Introduction to Oracle 11g Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

INTRODUCTION TO ORACLE 11G 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

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 stucture 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:** This course is designed for database application developers.
- **Prerequisites:** Familiarity with relational database concepts as well as a solid understanding of 3GL programming are required.
- **Student Materials:**
  - Student workbook
- **Classroom Environment:**
  - One workstation per student
  - Oracle 11*g*, Oracle Server, and SQL*Plus.

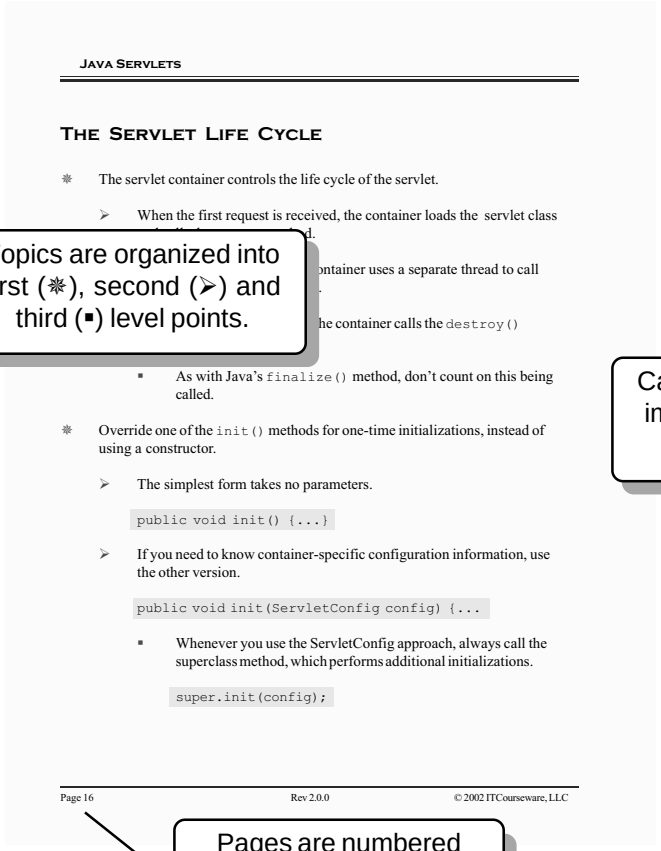
USING THE WORKBOOK

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.

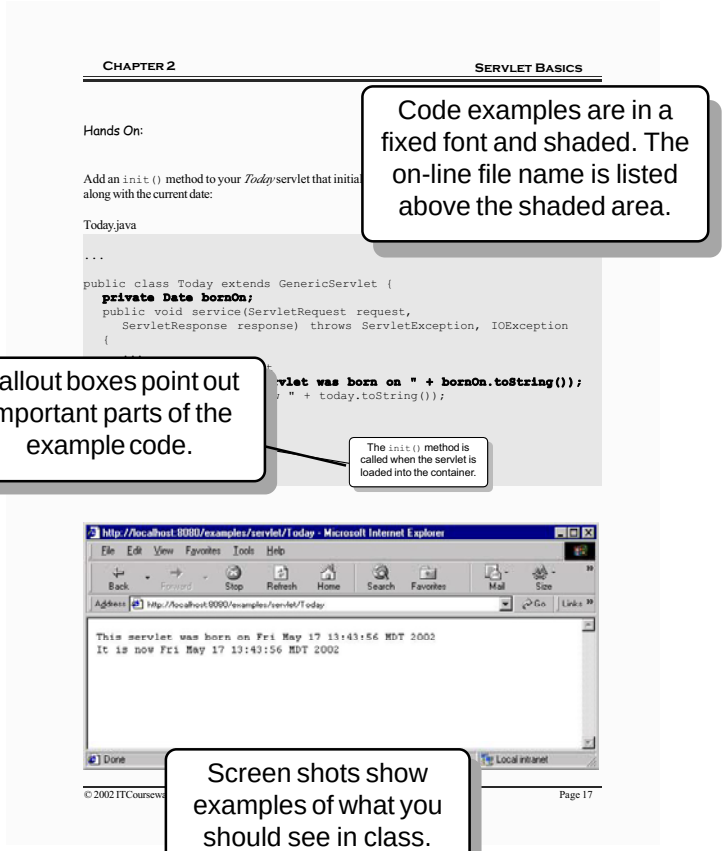
The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.



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

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



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.

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

McLaughlin, Michael. 2008. ***Oracle Database 11g PL/SQL Programming***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0071494456

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

Urman, Scott and Michael McLaughlin. 2004. ***Oracle Database 10g PL/SQL Programming***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072230665

http://asktom.oracle.com
http://tahiti.oracle.com
http://www.dbasupport.com
http://www.hot-oracle.com
http://www.oracle.com
http://www.oracle.com/technology/tech/pl_sql
http://www.searchdatabase.com
http://www.toadworld.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 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.

