

INTRODUCTION TO ORACLE 10G PROGRAMMING

Student Workbook

INTRODUCTION TO ORACLE 10G PROGRAMMING

Contributing Authors: Danielle Hopkins, Julie Johnson, John McAlister, Rob Roselius, and Rob Seitz

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, Roger Jones, Joe McGlynn, Jim McNally, and Kevin Smith, and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 2011 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photo-copying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, Colorado, 80112. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

Chapter 1 - Course Introduction	11
Course Objectives	12
Course Overview	14
Using the Workbook	15
Suggested References	16
Chapter 2 - Relational Database and SQL Overview	19
Review of Relational Database Terminology	20
Relational Database Management Systems	22
Introduction to SQL	24
Oracle Versioning and History	26
Logical and Physical Storage Structures	28
Connecting to a SQL Database	30
Datatypes	32
Sample Database	34
Chapter 3 - Using Oracle SQL*Plus	39
SQL*Plus	40
The SQL Buffer	42
Buffer Manipulation Commands	44
Running SQL*Plus Scripts	46
Tailoring Your SQL*Plus Environment	48
Viewing Table Characteristics	50
SQL*Plus Substitution Variables	52
Interactive SQL*Plus Scripts	54
SQL*Plus LOB Support	56
Using iSQL*Plus	58
Graphical Clients	60
Labs	62
Chapter 4 - SQL Queries — The SELECT Statement	65
The SELECT Statement	66
The CASE...WHEN Expression	68

Choosing Rows with the WHERE Clause	70
NULL Values	72
Compound Expressions	74
IN and BETWEEN	76
Pattern Matching: LIKE and REGEXP_LIKE	78
Creating Some Order	80
Labs	82
Chapter 5 - Scalar Functions	85
SQL Functions	86
Using SQL Functions	88
String Functions	90
Numeric Functions	92
Date Functions	94
Date Formats	96
Conversion Functions	98
Literal Values	100
Intervals	102
Oracle Pseudocolumns	104
Labs	106
Chapter 6 - SQL Queries — Joins	109
Selecting from Multiple Tables	110
Joining Tables	112
Self Joins	114
Outer Joins	116
Labs	118
Chapter 7 - Aggregate Functions and Advanced Techniques	121
Subqueries	122
Correlated Subqueries	124
The EXISTS Operator	126
The Aggregate Functions	128
Nulls and DISTINCT	130
Grouping Rows	132
Combining SELECT Statements	134
Labs	136

Chapter 8 - Data Manipulation and Transactions	139
The INSERT Statement	140
The UPDATE Statement	142
The DELETE Statement	144
Transaction Management	146
Concurrency	148
Explicit Locking	150
Data Inconsistencies	152
Loading Tables From External Sources	154
Labs	156
Chapter 9 - Data Definition and Control Statements	159
Datatypes	160
Defining Tables	162
Constraints	164
Inline Constraints	166
Modifying Table Definitions	168
Deleting a Table Definition	170
Controlling Access to Your Tables	172
Labs	174
Chapter 10 - Other Database Objects	177
Views	178
Creating Views	180
Updatable Views	182
Sequences	184
Synonyms	186
Labs	188
Chapter 11 - Triggers	191
Beyond Declarative Integrity	192
Triggers	194
Types of Triggers	196
Row-Level Triggers	198
Trigger Predicates	200
Trigger Conditions	202
Using SEQUENCES	204
Cascading Triggers and Mutating Tables	206

Generating an Error	208
Maintaining Triggers	210
Labs	212
Chapter 12 - PL/SQL Variables and Datatypes	215
Anonymous Blocks	216
Declaring Variables	218
Datatypes	220
Subtypes	222
Character Data	224
Dates and Timestamps	226
Date Intervals	228
Anchored Types	230
Assignment and Conversions	232
Selecting into a Variable	234
Returning into a Variable	236
Labs	238
Chapter 13 - PL/SQL Syntax and Logic	241
Conditional Statements — IF/THEN	242
Conditional Statements — CASE	244
Comments and Labels	246
Loops	248
WHILE and FOR Loops	250
SQL in PL/SQL	252
Local Procedures and Functions	254
Labs	256
Chapter 14 - Stored Procedures and Functions	259
Stored Subprograms	260
Creating a Stored Procedure	262
Procedure Calls and Parameters	264
Parameter Modes	266
Named Parameter Notation	268
Default Arguments	270
Creating a Stored Function	272
Stored Functions and SQL	274
Invoker's Rights	276
Labs	278

Chapter 15 - Exception Handling	281
SQLCODE and SQLERRM	282
Exception Handlers	284
Nesting Blocks	286
Scope and Name Resolution	288
Declaring and Raising Named Exceptions	290
User-Defined Exceptions	292
Labs	294
Chapter 16 - Records, Collections, and User-Defined Types	297
Record Variables	298
Using the %ROWTYPE Attribute	300
User-Defined Object Types	302
VARRAY and Nested TABLE Collections	304
Using Nested TABLEs	306
Using VARRAYs	308
Collections in Database Tables	310
Associative Array Collections	312
Collection Methods	314
Iterating Through Collections	316
Labs	318
Chapter 17 - Cursors	321
Multi-Row Queries	322
Declaring and Opening Cursors	324
Fetching Rows	326
Closing Cursors	328
The Cursor FOR Loop	330
FOR UPDATE Cursors	332
Cursor Parameters	334
The Implicit (SQL) Cursor	336
Labs	338
Chapter 18 - Bulk Operations	341
Bulk Binding	342
BULK COLLECT Clause	344
FORALL Statement	346
FORALL Variations	348
Bulk Returns	350

Bulk Fetching with Cursors	352
Labs	354
Chapter 19 - Using Packages	357
Packages	358
Oracle-Supplied Packages	360
The DBMS_OUTPUT Package	362
The DBMS_UTILITY Package	364
The UTL_FILE Package	366
Creating Pipes with DBMS_PIPE	368
Writing to and Reading from a Pipe	370
The DBMS_METADATA Package	372
XML Packages	374
Networking Packages	376
Other Supplied Packages	378
Labs	380
Chapter 20 - Creating Packages	383
Structure of a Package	384
The Package Interface and Implementation	386
Package Variables and Package State	388
Overloading Package Functions and Procedures	390
Forward Declarations	392
Strong REF CURSOR Variables	394
Weak REF CURSOR Variables	396
Labs	398
Chapter 21 - Working with LOBs	401
Large Object Types	402
Oracle Directories	404
LOB Locators	406
Internal LOBs	408
External LOBs	410
Temporary LOBs	412
The DBMS_LOB Package	414
Labs	416

Chapter 22 - Maintaining PL/SQL Code	419
Privileges for Stored Programs	420
Data Dictionary	422
PL/SQL Stored Program Compilation	424
Conditional Compilation	426
Compile-Time Warnings	428
The PL/SQL Execution Environment	430
Dependencies and Validation	432
Maintaining Stored Programs	434
Labs	436
Appendix A - The Data Dictionary	439
Introducing the Data Dictionary	440
DBA, ALL, and USER Data Dictionary Views	442
Some Useful Data Dictionary Queries	444
Appendix B - Dynamic SQL	447
Generating SQL at Runtime	448
Native Dynamic SQL vs. DBMS_SQL Package	450
The EXECUTE IMMEDIATE Statement	452
Using Bind Variables	454
Multi-row Dynamic Queries	456
Bulk Operations with Dynamic SQL	458
Using DBMS_SQL	460
DBMS_SQL Subprograms	462
Appendix C - PL/SQL Versions, Datatypes and Language Limits	465
Appendix D - Oracle 10g Supplied Packages	473
Solutions	485
Index	533

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.
- * Create triggers on database tables.
- * Use PL/SQL's datatypes for database and program data.
- * Use program structure and control flow to design and write PL/SQL programs.
- * Create PL/SQL stored procedures and functions.

