCHAR2(11)
STORE_NUMBER NUMBER
QUANTITY_ON_HAND NUMBER
QUANTITY_ON_ORDER NUMBER
ID = CUSTOMER_ID
ACCOUNT_NUMBER = ACCOUNT_NUMBER
STORE_NUMBER = STORE_NUMBER
PRODUCT_ID = PRODUCT_ID
PRODUCT
PRODUCT_ID VARCHAR2(11)
DESCRIPTION VARCHAR2(75)
VENDOR_ID VARCHAR2(4)
VENDOR_PART_NUMBER VARCHAR2(20)
PRICE NUMBER(7,2)
REORDER_THRESHOLD NUMBER
PRODUCT_CATEGORY_CODE CHAR(1)
WARRANTY_TEXT CLOB
PRODUCT_IMAGE BLOB
STORE_NUMBER = STORE_NUMBER
PO_HEADER
PO_NUMBER NUMBER
VENDOR_ID VARCHAR2(4)
STORE_NUMBER NUMBER
ORDER_DATE DATE
TAX_RATE NUMBER(5,5)
ORDER_HEADER
INVOICE_NUMBER NUMBER
STORE_NUMBER NUMBER
CUSTOMER_ID NUMBER
ACCOUNT_NUMBER NUMBER
ORDER_DATE DATE
EST_DELIVERY_DATE DATE
DELIVERY_DATE DATE
AMOUNT_DUE NUMBER(9,2)
PRODUCT_ID = PRODUCT_ID
PO_NUMBER = PO_NUMBER
PO_ITEM
PO_NUMBER NUMBER
VENDOR_PART_NUMBER VARCHAR2(20)
QUANTITY NUMBER
UNIT_COST NUMBER(7,2)
PRODUCT_ID VARCHAR2(11)
INVOICE_NUMBER = INVOICE_NUMBER
ORDER_ITEM
INVOICE_NUMBER NUMBER
PRODUCT_ID VARCHAR2(11)
QUANTITY NUMBER
DISCOUNT NUMBER(3,3)
PRODUCT_ID = PRODUCT_ID
VENDOR_ID = VENDOR_ID
VENDOR_ID = VENDOR_ID
VENDOR
VENDOR_ID VARCHAR2(4)
NAME VARCHAR2(40)
VENDOR_REP_ID NUMBER
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
COUNTRY VARCHAR2(15)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
PRODUCT_CATEGORY_CODE = PRODUCT_CATEGORY_CODE
PRODUCT_CATEGORY
PRODUCT_CATEGORY_CODE CHAR(1)
PRODUCT_CATEGORY_NAME VARCHAR2(15)