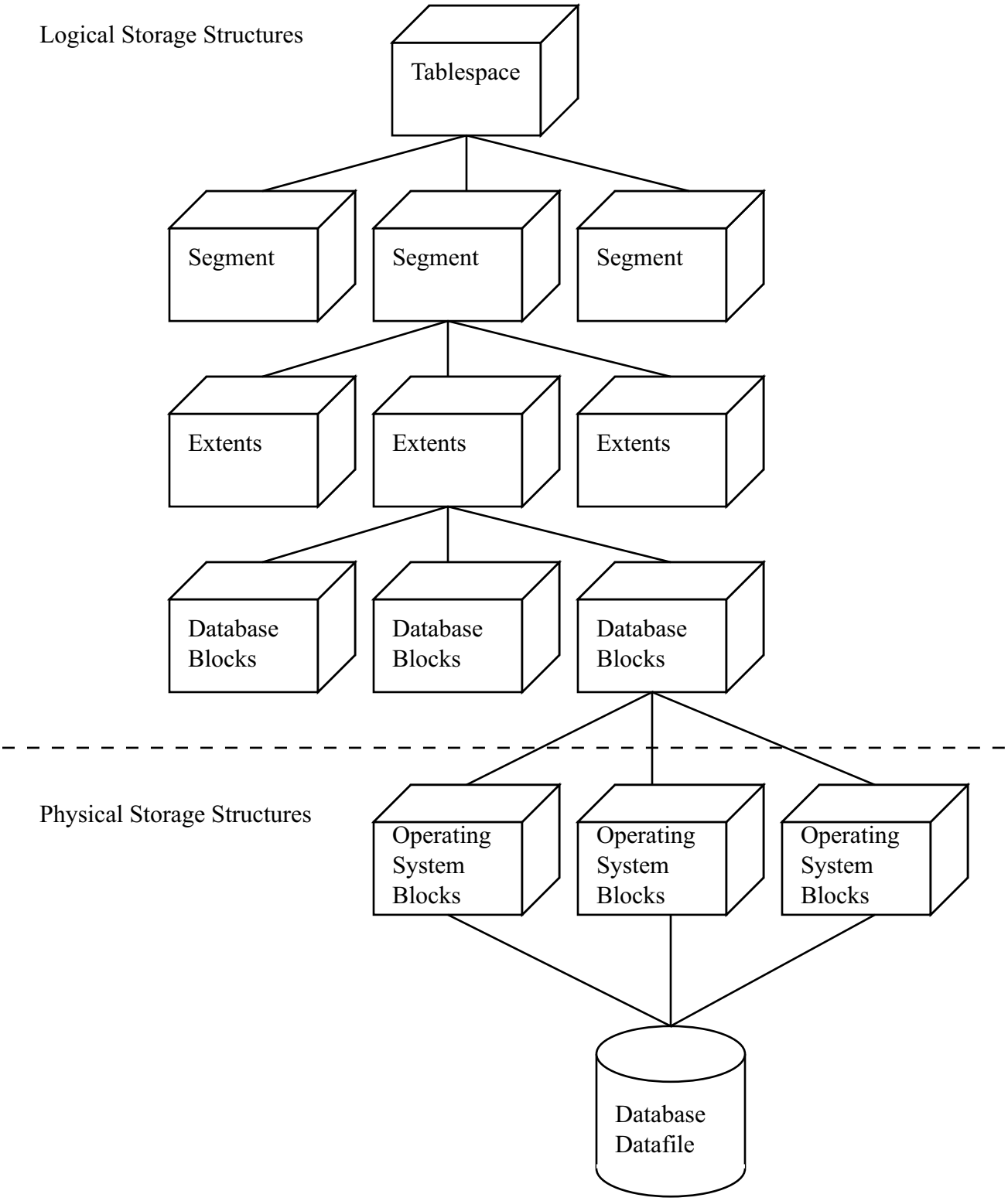
LOGICAL AND PHYSICAL STORAGE STRUCTURES

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.