- * Write robust programs that handle runtime exceptions.
- * Use PL/SQL's collection datatypes.
- * Use cursors to work with database data.
- * Use the packages supplied with Oracle.
- * Design and write your own packages.
- * Maintain and evolve your PL/SQL programs.
- * Manage the security of your stored PL/SQL programs.

COURSE OVERVIEW

- * **Audience:** Application developers, database administrators, system administrators, and users who write applications and procedures that access Oracle 10g.

- * **Prerequisites:** A solid understanding of 3GL programming is required.

- * **Student Materials:**
 - Student workbook

- * **Classroom Environment:**
 - One workstation per student

 - Oracle Server, SQL*Plus, and PL/SQL.

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
 - > The container uses a separate thread to call
 - > The 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

Topics are organized into first (*), second (>) and third (▪) level points.

CHAPTER 2 SERVLET BASICS

Hands On:

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

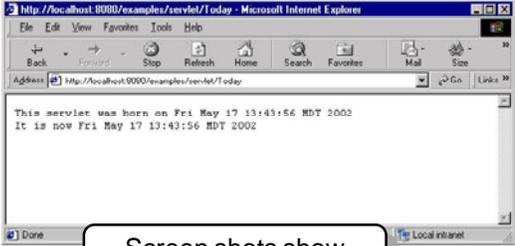
Today.java

```
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
        ServletResponse response) throws ServletException, IOException
    {
        ...
        Servlet was born on " + bornOn.toString();
        * + today.toString();
    }
}
```

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

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

Callout boxes point out important parts of the example code.



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

© 2002 ITCourseware, LLC Page 17

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

SUGGESTED REFERENCES

- Allen, Christopher. 2004. *Oracle Database 10g PL/SQL 101*. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072255404
- Boardman, Susan, Melanie Caffrey, Solomon Morse, and Benjamin Rosenzweig. 2002. *Oracle Web Application Programming for PL/SQL Developers*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0130477311
- Celko, Joe. 2005. *Joe Celko's SQL for Smarties: Advanced SQL Programming*. Academic Press/Morgan Kaufman, San Francisco, CA. ISBN 0123693799
- 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. 2003. *An Introduction to Database Systems*. Addison-Wesley, Boston, MA. ISBN 0321197844
- Feuerstein, Steven, Charles Dye, and John Beresniewicz. 1998. *Oracle Built-in Packages*. O'Reilly and Associates, Sebastopol, CA. ISBN 1565923758
- Feuerstein, Steven. 2007. *Oracle PL/SQL Best Practices, Second Edition*. O'Reilly and Associates, Sebastopol, CA. ISBN 0596514107
- Feuerstein, Steven. 2000. *Oracle PL/SQL Developer's Workbook*. O'Reilly and Associates, Sebastopol, CA. ISBN 1565926749
- Feuerstein, Steven and Bill Pribyl. 2005. *Oracle PL/SQL Programming, Fourth Edition*. O'Reilly and Associates, Sebastopol, CA. ISBN 0596009771
- Freeman, Robert G. 2004. *Oracle Database 10g New Features*. McGraw-Hill Osborne Media, Emeryville, CA. ISBN 0072229470
- Gennick, Jonathan. 2004. *Oracle Sql*Plus Pocket Reference, Third Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596008856

- Gennick, Jonathan. 2004. *Oracle SQL*Plus : The Definitive Guide, Second Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596007469
- Gruber, Martin. 2000. *SQL Instant Reference, Second Edition*. SYBEX, Alameda, CA. ISBN 0782125395
- Kline, Kevin. 2004. *SQL in a Nutshell, Second Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596004818
- Kreines, David. 2003. *Oracle Data Dictionary Pocket Reference*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005172
- Loney, Kevin. 2004. *Oracle Database 10g: The Complete Reference*. McGraw-Hill Osborne Media, ISBN 0072253517
- McDonald, Connor, Chaim Katz, Christopher Beck, Joel R. Kallman, and David C. Knox. 2004. *Mastering Oracle PL/SQL: Practical Solutions*. Apress, Berkeley, CA. ISBN 1590592174
- Mishra, Sanjay. 2004. *Mastering Oracle SQL, Second Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596006322
- Pribyl, Bill. 2001. *Learning Oracle PL/SQL*. O'Reilly and Associates, Sebastopol, CA. ISBN 0596001800
- Price, Jason. 2004. *Oracle Database 10g SQL*. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072229810
- Rosenzweig, Benjamin and Elena Silvestrova Rakhimov. 2008. *Oracle PL/SQL by Example*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0137144229

www.dbasupport.com
www.hot-oracle.com
www.oracle.com
www.toadworld.com
www.searchdatabase.com
tahiti.oracle.com

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 8i	1999	Support for the internet. Contained native Java Virtual Machine.
Oracle 9i	2001	Over 400 new features — flashback query, multiple block sizes, etc. Ability to manipulate XML documents.
Oracle 10g	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. Automation and ease of 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 10g adheres to SQL:1999 standards.

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

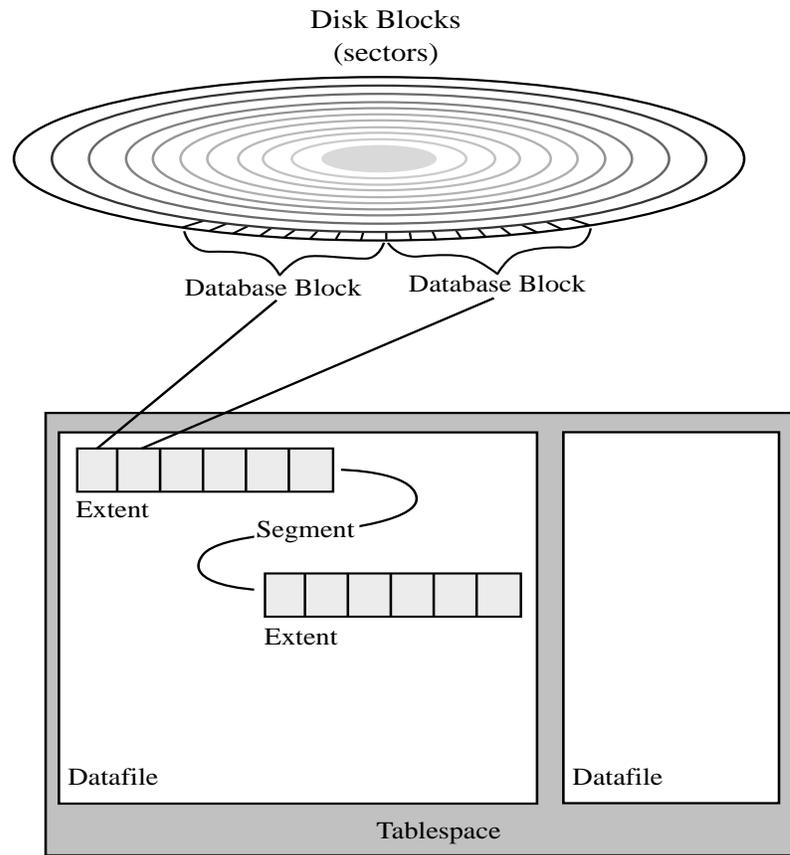
LOGICAL AND PHYSICAL STORAGE STRUCTURES

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*, such as a table or index, has its own segment storage.
 - When an object runs out of space, Oracle allocates another extent to the object.
- * Each extent is made up of adjacent database blocks, the smallest logical storage unit.