PERSON
ID NUMBER
LASTNAME VARCHAR2(25)
FIRSTNAME VARCHAR2(15)
MI CHAR(1)
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
ID = VENDOR_REP_ID
EMPLOYEE
ID NUMBER
STORE_NUMBER NUMBER
PAY_TYPE_CODE CHAR(1)
PAY_AMOUNT NUMBER(9,2)
TITLE VARCHAR2(15)
SUPERVISOR_ID NUMBER
ID = ID
ID = SUPERVISOR_ID
PAY_TYPE_CODE = PAY_TYPE_CODE
STORE
STORE_NUMBER NUMBER
MANAGER_ID NUMBER
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
SALES_TAX_RATE NUMBER(5,5)
ID = CUSTOMER_ID
ACCOUNT
ACCOUNT_NUMBER NUMBER
ACCOUNT_NAME VARCHAR2(25)
ACCOUNT_TYPE_CODE CHAR(1)
CUSTOMER_ID NUMBER
CREDIT_LIMIT NUMBER(9,2)
BALANCE NUMBER(9,2)
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
PAY_TYPE
PAY_TYPE_CODE CHAR(1)
PAY_TYPE_NAME VARCHAR2(15)
MINIMUM_WAGE NUMBER(9,2)
STORE_NUMBER = STORE_NUMBER
INVENTORY
PRODUCT_ID VARCHAR2(11)
STORE_NUMBER NUMBER
QUANTITY_ON_HAND NUMBER
QUANTITY_ON_ORDER NUMBER
ACCOUNT_TYPE_CODE = ACCOUNT_TYPE_CODE
ACCOUNT_TYPE
ACCOUNT_TYPE_CODE CHAR(1)
ACCOUNT_TYPE_NAME VARCHAR2(15)
INITIAL_DISCOUNT NUMBER(3,3)
ID = CUSTOMER_ID
ACCOUNT_NUMBER = ACCOUNT_NUMBER
STORE_NUMBER = STORE_NUMBER
PRODUCT_ID = PRODUCT_ID
PRODUCT
PRODUCT_ID VARCHAR2(11)
DESCRIPTION VARCHAR2(75)
VENDOR_ID VARCHAR2(4)
VENDOR_PART_NUMBER VARCHAR2(20)
PRICE NUMBER(7,2)
REORDER_THRESHOLD NUMBER
PRODUCT_CATEGORY_CODE CHAR(1)
WARRANTY_TEXT CLOB
PRODUCT_IMAGE BLOB
STORE_NUMBER = STORE_NUMBER
PO_HEADER
PO_NUMBER NUMBER
VENDOR_ID VARCHAR2(4)
STORE_NUMBER NUMBER
ORDER_DATE DATE
TAX_RATE NUMBER(5,5)
ORDER_HEADER
INVOICE_NUMBER NUMBER
STORE_NUMBER NUMBER
CUSTOMER_ID NUMBER
ACCOUNT_NUMBER NUMBER
ORDER_DATE DATE
EST_DELIVERY_DATE DATE
DELIVERY_DATE DATE
AMOUNT_DUE NUMBER(9,2)
PRODUCT_ID = PRODUCT_ID
PO_NUMBER = PO_NUMBER
PO_ITEM
PO_NUMBER NUMBER
VENDOR_PART_NUMBER VARCHAR2(20)
QUANTITY NUMBER
UNIT_COST NUMBER(7,2)
PRODUCT_ID VARCHAR2(11)
INVOICE_NUMBER = INVOICE_NUMBER
ORDER_ITEM
INVOICE_NUMBER NUMBER
PRODUCT_ID VARCHAR2(11)
QUANTITY NUMBER
DISCOUNT NUMBER(3,3)
PRODUCT_ID = PRODUCT_ID
VENDOR_ID = VENDOR_ID
VENDOR_ID = VENDOR_ID
VENDOR
VENDOR_ID VARCHAR2(4)
NAME VARCHAR2(40)
VENDOR_REP_ID NUMBER
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
COUNTRY VARCHAR2(15)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
PRODUCT_CATEGORY_CODE = PRODUCT_CATEGORY_CODE
PRODUCT_CATEGORY
PRODUCT_CATEGORY_CODE CHAR(1)
PRODUCT_CATEGORY_NAME VARCHAR2(15)

CHAPTER 4 - SQL QUERIES – THE SELECT STATEMENT

OBJECTIVES

❈ Retrieve data from a table using the SQL **SELECT** statement.

❈ Filter the returned rows with a **WHERE** clause.

❈ Locate fields which have not yet been populated.

❈ Combine multiple search criteria in a **WHERE** clause using logical operators.

❈ Use the **IN** and **BETWEEN** operators to match a column against multiple values.

❈ Use wildcards to search for a pattern in character data.

❈ Sort the output of a **SELECT** statement.

THE SELECT STATEMENT

❈ SQL data retrievals are done using the **SELECT** statement.

➢ Data retrievals are also called *queries*.

❈ A **SELECT** statement may make use of several *clauses*, but minimally must include the following two:

```
SELECT [DISTINCT] {item_list | *}
  FROM table;
```

❈ The **SELECT** clause specifies which data values will be retrieved, and will be followed by either an ***item_list*** or a *****.

➢ ***** represents all the columns in ***table***.

➢ ***item_list*** is a column, an expression, or a comma-separated list of any combination of these.

➢ Oracle will return each item in ***item_list*** under a column heading of the same name, unless an alias is specified.

➢ Use **DISTINCT** to eliminate duplicate rows from the displayed results.

❈ The **FROM** clause specifies one or more sources, typically tables, from which to retrieve data.

❈ A heading may be overridden with a *column alias*, a word used to replace the default heading.

product.sql

```
SELECT description name,
       price orig_price,
       price * .75 sales_price
  FROM product;
```

The **SELECT** statement may query one or many tables. The data that is retrieved may be thought of as a *virtual table* (i.e., one that exists in memory only until the query completes). This virtual table is also referred to as the query's *result set*.