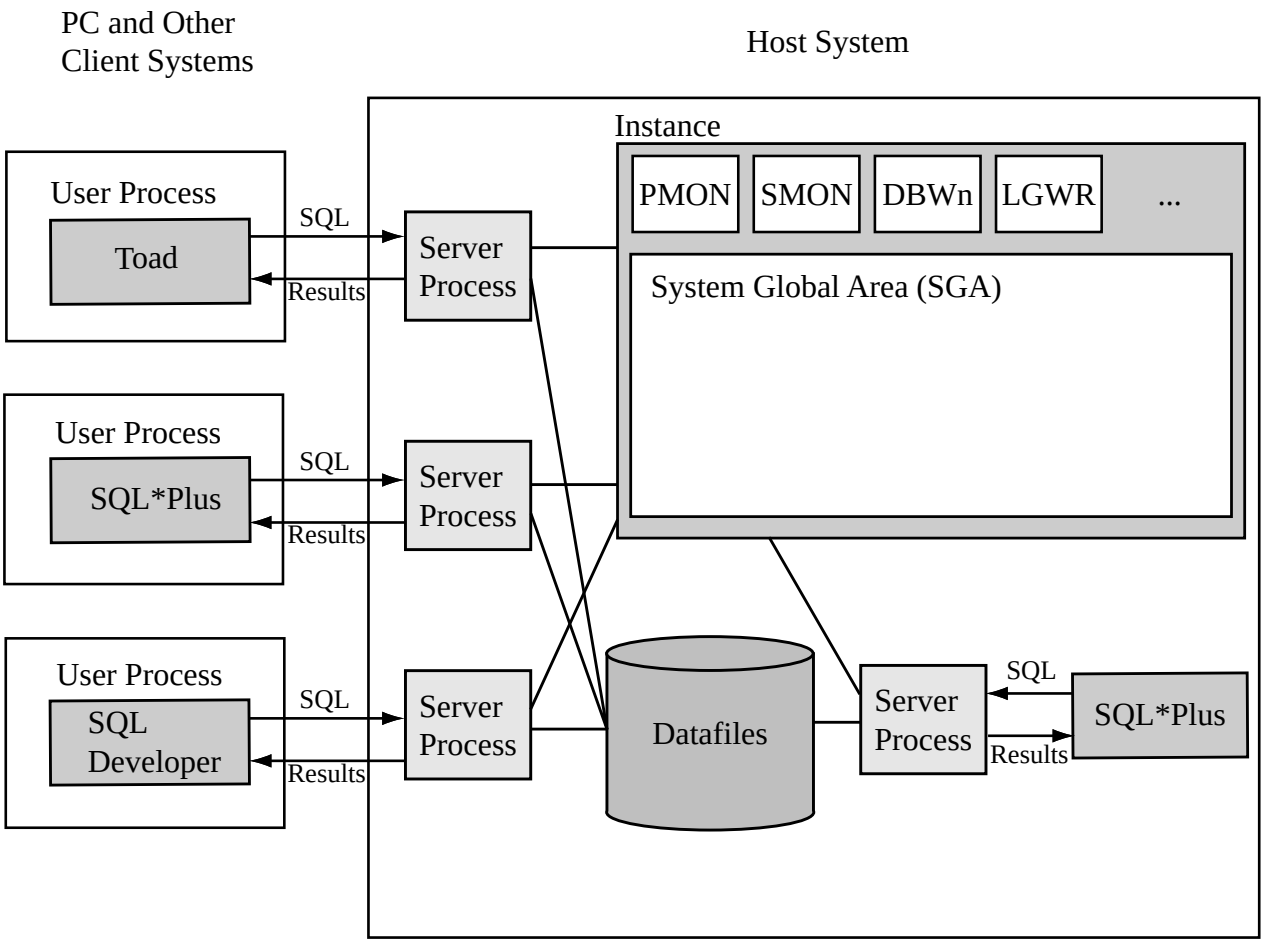
Physical Storage Structures:

- 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
PMON
SMON
DBWn
LGWR
...
System Global Area (SGA)
User Process
Toad
SQL
Results
Server
Process
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.