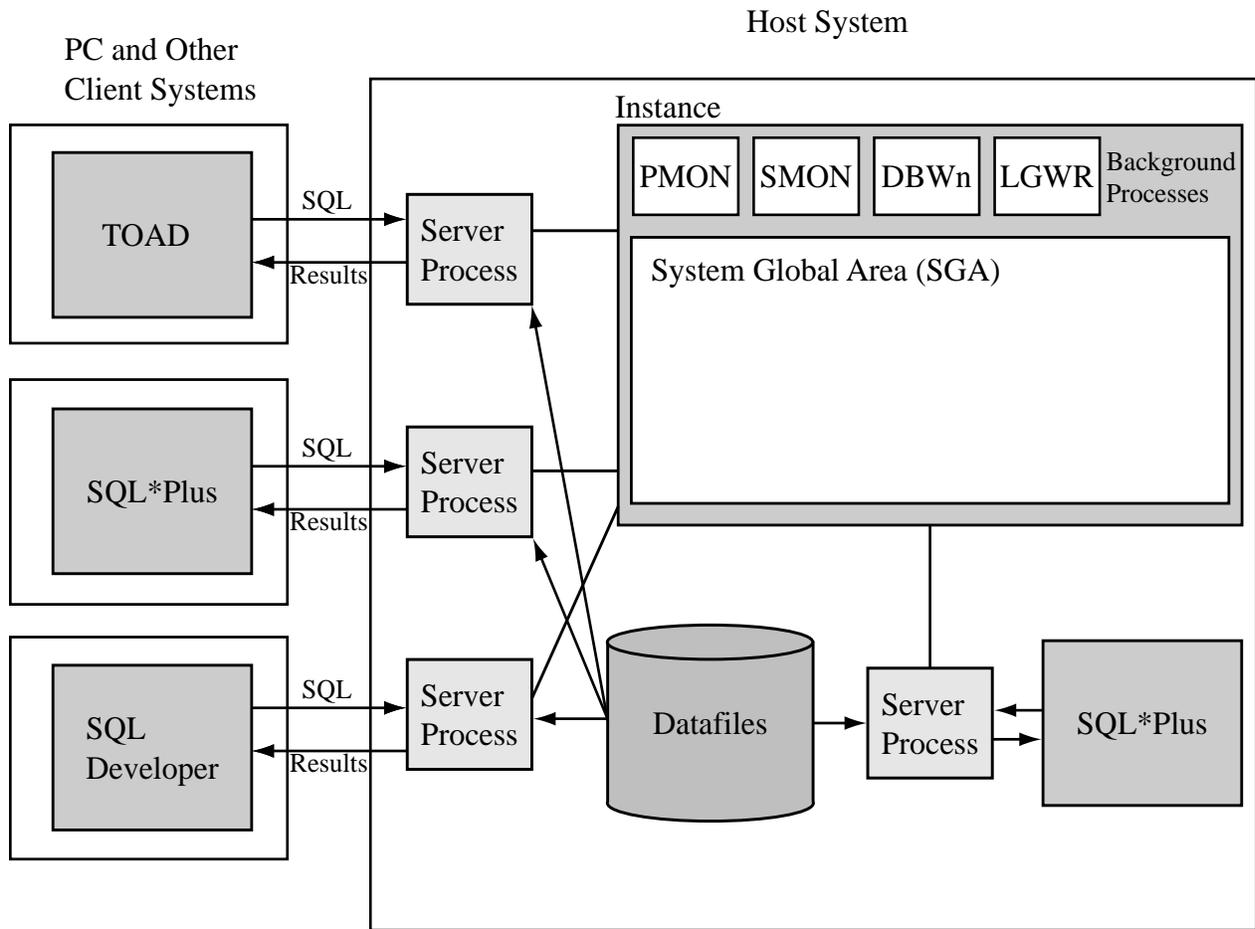
Physical Storage Structures:

- * A tablespace is comprised of 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.



CONNECTING TO A SQL DATABASE

- * An *Oracle instance* is a set of background 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).



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.

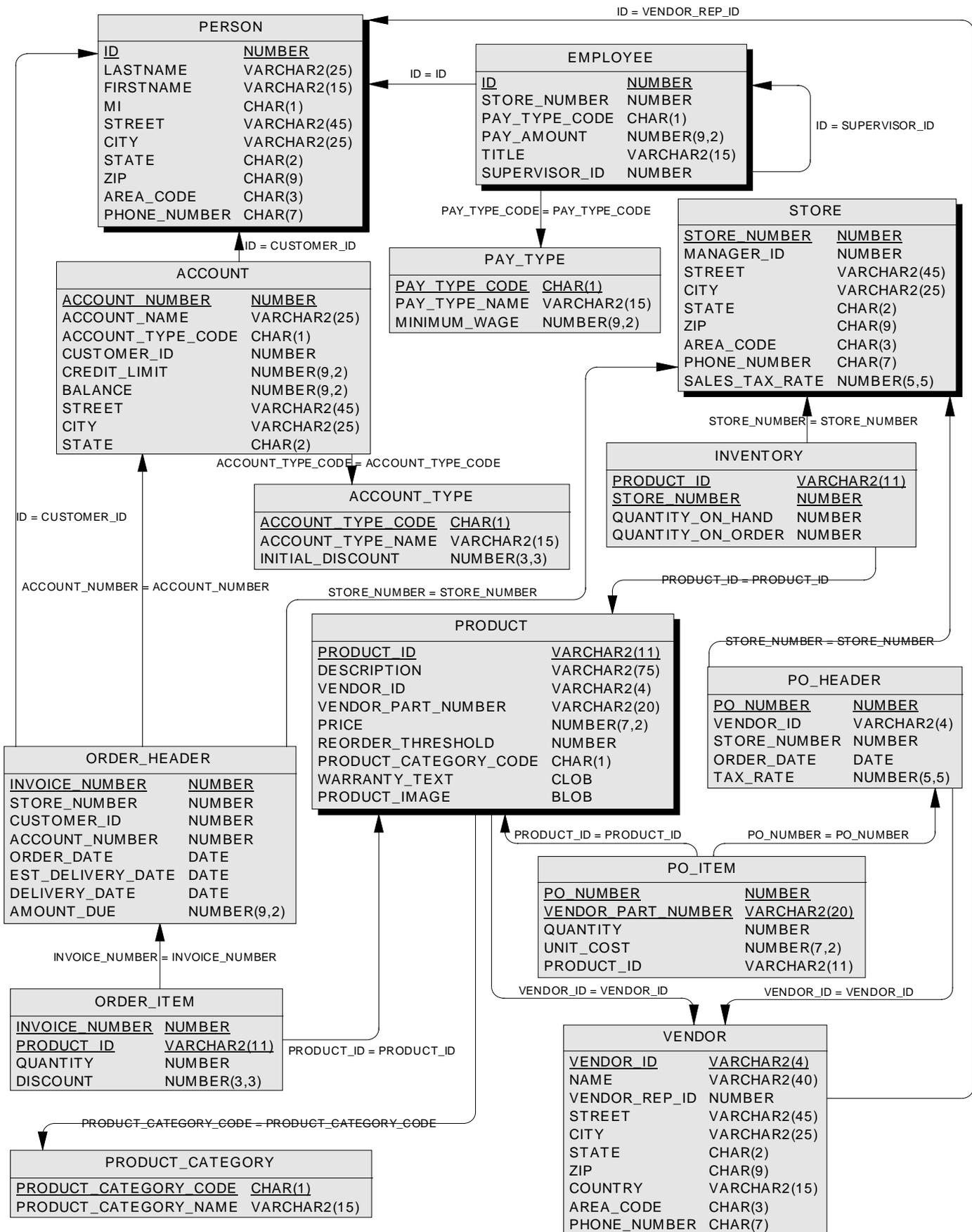
When a credit account is used for an order, the balance for the account must be updated to reflect the total amount of the invoice (including tax). The salesperson verifies (with a phone call) that the person is a valid user of the account.