The **SELECT** statement can be used to retrieve data from any or all columns from a table. For example, the following **SELECT** will display a list of everything about every store in the database:

store.sql

```
SELECT *
  FROM store;
```

The query below will list each state (once) in which our company has stores:

store2.sql

```
SELECT DISTINCT state
  FROM store;
```

This query will retrieve each store's number and location.

store3.sql

```
SELECT store_number, city, state
  FROM store;
```

Note:

Oracle allows column aliases to contain spaces. You must use double quotes if the alias includes one or more spaces.

product2.sql

```
SELECT description AS name,
       price AS orig_price,
       price * .75 AS "Sales Price"
  FROM product;
```

THE CASE...WHEN EXPRESSION

- The **CASE** expression is used to evaluate a column or expression against various values.

```
CASE [expression]
     WHEN value THEN result
     WHEN value THEN result
     ELSE result
 END;
```

  - A **CASE** expression results in a single value, either from one of the **WHEN** clauses, the **ELSE** clause, or otherwise a **NULL**.

- The *simple* **CASE** expression evaluates one expression and compares it for equality with the value of each of the **WHEN** clauses.

simple_case.sql

```
SELECT id, pay_amount, CASE pay_type_code
                              WHEN 'S' THEN 'Salaried'
                              WHEN 'C' THEN 'Commissioned'
                              WHEN 'T' THEN 'Temporary'
                              WHEN 'H' THEN 'Hourly'
                              ELSE 'Unknown Pay Type'
                          END
  FROM employee;
```

- A *searched* **CASE** evaluates the condition in each of the **WHEN** clauses, taking the first one that's true.

searched_case.sql

```
SELECT id, pay_amount,
  CASE WHEN pay_type_code='S' THEN 'Salaried'
       WHEN pay_type_code='C' THEN 'Commissioned'
       WHEN pay_type_code='T'
            AND pay_amount < 8 THEN 'Poorly Paid Temporary'
       WHEN pay_type_code='T' THEN 'Temporary'
       WHEN pay_type_code='H' THEN 'Hourly'
       ELSE 'Unknown Pay Type'
   END
  FROM employee;
```

A simple **CASE** expression can be rewritten as a searched **CASE**, but a searched **CASE** can't always be rewritten as a simple **CASE**:

searched_case2.sql
```
SELECT account_number, account_name, balance,
  CASE WHEN balance > credit_limit - 100
       THEN 'Reaching Limit'
   END warning,  credit_limit
  FROM account;
```

Other Functions

Before implementing the ANSI-standard **CASE** construct, Oracle supplied other functions. You will still find these used in current and legacy applications, though developers will probably choose the **CASE** expression for new code.

NVL() will replace a null value with another value, or leave a non-null value alone.

nvl.sql
```
SELECT id, NVL(store_number, 999) store FROM employee;
```

COALESCE() returns the first non-null value in a list of values.

coalesce.sql
```
SELECT COALESCE(delivery_date, est_delivery_date) delivery_date
  FROM order_header;
```

DECODE() is more general than **NVL()** in that it will search through a list of values to determine what value to return. Values not found will return **NULL**.

decode.sql
```
SELECT DECODE(state, NULL, '?', 'UT', 'Utah') state FROM vendor;
```

NULLIF() will return a null if both values are the same; otherwise, it will return the first value.

nullif.sql
```
SELECT product_id, NULLIF(quantity_on_hand, 0) quantity FROM inventory;
```

CHOOSING ROWS WITH THE WHERE CLAUSE

- Use the **WHERE** clause when only a subset of all the rows in a table is required.

  ```
  SELECT {item_list | *}
    FROM table
  [WHERE conditions];
  ```

- The **WHERE** clause is evaluated once for each row in ***table.***

- ***conditions*** will be one or more expressions that will evaluate to either true, false, or neither (null).

  - If the **WHERE** clause evaluates to true, then the current row is returned in the result set.

- Operators that may be used in the **WHERE** clause include:

=	equal to
!=, ^=, <>	not equal to
>**,** <	greater than, less than
>=**,** <=	greater than or equal to, less than or equal to

- Any character string values in conditions must be enclosed within single quotes.

  - Database data inside single quotes is case sensitive.

To display all information for each store with a store number less than 4, enter the following:

store4.sql

```
SELECT *
  FROM store
 WHERE store_number < 4;
```

To display each store's number, location, and phone number in the state of Nevada, enter the following:

store5.sql

```
SELECT store_number, street, city, area_code, phone_number
  FROM store
 WHERE state = 'NV';
```

NULL VALUES

- A *null* is the absence of a value.
  - Null values are not equal to **0**, blanks, or empty strings.
- Compare null values with the **IS** operator.

  person2.sql
  ```
  SELECT *
    FROM person
   WHERE mi IS NULL;
  ```
  - The **IS** operator will result in either true or false.
- To search for any non-null values, use **IS NOT NULL**.

  person3.sql
  ```
  SELECT *
    FROM person
   WHERE phone_number IS NOT NULL;
  ```
- **NULL** is not the same as a "false" value, but it isn't a "true" value, either.
  - **NULL** means "unknown;" any operation with a **NULL** (other than **IS** or **IS NOT**) yields a **NULL**.

    person4.sql
    ```
    SELECT *
      FROM person
     WHERE phone_number = NULL;
    ```
  - This will never be true, even for rows with null values in the **phone_number** column!