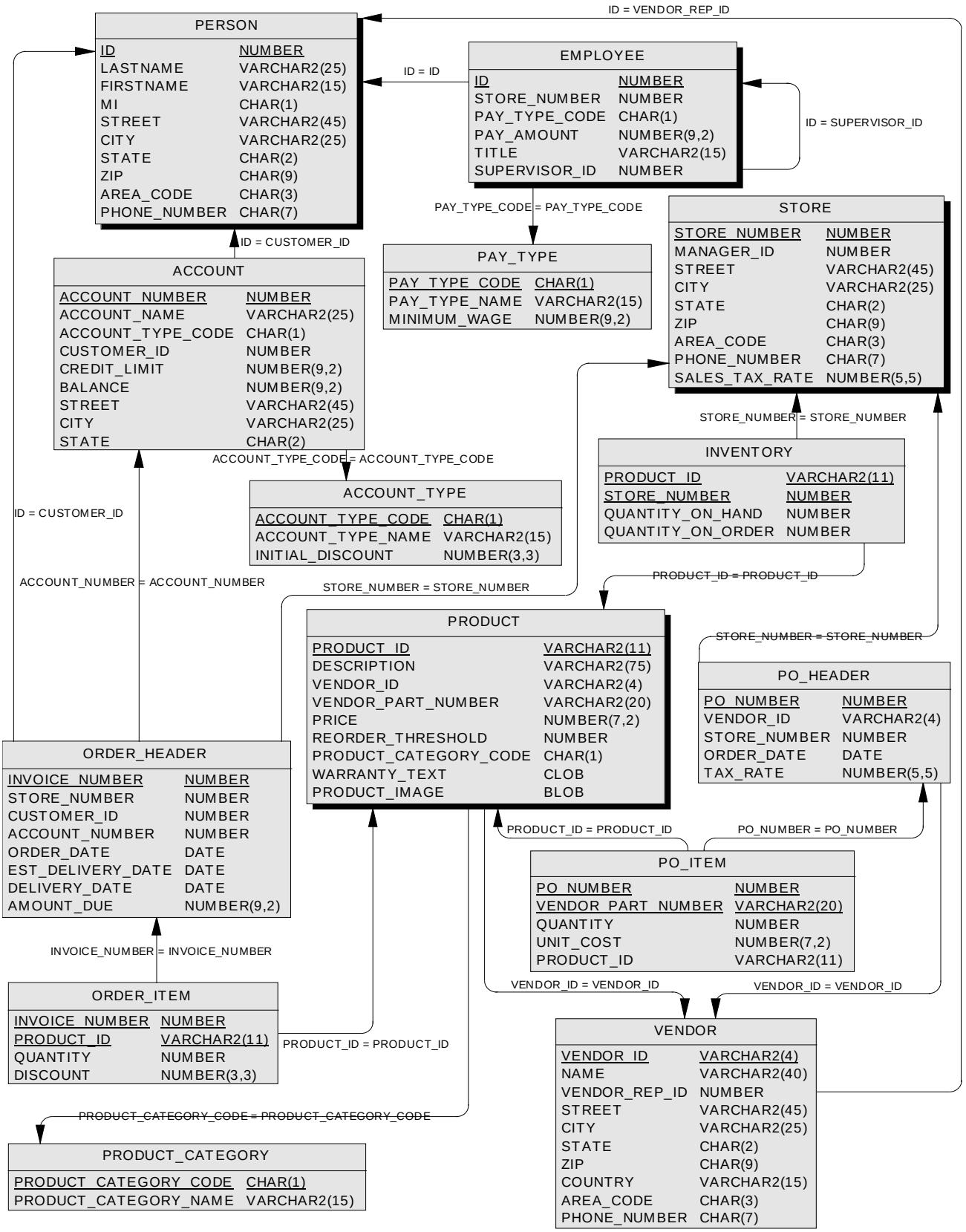
When a credit account is used for an order, the balance for the account must be updated to reflect the total amount of the invoice (including tax). The salesperson verifies (with a phone call) that the person is a valid user of the account.

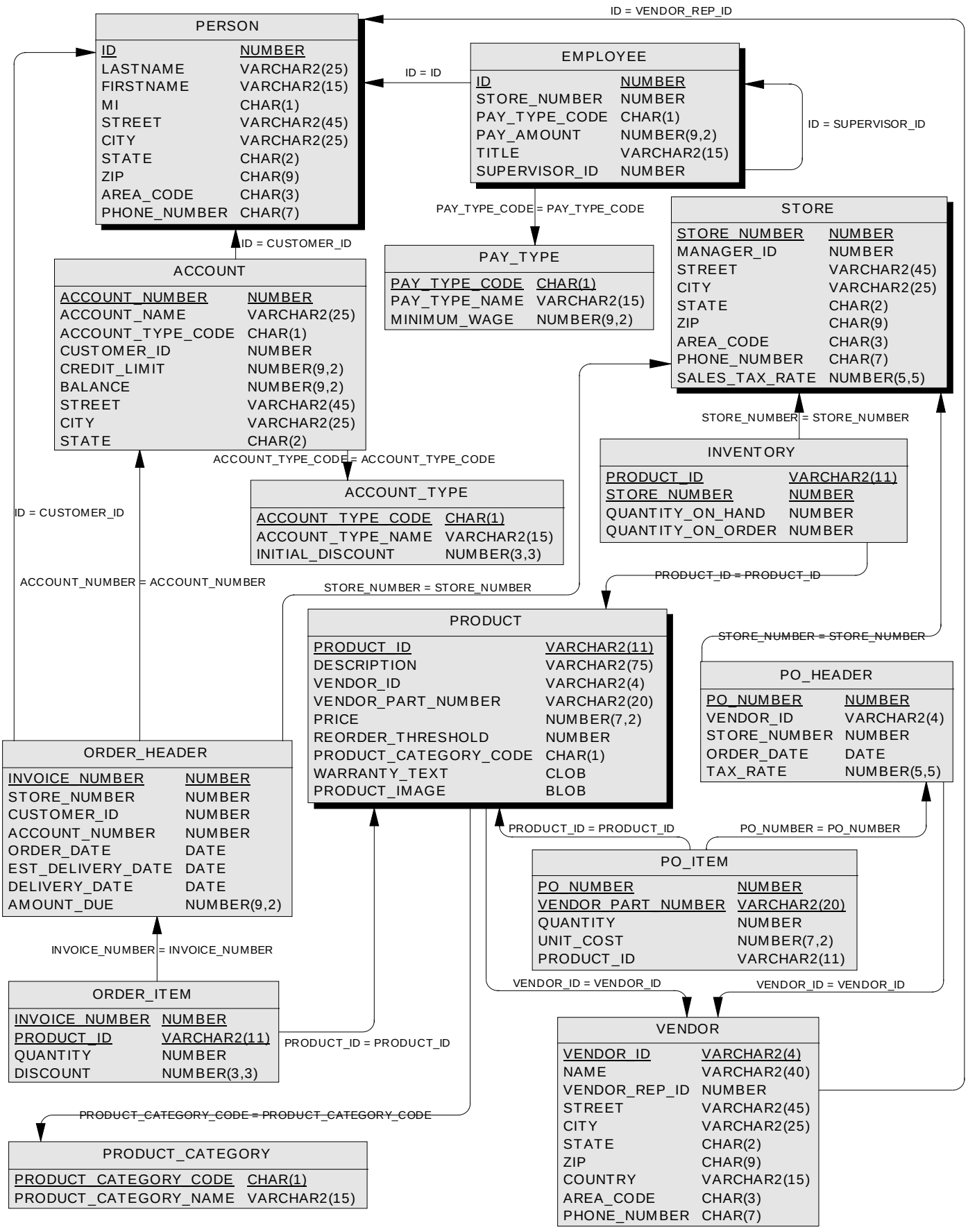
Each product we sell has a product ID and description. In addition, we keep track of the vendor from whom we purchase that product, the ID for that product, and the category ("Software," "Peripheral," or "Service"). We also store the address, phone, and sales rep ID for each individual vendor.