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.



CHAPTER 9 - DATA DEFINITION AND CONTROL STATEMENTS

OBJECTIVES

- * Describe the datatypes stored in your database.
- * Define your own tables.
- * Control the data allowed in your tables.
- * Modify the definitions of existing tables and columns.
- * Assure the integrity of your database.
- * Drop table definitions from your database.
- * Control access by other users to your tables.

DATATYPES

- * When you create a table, you must choose a datatype for each column within the table.
 - Oracle will issue an error message if you try to **INSERT** or **UPDATE** a value that does not match the column's datatype.
- * String types allow you to hold text in several different formats.
 - **CHAR(size)** — Fixed length, padded strings; these should only be used when you are certain that all records will have the same length strings.
 - **VARCHAR2(size)** — Variable length strings, up to *size*; these are heavily used to store smaller amounts of text.
 - **CLOB** — Arbitrary length strings; these are used for large amounts of text.
- * Numeric types allow you to hold integer and floating point values.
 - **NUMBER(p,s)** — Integer or floating point values where the precision, *p*, and scale, *s*, are optional; these are routinely used to hold dollar amounts and other basic numbers.
 - **BINARY_FLOAT/BINARY_DOUBLE** — 32- or 64-bit floating point values; these are useful for values that need good precision for calculations.
- * Binary types allow you to store data in arbitrary formats, such as images, movies, sounds, or compiled programs.
 - **RAW(size)** — Small binary values; these are typically icons or other small media values.
 - **BLOB** — Arbitrarily large binary values; these are useful for large media values, program code, and other binary data.
- * Date types allow you to hold date/time values.
 - **DATE** — Date/time value with one second granularity; these are general purpose dates that are commonly used in non-globalized databases.
 - **TIMESTAMP** — Date/time value with nanosecond granularity; versions are available to also store time zone information based on the Oracle server or client locale.

String Types	Description	Limits
CHAR(<i>size</i>)	Fixed-length character set data of <i>size</i> length. Default size is 1.	2000 bytes
NCHAR(<i>size</i>)	Fixed-length unicode-only datatype. National character set determines the max length of the column. Default size is 1.	2000 bytes
VARCHAR2(<i>size</i>)	Variable-length character string with max size of <i>size</i> bytes. Must specify size.	4000 bytes
NVARCHAR2(<i>size</i>)	Variable-length character string for national character set with max size of <i>size</i> bytes. The national character set determines the max length of the column.	4000 bytes
CLOB	Character Large Object.	4GB*DB_BLOCK_SIZE
NCLOB	National Character Large Object containing Unicode. National character set determines the max length of the column.	4GB*DB_BLOCK_SIZE
LONG (deprecated)	Variable-length character data. Use CLOB instead.	2GB
Numeric Types	Description	Limits
NUMBER(<i>p,s</i>)	Number with precision of <i>p</i> and scale of <i>s</i> . Must be integer if only <i>p</i> is provided.	Precision is 38, Scale is -84 to 127.
BINARY_FLOAT	32-bit, single-precision floating-point number.	Fixed in length - requires 5 bytes of storage.
BINARY_DOUBLE	Double-precision floating-point number, including the bit length. Supports the values infinity and NaN(not a number).	Fixed in length - requires 9 bytes of storage.
Binary Types	Description	Limits
RAW(<i>size</i>)	Binary data of <i>size</i> bytes. Size must be specified.	2000 bytes
BLOB	Binary Large Object.	4GB*DB_BLOCK_SIZE
BFILE	Reference to a binary file on disk.	4GB
LONG RAW (deprecated)	Binary data of variable length. Use BLOB instead.	2GB
Date Types	Description	Limits
DATE	Date range.	From Jan 1, 4712BC to Dec 31, 9999AD.
TIMESTAMP(<i>fractional_seconds</i>)	All the information contained in date, plus <i>fractional_seconds</i> – the number of digits in the fractional part of the SECONDS datetime field.	<i>fractional_seconds</i> may be 0-9, default is 6.
TIMESTAMP (<i>fractional_seconds</i>) WITH TIME ZONE	Same as above, with timezone displacement value.	0-9, default is 6.
TIMESTAMP (<i>fractional_seconds</i>) WITH LOCAL TIME ZONE	Same as TIMESTAMP WITH TIME ZONE , except the data is adjusted to database time zone when stored in database. When data is queried, users see data in their session time zone.	0-9, default is 6.
INTERVAL YEAR (<i>year_precision</i>) TO MONTH	Period of time in years and months. <i>year_precision</i> is number of digits in YEAR datetime field.	0-9, default is 2.
INTERVAL DAY (<i>day_precision</i>) TO SECOND (<i>fractional_seconds</i>)	Period of time in days, hours, minutes, seconds. <i>day_precision</i> is max number of digits in DAY. <i>fractional_seconds</i> is max number of digits in SECONDS field.	<i>day_precision</i> 0-9, default is 2. <i>fractional_seconds</i> 0-9, default is 6.
Miscellaneous Types	Description	Limits
ROWID (deprecated)	Represents address of row in table.	Only physical rowids.
UROWID	Represents address of a row in a table.	