To display a list of people who have middle initials, enter the following:

person5.sql

```
SELECT lastname, firstname, mi, area_code, phone_number
  FROM person
 WHERE mi IS NOT NULL;
```

To display a list of employees who are not managed by employee 7881:

employee.sql

```
SELECT id
  FROM employee
 WHERE supervisor_id != 7881 OR
       supervisor_id IS NULL;
```

Compound Expressions

❋ Use logical operators to group conditions in a **WHERE** clause.

- **NOT** will negate the condition following it.
- **AND** will result in true if both conditions are true for a row.
- **OR** will result in true if either condition is true for a row.

❋ The logical operators are evaluated based on precedence: **NOT** has the highest precedence, then **AND**, and then **OR**.

- If a **WHERE** clause contains both an **AND** and an **OR**, the **AND** will always be evaluated first.

  and.sql
  ```
  SELECT *
    FROM  person
   WHERE lastname = 'Johnson'
      OR lastname = 'Johanson'
     AND firstname = 'Carole';
  ```

  - This query will find **Carole Johanson,** as well as anyone with the last name of **Johnson**.

❋ Use parentheses to override the precedence of these operators.

and2.sql
```
SELECT *
  FROM person
 WHERE (lastname = 'Johnson'
    OR lastname = 'Johanson')
   AND firstname = 'Carole';
```

- This query will find all people with a **firstname** of **Carole** and a **lastname** of either **Johnson** or **Johanson**.

The query below will find all accounts having a minimum $1000 credit limit that have a non-zero balance:

account.sql

```
SELECT account_number "ACCT NUMBER",
       account_name name,
       credit_limit limit,
       balance
  FROM account
 WHERE credit_limit >= 1000 AND
       balance > 0;
```

When an employee enters data into the system, a field for which there is no data may be skipped or have a **0** entered. When checking on the status of a store's inventory, the query below will display any products at each store that may be running low in inventory and haven't yet been reordered:

inventory.sql

```
SELECT product_id, store_number
  FROM inventory
 WHERE quantity_on_hand < 20 AND
       (quantity_on_order = 0 OR
       quantity_on_order IS NULL);
```

IN AND BETWEEN

- Use the **IN** (or **NOT IN**) operator to compare a column against several possible values.

  in.sql
  ```
  SELECT lastname, firstname
    FROM person
   WHERE state IN ('CA', 'CO');
  ```

  - **IN** is equivalent to **OR**ing several conditions together.

    or.sql
    ```
    SELECT lastname, firstname
      FROM person
     WHERE state = 'CA'
        OR state = 'CO';
    ```

- Use the **BETWEEN** operator to compare a column against a range of inclusive values.

  between.sql
  ```
  SELECT lastname, firstname
    FROM person
   WHERE id BETWEEN 6900 AND 7000;
  ```

  - The **AND** is part of the **BETWEEN** operator.

  - Make sure that lower values appear before higher ones:

    between_date.sql
    ```
    SELECT invoice_number, order_date
      FROM order_header
     WHERE order_date BETWEEN '01-NOV-95' AND '01-DEC-95';
    ```

To display the same information for both Washington and Colorado, enter the following:

store6.sql
```
SELECT store_number, street, city, area_code, phone_number
  FROM store
 WHERE state = 'WA' OR state = 'CO';
```

An equivalent query makes use of the **IN** operator:

store7.sql
```
SELECT store_number, street, city, area_code, phone_number
  FROM store
 WHERE state IN ('WA', 'CO');
```

To list all store locations in California outside the 415 and 213 area codes:

store8.sql
```
SELECT store_number, street, city, area_code, phone_number
  FROM store
 WHERE area_code NOT IN ('213', '415')
   AND state = 'CA';
```

To display a specific vendor's products that fall in the $100 to $500 price range:

product7.sql
```
SELECT description, price
  FROM product
 WHERE vendor_id = 'IBM' AND price BETWEEN 100 AND 500;
```

To display all orders placed between July 20, 1996 and August 1, 1996:

order.sql
```
SELECT invoice_number, customer_id, order_date
  FROM order_header
 WHERE order_date BETWEEN '20-JUL-1996' AND '01-AUG-1996';
```