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)
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 = VENDOR_REP_ID
ID = SUPERVISOR_ID
PAY_TYPE_CODE = PAY_TYPE_CODE
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)
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)
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 8 - DATA MANIPULATION AND TRANSACTIONS

OBJECTIVES

- Add new rows to database tables.
- Modify rows in tables.
- Delete rows from tables.
- Perform tasks consisting of more than one SQL statement.
- Make your changes permanent and visible to other users.
- Undo the effects of SQL statements.
- Access database tables concurrently with other users.
- Prevent other users from interfering with your changes.

THE INSERT STATEMENT

- Use the **INSERT** statement to add new rows of data to a table.

```
INSERT INTO table_name [(column_name_list)]
  {VALUES (value_list) | subquery };
```

  - Use ***column_name_list*** to specify the columns for which you will be providing values.
  - You must own the table or have **INSERT** privilege on the table.

- To add a single row to the table, use the **VALUES** clause.

insert_order.sql

```
INSERT INTO order_header
   (invoice_number, store_number, customer_id,
    order_date, est_delivery_date, amount_due)
  VALUES (3001, 7, 7878, sysdate, sysdate + 4, 0);
```

  - The order and number of values must match the column list.

- To add zero or more rows at once by copying from existing data, use a subquery.

insert_items.sql

```
INSERT INTO order_item
  (SELECT 3001, product_id, 1, .05
     FROM inventory
    WHERE store_number = 7 AND product_id like 'IBM%'
      AND quantity_on_hand > 0);
```

  - The subquery's **SELECT** list can include column values, expressions, and literals.
    - You can generate rows in which some column values come from existing records, while others are provided by the statement itself.

- Use the **NULL** or **DEFAULT** keywords in place of a value to insert a null, or allow a column to be populated with its default value.

inv.sql

```
INSERT INTO inventory
      VALUES ('BORL0000014', 7, 25, NULL);
```


A company employee may enter a product into the inventory at Store #7 with the following:

inv.sql
```
INSERT INTO inventory
VALUES ('BORL0000014', 7, 25, NULL);
```

This format of the **INSERT** command requires that the number and exact order of all columns in the **inventory_item** table be known. If the table definition should change, this statement would become invalid. A more reliable (and maintainable) format is to specify the order and names of the columns that are being populated:

inv2.sql
```
INSERT INTO inventory (product_id, store_number, quantity_on_hand,
                       quantity_on_order)
VALUES ('BORL0000014', 7, 25, NULL);
```

This format is required when populating only a subset of all columns in a table. Any skipped columns will be automatically assigned the specified default or (if allowed) null value. The **INSERT** below is equivalent to those above:

inv3.sql
```
INSERT INTO inventory (product_id, store_number, quantity_on_hand)
VALUES ('BORL0000014', 7, 25);
```

A subquery can be used to easily populate rows from other data in the database. To add another person to the database with the same address and phone information as an existing person:

person.sql
```
INSERT INTO person (id, lastname, firstname, mi, street, city, state,
                    zip, area_code, phone_number)
       (SELECT 9999, 'Striker', 'Joan', 'R', street, city, state, zip,
               area_code, phone_number
          FROM person
         WHERE id=7881);
```

THE UPDATE STATEMENT

- Use the **UPDATE** statement to change existing data in a single table.

```
UPDATE table_name
  SET column=expression [,column=expression,...]
[WHERE condition];
```

  - You must either own the table or have **UPDATE** privilege.

update_inv.sql

```
UPDATE inventory
   SET quantity_on_hand = 42,
       quantity_on_order = 0
 WHERE store_number = 7
   AND product_id = 'BORL0000014';
```

- The ***expression*** may be a subquery that must return only one row.

update_emp.sql

```
UPDATE employee
   SET supervisor_id = (SELECT manager_id FROM store
                        WHERE store_number = 7)
 WHERE store_number = 7
   AND supervisor_id IS NULL;
```

  - 9*i* and later versions allow you to specify **DEFAULT** to set a value back to the default value specified for this column when the table was created.

```
UPDATE employee SET supervisor_id = DEFAULT
 WHERE id = 6535;
```

    - A **NULL** will be issued if there is no default for this column.