DEFINING TABLES

- * Create new tables in your schema with the **CREATE TABLE** statement:

```
CREATE TABLE tablename
(
  colname datatype [DEFAULT value] [NOT NULL],
  ...
  colname datatype [DEFAULT value] [NOT NULL]
);
```

- * You must specify a name and datatype, and possibly a data size, for each column.
- * The order of the columns in the **CREATE TABLE** statement will be the order in which the columns are stored.
 - The column order is the default order used when issuing a **SELECT *** or **INSERT**.
- * **NOT NULL** in a column definition specifies that every row in the table must have a non-**NULL** value for that column.
 - **INSERT**s or **UPDATE**s that attempt to violate this will fail.
- * **DEFAULT** specifies a default value to be supplied for a column, if an **INSERT** statement omits a value for the column.
 - Beginning in 9i, this value will also be used for an **UPDATE** or **INSERT** where a column is set to **DEFAULT**.

```
UPDATE tablename
SET columnname = DEFAULT;
```

Physical data storage is managed at different levels. The smallest disk storage space may be a data block or page, which could be as small as 2k-32k (minimum and maximum are often OS-dependent). Contiguous blocks or pages are grouped into extents, which are then used to manage space allocation when creating database objects.

Oracle permits you to specify a number of storage options, such as the tablespace to be used, when you create a table:

pref_cust.sql

```
CREATE TABLE preferred_customer
(
  id NUMBER,
  discount NUMBER(2,2) DEFAULT .05,
  description VARCHAR2(78)
) TABLESPACE users;
```

If the DBA assigned a default tablespace to you, your database objects will automatically be placed there. Otherwise, they will be placed in the **SYSTEM** tablespace or the database wide **DEFAULT TABLESPACE**. You can find what your default tablespace is with the following query:

```
SELECT default_tablespace
FROM user_users;
```

DDL statements (such as **CREATE TABLE** or **DROP TABLE**) are not logged. That is, they are not considered to be part of a transaction and, normally, do not need to be committed (and cannot be rolled back).

Oracle automatically performs a **COMMIT** immediately before and after each DDL statement. This means that if you try to execute a DDL statement within a transaction, your transaction will automatically be committed and you cannot then roll back the transaction.

CONSTRAINTS

* A *constraint* limits the allowed values for a column.

- **PRIMARY KEY** — Unique value of column(s) in all rows. Nulls not allowed. There can only be one primary key per table.

```
PRIMARY KEY (colname[, colname, ...])
```

- **UNIQUE** — Unique value of column(s) in all rows. Nulls allowed.

```
UNIQUE (colname[, colname, ...])
```

- **FOREIGN KEY** — Value(s) in foreign key column(s) must match values in the corresponding primary or unique key column(s) of the referenced table.

```
FOREIGN KEY (colname[, colname, ...])
REFERENCES table [(colname[, colname, ...])]
[ON DELETE {CASCADE|SET NULL}];
```

- The primary key columns in the referenced table will be used if not specified.

- **CHECK** — Value(s) must satisfy the specified condition.

```
CHECK (condition)
```

- **NOT NULL** is similar to a **CHECK** constraint.

```
id NUMBER NOT NULL,
```

```
id NUMBER CHECK (id IS NOT NULL)
```

* Provide a meaningful constraint name when creating the constraint, otherwise the system will provide a default name that is difficult to read in error messages.

```
[CONSTRAINT name] constraint_definition
```

pref_cust2.sql

```
CREATE TABLE preferred_customer
(
  id NUMBER,
  discount NUMBER(2,2) DEFAULT .05,
  description VARCHAR2(78),
  CONSTRAINT pk_pref_cust PRIMARY KEY (id),
  CONSTRAINT fk_prefcust_person FOREIGN KEY (id)
    REFERENCES person (id),
  CONSTRAINT ck_prefcust_discount
    CHECK ( discount BETWEEN 0 AND .25 )
);
```

You can query the Data Dictionary for constraint information:

constraints.sql

```
SELECT table_name, constraint_name,
       CASE constraint_type
         WHEN 'R' THEN 'Foreign Key'
         WHEN 'P' THEN 'Primary Key'
         WHEN 'C' THEN 'Check Constraint'
         WHEN 'U' THEN 'Unique Constraint'
       END "Constraint Type"
FROM user_constraints
ORDER BY 1, 2;
```

foreign_keys.sql

```
SELECT f.table_name || '(' || fc.column_name || ')' references ' ||
       r.table_name || '(' || rc.column_name || ')' "Foreign keys"
FROM user_cons_columns fc JOIN user_constraints f
      ON fc.constraint_name = f.constraint_name
JOIN user_constraints r
      ON f.r_constraint_name = r.constraint_name
JOIN user_cons_columns rc
      ON r.constraint_name = rc.constraint_name
ORDER BY 1;
```

INLINE CONSTRAINTS

- * A constraint involving only one column can be specified in the column definition, using *inline constraint* syntax:

```
colname datatype UNIQUE,
```

```
colname datatype PRIMARY KEY,
```

```
colname datatype REFERENCES table [(colname)],
```

```
colname datatype CHECK (condition),
```

- Multiple constraints can be placed inline and each can be named.

pref_cust3.sql

```
CREATE TABLE preferred_customer
(
  id NUMBER CONSTRAINT pk_prefcust PRIMARY KEY
  CONSTRAINT fk_prefcust_person
  REFERENCES person (id),
  discount NUMBER(2,2) DEFAULT .05
  CONSTRAINT ck_prefcust_discount
  CHECK ( discount BETWEEN 0 AND .25 ),
  description VARCHAR2(78)
);
```

- * Constraint definitions listed at the end of the **CREATE TABLE** statement are *out-of-line constraint* syntax.
 - These syntactic differences for constraint definition have no bearing on the nature of the constraints themselves.
 - Older Oracle documentation referred to inline constraints as "column constraint" syntax, and out-of-line constraints as "table constraint" syntax.

A *foreign key*, or *referential integrity constraint*, is a combination of columns that depends on a primary or unique key in some other table. A *parent row* is a row with foreign key values referencing it; a *parent key* is the referenced primary or unique key in a parent row. A *child row* is the row referencing the parent row. The table containing the parent key is the *parent table* and the table with the foreign key is the *child table*. Before a row is inserted or updated into the child table, the values of the foreign key columns will be checked for rows in the parent table with matching values in the parent key. If there is no match, then the insert will not be allowed. We define this validation of corresponding values as *referential integrity*. Though referential integrity has always been part of the relational model, only recently have RDBMSs begun incorporating primary/foreign key constraints.

To enforce referential integrity, you may determine the action to take on child rows when a parent row value is deleted, the possibilities are:

- **CASCADE** — If the parent row is deleted, delete the child row automatically.
- **SET NULL** — If the parent row is deleted, set the child row's corresponding foreign key values to **NULL**.

Without the **ON DELETE** clause, no action will be taken on the child row. If deleting a parent row will break referential integrity, then the deletion is not allowed. This is the default. The examples opposite use *inline constraint* syntax, which immediately follow the column definition. You may choose instead to use *out-of-line constraint* syntax. Syntactic differences are minor:

- Inline constraints may only refer to one column and are appended directly onto the column definition.
- Out-of-line constraints may refer to one or several columns and are appended onto the end of the table definition.
- Constraints on multiple columns must be out-of-line constraints.
- The **NOT NULL** constraint requires inline constraint syntax.