PATTERN MATCHING: LIKE AND REGEXP_LIKE

- The **LIKE** operator provides pattern matching for character data.
- With the **LIKE** operator, you can use *wildcards*:
  - **%** Matches zero or more characters.
  - **_** Matches exactly one character and is position-dependent.

    vendor_like.sql
    ```
    SELECT vendor_id, name
      FROM vendor
     WHERE name LIKE '%Software%';
    ```
- The string containing wildcards must be enclosed in quotes.
- Beginning with Oracle 10*g*, you may also use regular expressions to find string patterns.
- The **REGEXP_LIKE** operator works like a function that takes two arguments, the string to search in, and the pattern to look for.

  reg_person.sql
  ```
  SELECT firstname
    FROM person
   WHERE REGEXP_LIKE(firstname, '^Ste(v|ph)en$');
  ```
  - Use full regular expression syntax, similar to Perl, to find very complex patterns.
  - **REGEXP_LIKE** returns a boolean value that can be tested in a **WHERE** clause.
  - See Appendix C in the ***SQL Language Reference*** for Oracle's regular expression syntax.

person_like.sql

```
SELECT id, firstname, lastname
  FROM person
 WHERE lastname LIKE 'M%ll_';
```

This query will match names such as:

```
ID      FIRSTNAME     LASTNAME
------- ------------- ---------------
   6873 L. E.         McAnally
   8993 Madonna       Mullally
   9166 Angie         Matilla
   9412 Bruce         McNally
   9845 Neil          Montville
```

Or, to find names that end in **son**:

person_like2.sql

```
SELECT id, lastname, firstname
  FROM person
 WHERE lastname LIKE '%son';
```

If you need to include the **%** or **_** characters in the search pattern, specify an escape character. An *escape character* removes the special meaning of the character following it. For example, to locate vendor company names that contain an underscore, modify the query with an escape parameter.

vendor_like2.sql

```
SELECT name
  FROM vendor
 WHERE name LIKE '%\_%' ESCAPE '\';
```

To find names that contain a single quote, escape the single quote with another single quote:

person_like3.sql

```
SELECT  id,lastname,firstname
  FROM person
 WHERE lastname like '%''%';
```

CREATING SOME ORDER

- Data is not physically stored in a table in any specified order.
  - Unless the table is an Index-Organized Table (IOT), records are usually appended to the end of the table.
- The **SELECT** statement's **ORDER BY** clause is the only way to enforce sequencing on the result set.
- The **ORDER BY** clause may be included in the **SELECT** statement:

```
   SELECT {item_list | *}
     FROM table
[ORDER BY column [ASC | DESC],...];
```

  - ***column*** can either be the name or the numeric position of the column in the ***item_list***.
- The result set is sorted by the first column name in the **ORDER BY** clause.
  - If there are duplicate values in the first sort column, the sort is repeated on the second column in the list, and so on.
  - The rows may be sorted in ascending (the default) or descending order.
  - **ASC** and **DESC** may be applied independently to each column.
  - **NULL** values will be sorted after all non-**NULL** values in **ASC** order (and before all non-**NULL**s when you use **DESC**).

To display each store's number and location in descending order by state, and ascending order by city in each state, enter the following:

store9.sql
```
  SELECT store_number, city, state
    FROM store
ORDER BY state DESC, city ASC;
```

The description, and original and sale price of an item may be displayed in descending price order with the following:

product5.sql
```
  SELECT description, price,
         price * .75
    FROM product
ORDER BY 3 DESC;
```

An equivalent, and usually more readable, form of the above makes use of column aliases:

product6.sql
```
  SELECT description Name, price Original_Price,
         price * .75 sales_price
    FROM product
ORDER BY sales_price DESC;
```

Labs

Write queries to:

❶ Retrieve a list of all store locations and their manager's ids.
(Solution: *store.sql*)

❷ List the states in which there are stores, listing each state only once.
(Solution: *store_states.sql*)

❸ List the states in which there are people, listing each state only once.
(Solution: *person_states.sql*)

❹ List the people who live in Glendale, California.
(Solution: *glendale.sql*)

❺ List the last name, first name, city, and state of everyone who lives in a city that starts with "Las" or "Los" or "San".
(Solution: *las_los_san.sql*)

❻ Show which stores, if any, have no store manager.
(Solution: *no_manager.sql*)

❼ Write a query that displays the account name and type. Use **CASE...WHEN** to show the account type.
(Solution: *account.sql*)