- The **WHERE** clause identifies the rows to be updated.

  - If no **WHERE** clause is used, all rows in ***table_name*** are updated.

With a correlated subquery, we can update each employee based on the store at which they work.

update2.sql
```
UPDATE employee
   SET supervisor_id =
       (SELECT manager_id
          FROM store
         WHERE store_number = employee.store_number)
 WHERE supervisor_id IS NULL;
```

In this example, however, the subquery must be executed once for every row processed by the **UPDATE**, since the value returned by the **SELECT** will depend on the value of the **employee.store_number** in the current row being updated.

Oracle also allows several columns to be set with one subquery:

update_person.sql
```
UPDATE person
   SET (area_code, phone_number) =
       (SELECT area_code, phone_number
          FROM store
         WHERE store_number = 7)
 WHERE id = 7881;
```

THE DELETE STATEMENT

- The **DELETE** statement removes rows from a table.
- You must either own the table or have **DELETE** privilege.
- The format of the **DELETE** command is:

```
DELETE FROM table_name
[WHERE condition];
```

- The **WHERE** clause identifies the rows to be deleted.

delete.sql

```
DELETE FROM order_item
  WHERE invoice_number = 2960
    AND product_id = 'ORCL0000014';
```

  - The ***condition*** can be the same as those found in the **SELECT** statement.
  - If no **WHERE** clause is used, all rows in ***table_name*** are deleted!

In addition, Oracle provides the **TRUNCATE** statement. **TRUNCATE** deletes all of the rows in a table. Unlike the **DELETE** statement, you cannot specify any conditions on which rows are to be deleted. The example below will delete the contents of the **person** table:

```
TRUNCATE TABLE person;
```

TRUNCATE is used instead of the **DELETE** statement to free space allocated for the table; and, because each deletion is not logged, the **TRUNCATE** statement is usually faster than **DELETE**. The **TRUNCATE** statement cannot be undone (see **ROLLBACK**) and should, therefore, be used with caution.

TRANSACTION MANAGEMENT

- Once you connect to a database, your statements are all considered to be a single logical unit of work, called a *transaction*.

- At any point in a transaction, you can roll back all of your DML statements.

```
ROLLBACK;
```

  - All of your changes are discarded and a new transaction begins.

- None of your changes are visible in the database until you commit your transaction.

```
COMMIT;
```

  - All your changes are made permanent and a new transaction begins.

- When should you **COMMIT** or **ROLLBACK**?

  - Commit as soon as you have completed the statements making up a logical unit of work.

  - Remember:

    - **COMMIT** commits everything since the last commit or rollback.

    - **ROLLBACK** rolls back everything since the last commit or rollback.

- Add savepoints to your transactions to give yourself a place to partially roll back to.

```
SAVEPOINT a;
```

```
ROLLBACK TO a;
```


Transactions help you manage tasks that must succeed as a logical group. To change an employee's id from 8119 to 519, for example, you must also change the corresponding person record:

1. Insert a new person record:

```
INSERT INTO person (SELECT 519, lastname, firstname, mi,
                           street, city, state, zip,
                           area_code, phone_number
                      FROM person
                     WHERE id = 8119 );
```

2. Update the employee record:

```
UPDATE employee SET id = 519 WHERE id = 8119;
```

3. Delete the old person record:

```
DELETE FROM person WHERE id = 8119;
```

4. If all three statements succeed, then commit the work:

```
COMMIT;
```

However, if the **DELETE** statement fails, (for example, if the person has an account or an order record), and we haven't committed yet, then we can roll the work back:

```
ROLLBACK;
```

Note that if we had committed, then the **ROLLBACK** statement would have no effect.

Since the RDBMS must temporarily store enough information to either commit or roll back your entire transaction, it could run out of space if you do not commit often enough. A **DELETE** statement that deletes 100,000 rows from a table may cause the RDBMS to make temporary copies of all those rows. If the system does not have sufficient space for this, the delete will fail. You might have to delete those rows in smaller chunks, committing each time.

CONCURRENCY

- Oracle uses *locks* of various types to coordinate data manipulation, and many other activities, among concurrent sessions.

- When concurrent sessions access the same rows and perform DML statements — **INSERT**s, **UPDATE**s, and **DELETE**s — Oracle isolates each session's work.

  - For example, Oracle assures one session can't update or delete a row another session has already updated, until the other session does a **COMMIT** (or **ROLLBACK**).