The example below creates a table tracking office data. An office may have several managers. Each **manager_id** will refer back to a **person** record. If the referenced **person** record is deleted, then the corresponding **office** record is also automatically deleted:

office.sql

```
CREATE TABLE office
(
    office_id NUMBER,
    manager_id NUMBER,
    pager_number CHAR(8) UNIQUE NOT NULL,
    CONSTRAINT fk_office_person FOREIGN KEY (manager_id)
        REFERENCES person (id) ON DELETE CASCADE,
    CONSTRAINT pk_office PRIMARY KEY (office_id, manager_id)
);
```

MODIFYING TABLE DEFINITIONS

- * **ALTER TABLE** changes the definition of an existing table:

```
ALTER TABLE table_name action
```

- * **action** may be any of the following:

- You may **ADD**, **RENAME**, or **DROP** a column (the **DROP** column feature was added in Oracle 8i):

```
ADD columnname column_definition
```

```
RENAME COLUMN columnname TO newcolumnname
```

```
DROP columnname [RESTRICT | CASCADE]
```

- You may **ADD**, **RENAME**, **DROP**, or **MODIFY** the state of a constraint:

```
ADD out-of-line_constraint_definition
```

```
DROP CONSTRAINT cons_name [RESTRICT | CASCADE]
```

- You may **MODIFY** the properties of a column:

```
MODIFY columnname DEFAULT value
```

```
MODIFY columnname DEFAULT NULL
```

```
MODIFY columnname NOT NULL
```

```
MODIFY columnname datatype(size)
```

- * Several actions may be used in one **ALTER TABLE**, but each action type may appear only once per **ALTER TABLE** statement.

You can alter an existing table definition:

ALTER TABLE syntax:

The statement below will add a **salesperson_id** column to the **order_header** table:

alter_order_header.sql

```
ALTER TABLE order_header
    ADD salesperson_id NUMBER REFERENCES person;
```

You can use the **MODIFY** clause to change only certain column characteristics:

- The datatype of a column (if existing row values are a compatible datatype or null).
- The maximum length of a character column or precision of a numeric column.
- The **DEFAULT** value of a column.
- The **NOT NULL** constraint of a column.

alter_order_item.sql

```
ALTER TABLE order_item
    MODIFY product_id NOT NULL;
```

The above will work only if all rows currently have non-null **product_id** values.

You can use **ALTER TABLE** to **ADD** or **DROP** constraints. When adding a constraint, you must use out-of-line constraint syntax. To drop a foreign key or check constraint, use the constraint name (from the **USER_CONSTRAINTS** system catalog table).

alter_pref_cust.sql

```
ALTER TABLE preferred_customer
    DROP PRIMARY KEY;
```

alter_employee.sql

```
ALTER TABLE employee
    ADD hire_date DATE;
```

You can also temporarily turn off constraints (instead of permanently removing them) with the **DISABLE/ENABLE** clauses:

alter_inventory.sql

```
ALTER TABLE inventory DISABLE PRIMARY KEY;
```

This is typically done for bulk loading of data.

DELETING A TABLE DEFINITION

- * **DROP TABLE** removes a table definition, with all of its data, from your schema.

```
DROP TABLE tablename [CASCADE CONSTRAINTS] [PURGE];
```

- Use **CASCADE CONSTRAINTS** to automatically drop any foreign keys that reference this table.
 - Data in the referencing table is not modified or deleted.
 - Without this option, you must use **ALTER TABLE** to remove all foreign key constraints in referencing tables before this table can be dropped.
- Starting in Oracle 10g, dropped tables are preserved in a table called the **recyclebin** and are not actually removed from disk unless the **PURGE** option is specified.
 - You can recover a table from the **recyclebin** with the **FLASHBACK TABLE** statement.
 - Objects are automatically removed from the **recyclebin** when space limites are exceeded.
- * All privileges that have been granted on the dropped table are revoked.
- * All triggers on the table are dropped.
- * Any views, stored procedures, or other objects referencing the dropped table are marked invalid and will be revalidated the next time they are used.

Note:

Every user has a **recyclebin**, which is a Data Dictionary table containing information on the user's dropped objects. You can view it two ways:

```
SELECT * FROM recyclebin;
```

```
SHOW recyclebin;
```

CONTROLLING ACCESS TO YOUR TABLES

- * You control access to all tables in your schema by granting or revoking privileges.

```
GRANT privilege(s) ON table TO {user|PUBLIC};
```

- * You can grant other users privileges to:

- **SELECT** data from your tables.
- **INSERT, UPDATE, and DELETE** data in your tables.
- Create and alter tables or otherwise modify your schema.

```
GRANT INSERT, UPDATE ON product TO clerk3;
```

```
grant_select.sql
```

```
GRANT SELECT ON inventory TO PUBLIC;
```

- * When you grant a privilege to a user, you can allow that user to pass on the same privilege to other users.

```
GRANT UPDATE ON employee TO dobbs  
WITH GRANT OPTION;
```

- * Remove privileges with **REVOKE**:

```
REVOKE privilege(s) ON table FROM {user|PUBLIC};
```

- * A DBA account (**SYSTEM**, or an account with similar privileges) can control privileges on tables in any user's schema.

Roles

You can group several privileges together as a role. You must have the **CREATE ROLE** privilege to do it:

```
CREATE ROLE rolename;
```

To use roles:

1. Create the role.
2. Grant privileges to the role.
3. Grant the role to those users who need those privileges.

role.sql

```
CREATE ROLE dbtester;
```

```
GRANT CREATE SESSION, CREATE TABLE, ALTER ANY TABLE, DROP ANY TABLE  
  TO dbtester;
```

```
GRANT SELECT ANY TABLE, UPDATE ANY TABLE, DELETE ANY TABLE  
  TO dbtester;
```

```
GRANT dbtester TO dobbs;
```

LABS

- ❶ We are creating a preferred customer program.
 - a. Create a table to maintain the list of preferred customers. Each preferred customer will have a discount (normally 5%). We will want to have a brief description for each one. Each preferred customer must have a record in the person table.
(Solution: *create_pref_cust.sql*)
 - b. Add a preferred customer record for everyone who has placed an order. For the preferred customer description, use 'Special Order Customer.'
(Solution: *insert_pref_cust.sql*)
- ❷ Upon examination, the existing definition of our database has several omissions. Make the appropriate changes to the database definition:
 - a. The database should ensure that every store manager is indeed a current employee.
(Solution: *store_manager.sql*)
 - b. Each account should have its own discount.
(Solution: *account_discount.sql*)
 - c. On order headers we need to be able to list the id of the salesperson who took the order.
(Solution: *sales_person.sql*)
- ❸ Create a table called **city** containing the name of each distinct city and state in the person table. Define a compound primary key for this table.
(Solution: *city.sql*)
- ❹ Create another table called **calif_person** containing the id, firstname, lastname, city, and state of each person in California. Each person's city and state must exist in the **city** table.
(Solution: *calif_person.sql*)
- ❺ (Optional) Delete Los Angeles from the **city** table. Can you? How?
(Solution: *delete_la.sql*)
- ❻ (Optional) Drop the **city** table. Can you? How?
(Solution: *drop_city.sql*)

Oracle provides a convenient syntax for creating a table and populating it with data from an existing table:

```
CREATE TABLE table_name AS subquery;
```

The columns of the new table will have the same names and datatypes as the columns in the **SELECT** list of the subquery.

```
CREATE TABLE area_codes AS
  SELECT DISTINCT area_code, state
     FROM person
     WHERE area_code IS NOT NULL;
```