- When a session begins manipulating data, it creates an *Exclusive Lock,* and associates all modified rows with that lock.

  - It also creates a second lock associated with the affected table, to prevent others from altering the table's definition until the transaction is done.

- Another session trying to manipulate any of the modified rows will find the lock.

  - By default, the second session will add (*enqueue*) itself to the list of sessions interested in rows controlled by the first session.

    - When the first session commits or rolls back, its lock clears, and the next session in the queue now controls the rows.

  - Your DML statement can include **NOWAIT**, telling Oracle to return an immediate error instead of enqueueing and waiting.

- A deadlock can occur if one transaction requires access to data locked by another in order to complete, while at the same time holding a lock that the other transaction is waiting for.

  - Oracle automatically detects this situation, rolling back the statement that created the deadlock.

Because in normal operation a session *enqueues* (adds itself to the wait list, or queue) on a lock held by another session, the locks themselves are often refered to as "enqueues."

A lock is a data structure created in shared memory (memory that all programs accessing the database can access). Oracle defines different categories and levels of lock, with many individual types of lock with different meanings, for various specific circumstances.

For example, when one session begins performing **INSERT**s, **UPDATE**s, and **DELETE**s, (that is, DML statements), it creates a lock structure representing its current transaction (set of statements that it will **COMMIT** or **ROLLBACK** later). It then marks each row it updates, in such a way that another session accessing that row will be directed to the lock. This lock (known as a *TX* lock) is in the category of DML locks, its level is Row-Level, and its specific type is an *Exclusive Lock*, indicating it has started a transaction and updated rows; other sessions wanting to update any of those same rows must wait for the first session to release the TX lock. However, other sessions can still read those rows (to perform **SELECT** statements). Only another session wanting to update or delete those rows will have to wait.

The session also creates another lock (called a *TM* lock) and associates it with the table itself. This one, a DML-category lock, is a *Table-Level* lock and its specific type is *Row Exclusive* (*RX*), indicating the session has exclusively locked rows in that table. Another session wanting to drop the table, or alter its structure, will have to wait for the first session to clear this lock.

The *Concepts* book, in the Oracle Online Documentation Library, describes concurrency issues, the different types of lock, and the operations that create them.

Locks are created and managed automatically when you perform DML statements. **COMMIT** or **ROLLBACK** ends your current transaction and releases its locks.

EXPLICIT LOCKING

- Exclusive locks are automatically placed whenever you execute **INSERT**, **UPDATE**, or **DELETE** statements.
  - Locks are released by a **COMMIT** or **ROLLBACK**.
- You can explicitly place a lock on an entire table.

```
LOCK TABLE table IN lock-mode MODE [NOWAIT|WAIT seconds];
```

  - **EXCLUSIVE** (or *Write*) *Lock* is a lock mode that does not allow any other user to place any lock on that row or table; other users can still read uncommitted data, however.
  - **SHARE** (or *Read*) *Lock* is a mode in which other users can also place share locks, but no user can obtain an exclusive lock.
- You can place a lock on a subset of rows through a **SELECT** statement.

```
SELECT col1, col2, col3, col4
  FROM table
 WHERE condition
   FOR UPDATE [OF table.col][NOWAIT|WAIT seconds];
```

- The **NOWAIT** option governs what happens if some or all of these records are already locked.
  - If **NOWAIT** is specified, control is returned to the user.
  - Otherwise, the statement will wait for the lock to be released before executing.

lock_person.sql

```
-- Lock records in the person table
SELECT id, lastname, firstname
  FROM person
 WHERE area_code = '303'
   FOR UPDATE;
```

update_person2.sql

```
-- All 303 area_code records are now locked
UPDATE person
   SET area_code = '720'
 WHERE area_code = '303';
```

lock_person2.sql

```
SELECT person.id, lastname, firstname
  FROM person JOIN employee ON person.id = employee.id
 WHERE area_code = '303'
   FOR UPDATE OF person.id;
```

update_person3.sql

```
-- Employee 303 area_code records in person are now locked
-- Corresponding rows in employee are not locked
UPDATE person
   SET area_code = '720'
 WHERE area_code = '303'
   AND id IN (SELECT id FROM employee);
```

lock_person3.sql

```
-- Lock the entire person table
LOCK TABLE person IN EXCLUSIVE MODE NOWAIT;
```

Note:

The option to **WAIT** for a specified number of seconds was added in version 11*g*.