The new table will have no constraints defined, so they would have to be added afterwards:

```
ALTER TABLE area_codes
  ADD CONSTRAINT pk_area_codes PRIMARY KEY (area_code, state);
```

The alternative is to create the table, then copy the rows:

```
CREATE TABLE area_codes (
  area_code CHAR(3),
  state     CHAR(2),
  CONSTRAINT pk_area_codes PRIMARY KEY (area_code, state)
);

INSERT INTO area_codes
  SELECT DISTINCT area_code, state
     FROM person
     WHERE area_code IS NOT NULL;
```


CHAPTER 18 - BULK OPERATIONS

OBJECTIVES

- * Utilize bulk binding techniques to improve PL/SQL performance.
- * Use the **BULK COLLECT** clause to retrieve multiple records at once from a table.
- * Issue a DML statement against an entire collection with the **FORALL** statement.
- * Retrieve affected DML records into your program with bulk returns.
- * Retrieve cursor data all at once or in chunks with **BULK FETCH**.

BULK BINDING

- * SQL statements issued from a PL/SQL block are sent to the SQL engine for execution.
 - This involves a context switch between the SQL and PL/SQL engines, as well as data transfer between the two execution environments.
- * Certain techniques allow you to minimize this overhead by allowing more data to be transferred at one time.
- * Bulk binding allows an entire collection to be passed between the two engines for greatly improved performance.
 - Collection types that can be used in bulk bind operations are **VARRAY**, **TABLE**, and associative arrays that are indexed by integers.

The following program loops seven times, issuing seven **SELECT** statements and seven **INSERT** statements. This is a total of 28 context switches, two for each SQL statement.

store_history.sql

```
CREATE TABLE store_history(  
    store_num NUMBER,  
    emp_count NUMBER,  
    dt DATE)  
/  
  
DECLARE  
    ecount NUMBER;  
BEGIN  
    FOR s IN 1..7 LOOP  
        SELECT count(id)  
            INTO ecount  
            FROM employee  
            WHERE store_number = s;  
        INSERT INTO store_history  
            VALUES(s, ecount, sysdate);  
    END LOOP;  
END;  
/
```

BULK COLLECT CLAUSE

- * Use the **BULK COLLECT** clause to bring a large quantity of data into a PL/SQL collection.
 - **BULK COLLECT** works with **VARRAY**, **TABLE**, and associative arrays that use integers for the index.
- * The **SELECT** statement uses a **BULK COLLECT** clause to populate collections.

```
SELECT lastname, firstname
       BULK COLLECT INTO all_lastnames, all_firstnames
FROM person;
```

- To limit the number of rows returned into the collections, use **ROWNUM** in the **WHERE** clause of the **SELECT** statement.

```
SELECT lastname, firstname
       BULK COLLECT INTO all_lastnames, all_firstnames
FROM person
WHERE rownum < 100;
```

The following program uses **BULK COLLECT** with a **SELECT** statement to obtain all of the employee information at once. It then loops seven times, issuing seven **INSERT** statements. This is a total of 16 context switches, two for each **INSERT** statement and two for the single **SELECT** statement.

bulk_store_history.sql

```
CREATE TABLE store_history(  
    store_num NUMBER,  
    emp_count NUMBER,  
    dt DATE)  
/  
  
DECLARE  
    TYPE num_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;  
    snums num_type;  
    counts num_type;  
BEGIN  
  
    SELECT store_number, count(id)  
        BULK COLLECT INTO snums, counts  
    FROM employee  
    WHERE store_number IS NOT NULL  
    GROUP BY store_number;  
  
    FOR s IN 1..snums.COUNT LOOP  
        INSERT INTO store_history  
            VALUES(snums(s), counts(s), sysdate);  
    END LOOP;  
END;  
/
```

FORALL STATEMENT

- * A **FORALL** statement is applied to a single DML statement, issuing the **INSERT**, **UPDATE**, or **DELETE** for multiple values of a collection.

```
FORALL s IN 1..snums.COUNT
  INSERT INTO store_history
    VALUES(snums(s), counts(s), sysdate);
```

- The **FORALL** statement is used in place of a traditional PL/SQL looping construct, which would invoke the DML statement repeatedly.
- * A single roundtrip between the SQL and PL/SQL engine happens, with the entire collection being sent at once.
 - Oracle's documentation states that any collection of four or more elements will perform better with this bulk operation.
- * Values from the collection must be used within the DML statement.
 - One restriction is that collections must be indexed by plain variables, not by expressions.

The following program uses **BULK COLLECT** with the **SELECT** statement to obtain all of the employee information at once. It then uses **FORALL** to process all values within the collections, issuing one **INSERT** statement. This is a total of four context switches, two for the **SELECT** and two for the **INSERT**.

forall_store_history.sql

```
CREATE TABLE store_history(  
    store_num NUMBER,  
    emp_count NUMBER,  
    dt DATE)  
/  
  
DECLARE  
    TYPE num_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;  
    snums num_type;  
    counts num_type;  
BEGIN  
  
    SELECT store_number, count(id)  
        BULK COLLECT INTO snums, counts  
        FROM employee  
        WHERE store_number IS NOT NULL  
        GROUP BY store_number;  
  
    FORALL s IN 1..snums.COUNT  
        INSERT INTO store_history  
            VALUES(snums(s), counts(s), sysdate);  
END;  
/
```

Note:

As of 10g, if you create a collection of records, you cannot access individual record fields from within the DML statement inside of the **FORALL**. Workarounds for this exist, including creating individual collections as shown above, or using database object types and casting with the **TREAT()** function.

FORALL VARIATIONS

- * **FORALL** has three ways of iterating through the elements in the collection.
- * When, using a range similar to the traditional **FOR** loop, all elements in the range must exist in the collection.

```
FORALL index_var IN lower_range .. upper_range
```

- * Use the **INDICES OF** clause when working with indexes that might not be consecutive.

```
FORALL index_var IN INDICES OF collection
```

- * Use **VALUES OF** when one collection contains the indices of a second collection.

```
FORALL index_var IN VALUES OF  
collection_containing_index
```

- Both **INDICES OF** and **VALUES OF** were introduced in Oracle 10g.

This example goes through a collection of **store_numbers** and gives the employees a raise based on values in a raise collection.

raise1.sql

```

DECLARE
  TYPE raise_amt_type IS TABLE OF number;
  TYPE store_no_type IS TABLE OF number;
  raise raise_amt_type :=
    raise_amt_type(1.06,1.1,1.07,1.05,1.1,1.12,1.02);
  stores store_no_type := store_no_type(1,2,3,4,5,6,7);
BEGIN
  FORALL j IN 1..raise.COUNT
    UPDATE employee SET pay_amount = pay_amount * raise(j)
      WHERE store_number = stores(j);
END;
/

```

Store #3 is deleted from the collection of stores making the table no longer consecutive.

raise2.sql

```

DECLARE
  TYPE raise_amt_type IS TABLE OF number;
  TYPE store_no_type IS TABLE OF number;
  raise raise_amt_type :=
    raise_amt_type(1.06,1.1,1.07,1.05,1.1,1.12,1.02);
  stores store_no_type := store_no_type(1,2,3,4,5,6,7);
BEGIN
  stores.DELETE(3);
  FORALL j IN INDICES OF stores
    UPDATE employee SET pay_amount = pay_amount * raise(j)
      WHERE store_number = stores(j);
END;
/

```

In the following example, the **store_indexes** collection contains the indexes of the stores that should get 10% raises.

raise3.sql

```

DECLARE
  TYPE index_type IS TABLE OF pls_integer;
  TYPE store_no_type IS TABLE OF number;
  store_indexes index_type := index_type(2,5,7);
  stores store_no_type := store_no_type(1,2,3,4,5,6,7);
BEGIN
  FORALL j IN VALUES OF store_indexes
    UPDATE employee SET pay_amount = pay_amount * 1.1
      WHERE store_number = stores(j);
END;
/

```

BULK RETURNS

- * Use **BULK COLLECT** in conjunction with the **RETURNING** clause of **INSERT**, **UPDATE**, and **DELETE** statements to return information about affected rows.
- * **RETURNING...BULK COLLECT** will cause all of the data from one column to be placed into a single collection.

```
DELETE FROM employee
WHERE store_number IS NULL
RETURNING id BULK COLLECT INTO deleted_emps;
```

- * Combining **RETURNING...BULK COLLECT** with the **FORALL** statement allows data to be appended into the collection on each iteration of the loop.

```
FORALL s IN INDICES OF stores
UPDATE employee
SET pay_amount = pay_amount * (1 +
(SELECT SUM(quantity * price) * .00001
FROM order_item oi JOIN order_header oh
USING (invoice_number)
JOIN product p
USING (product_id)
WHERE store_number = stores(s)))
WHERE store_number = stores(s)
RETURNING id, pay_amount
BULK COLLECT INTO updated_emps, new_pay;
```

- As the **FORALL** is processing the first collection, performing the DML statement on each element, it is appending into a second collection in chunks.
 - This combination is very efficient, as each collection is only passed once between the SQL and PL/SQL engines.
- Using a bulk return with a **FOR** loop would cause the second collection to be overridden with the data from each successive iteration.

bulk_return.sql

```

CREATE TABLE testoutput ( output NUMBER )
/

DECLARE
    TYPE emp_num_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    deleted_emps emp_num_type;
BEGIN
    DELETE FROM employee
        WHERE store_number IS NULL
    RETURNING id BULK COLLECT INTO deleted_emps;
    FOR i IN 1..deleted_emps.COUNT LOOP
        INSERT INTO testoutput (output)
            VALUES ( deleted_emps(i) );
    END LOOP;
END;
/

SELECT output FROM testoutput
/

```

bulk_forall.sql

```

CREATE TABLE testoutput ( output VARCHAR2(3000) )
/

DECLARE
    TYPE store_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    stores store_type;

    TYPE id_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    updated_emps id_type;

    TYPE amount_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    new_pay amount_type;
BEGIN
    SELECT store_number
        BULK COLLECT INTO stores
    FROM store
    WHERE manager_id IS NOT NULL;

    FORALL s IN INDICES OF stores
        UPDATE employee
            SET pay_amount = pay_amount * (1 +
                (SELECT SUM(quantity * price) * .00001
                 FROM order_item oi JOIN order_header oh
                 USING (invoice_number)
                 JOIN product p
                 USING (product_id)
                 WHERE store_number = stores(s)))
            WHERE store_number = stores(s)
    RETURNING id, pay_amount
    BULK COLLECT INTO updated_emps, new_pay;

    FORALL i IN INDICES OF updated_emps
        INSERT INTO testoutput (output)
            VALUES ( updated_emps(i) || ' now makes $' || new_pay(i));
END;
/

SELECT output FROM testoutput
/

```

BULK FETCHING WITH CURSORS

- * Combine **BULK COLLECT INTO** with a **FETCH** to gain the performance benefits of bulk binding with your cursors.
 - Traditional cursor processing retrieves a single row at a time, with the **FETCH** retrieving just a single record.
- * Fetching into one or more collections will cause the entire result set to be pulled into your program at once.

```
FETCH order_curs  
BULK COLLECT INTO cur_orders;
```

- This technique can also be used with a **REF CURSOR**, which gives the calling program some flexibility in how to process the cursor.
- * To process a subset of rows, add a **LIMIT** clause to your bulk fetch.

```
FETCH order_curs  
BULK COLLECT INTO cur_orders LIMIT 10;
```

- Limiting the bulk fetch will allow you to keep the memory requirements down, while still gaining the benefits of bulk binding.
- When limiting a bulk fetch, you must use **COUNT** on the collection to determine when to quit fetching, as the cursor's **%FOUND** and **%NOTFOUND** attributes will be inaccurate.

```
WHILE cur_orders.COUNT != 0 LOOP
```

bulk_order_loop.sql

```
CREATE TABLE bad_credit (  
    account_number NUMBER,  
    over_limit NUMBER(9,2)  
)  
/  
  
DECLARE  
    CURSOR order_curs IS  
        SELECT account_number, SUM(amount_due) AS total_due  
            FROM order_header  
            WHERE amount_due IS NOT NULL  
              AND amount_due > 0  
              AND account_number IS NOT NULL  
            GROUP BY account_number;  
  
    TYPE order_tab_type IS TABLE OF order_curs%ROWTYPE  
        INDEX BY PLS_INTEGER;  
    cur_orders order_tab_type;  
    bal account.balance%TYPE;  
    limit account.credit_limit%TYPE;  
    remaining NUMBER;  
BEGIN  
    OPEN order_curs;  
    FETCH order_curs  
    BULK COLLECT INTO cur_orders LIMIT 10;  
    WHILE cur_orders.COUNT != 0 LOOP    -- can't use %FOUND  
  
        FOR i IN 1..cur_orders.COUNT LOOP  
            -- process records  
            SELECT balance, credit_limit  
                INTO bal, limit  
                FROM account  
                WHERE account_number = cur_orders(i).account_number;  
  
            remaining := limit - bal;  
            IF remaining < cur_orders(i).total_due THEN  
                INSERT INTO bad_credit  
                    VALUES (cur_orders(i).account_number,  
                        cur_orders(i).total_due - remaining);  
            END IF;  
        END LOOP;  
  
        FETCH order_curs  
        BULK COLLECT INTO cur_orders LIMIT 10;  
    END LOOP;  
  
    CLOSE order_curs;  
  
END;  
/
```

LABS

- ❶ Create a nested table of employee rowtype records. Use **BULK COLLECT** to populate the table with all the employees of the Las Vegas store.
(Solution: *las_vegas_bulk.sql*)

- ❷ Write an anonymous block to store information about accounts and their credit rating.
 - a. Create a **credit_rating** table that contains columns for an account number and a credit rating.

 - b. Retrieve an account number and total of all **amount_due** data for each account from the **order_header** table. Use two collections to hold this data for all accounts that have had orders placed.

 - c. Process each account in the collections using **FORALL**. Place accounts in the **credit_rating** table along with their credit rating, according to the following formula:
rating = amount_due / (limit - balance) * 100
Note: Round the credit ratings to the nearest whole number.
(Solution: *credit_rating.sql*)