DATA INCONSISTENCIES

- When multiple users try to access and manipulate the same data, several data inconsistencies can arise:
  - A *Dirty Read* occurs when a transaction reads uncommitted data by another transaction.
  - A *Non-Repeatable Read* occurs when a user reads rows, another user modifies or deletes the rows and commits, and the first user then tries to read the same rows, getting different data.
  - A *Phantom Read* occurs when a user runs a query, another user inserts rows and commits, then the first user reruns the query, finding the new rows.
- A read-only statement (that is, a **SELECT**) may:
  1. Ignore locks and read data that has been inserted or modified, but not yet committed (Read Uncommitted).
  2. Read only rows that are committed at the time of the read (Read Committed).
  3. Read any rows committed at the time of the read and, possibly, phantom rows added since the last read (Repeatable Read).
  4. The final isolation level (Serializable) is the same as Repeatable Read, except that a re-execution of your **SELECT** will not pick up phantom rows.
- Oracle only supports **READ COMMITTED** and **SERIALIZABLE** on a per-session basis; **READ COMMITTED** is the default.

```
SET TRANSACTION ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED};
```


		Possible Phenomena		
		Dirty Read	Non-Repeatable Read	Phantom Rows
Isolation Level	Read Uncommitted	Yes	Yes	Yes
	Read Committed	No	Yes	Yes
	Repeatable Read	No	No	Yes
	Serializable	No	No	No

These *isolation levels*, defined by SQL92, control which of the data inconsistencies will be prevented during the transaction. For example, in an application where it is highly unlikely that two transactions will attempt to update the same row at the same time, **READ COMMITTED** provides concurrency at the potential cost of data inconsistency. **SERIALIZABLE** transactions, on the other hand, ensure that data will be consistent, but may force another transaction to wait for data access.

Transaction inconsistencies can occur when multiple users are simultaneously updating the same data.

For instance, with Read Committed, if a **SELECT** is performed more than once in the same transaction, another user might have locked some of the rows, modified or deleted them, and then committed; thus, your **SELECT** might have different results at different times in your transaction.

Likewise, with Repeatable Read, if another transaction modifies or deletes rows and commits its changes, a re-execution of your **SELECT** will not see those changes. However, if another transaction adds rows, a repeat of your **SELECT** will see those phantom rows.

LOADING TABLES FROM EXTERNAL SOURCES

- Many systems provide statements or utilities that allow you to load bulk data from external files.

- Oracle's SQL*Loader utility loads rows into a table from external data files, including character and binary formats.

- Oracle provides the export (**exp**) and import (**imp)** utilities to copy or backup database objects and data.

```
exp scott/tiger file=mydata.dmp tables=employee,person,store
```

- Oracle 10*g* introduced Data Pump technology, which is much more efficient than Oracle's original export/import.

  - You invoke the Data Pump **expdp** and **impdp** commands much like the original **exp** and **imp**.

    - The files created by Data Pump are not compatible with files created by the original export/import utilities.

    - The original **exp** and **imp** support all 9*i* features and dataypes, while Data Pump supports 10*g* and higher features.

  - Data Pump requires additional system setup and configuration.

  - Data Pump places all dump and log files on the server's file system.

Oracle provides a standalone utility, SQL*Loader, for loading rows into a table from an external source. SQL*Loader can read a wide variety of data files, parsing records (according to your specifications) into values of the correct types for your table's columns. You create a control file that specifies how SQL*Loader will parse the datafile. You then run the **sqlldr** command from the operating system command line:

```
sqlldr / CONTROL=empdata.ctl LOG=empdata.log
```

. . . where *empdata.ctl* might contain:

```
LOAD DATA
INFILE 'empdata.txt'
BADFILE 'empdata.bad'
DISCARDFILE 'empdata.dsc'
APPEND INTO TABLE employee
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY X'27' TRAILING NULLCOLS
                      (id, store_number, pay_type_code,
                      pay_amount, title, supervisor_id );
```

LABS

❶ Insert your own name, address, and phone number into the **person** table.
(Solutions: *insert_person.sql, insert_person2.sql*)

❷ Make yourself an employee. Use the store furthest from your actual home. Your starting hourly wage is $8.25.
(Solution: *insert_employee.sql*)

❸ With a single statement, make the store manager of your store your supervisor.
(Solution: *update_emp.sql*)

❹ You must move to the city in which your store is located. Change your address to reflect your store's city, state, and zip; you don't yet know what your street address will be. Be sure to save your changes!
(Solution: *change_address.sql*)

❺ Congratulations — for doing all this, you get a raise to $10 an hour.
(Solution: *pay_increase.sql*)

❻ If our **product** table contains a product for which there is no vendor, we want to delete it.

a. Are there any such products?
b. If so, can you simply delete them from the **product** table?
c. Take all steps necessary to get rid of any such bogus products.
d. Make sure your changes are made permanent.

(Solutions: *product1.sql, product2.sql, product3.sql, product4.sql*)

❼ Delete all the inventory items for your store.
(Solution: *delete_inv.sql*)

❽ What — are you nuts!? Get those inventory items back!
(Solution: *rollback.sql*)

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(*size*)	Fixed-length character set data of *size* length. Default size is 1.	2000 bytes
NCHAR(*size*)	Fixed-length unicode-only datatype. National character set determines the max length of the column. Default size is 1.	2000 bytes
VARCHAR2(*size*)	Variable-length character string with max size of *size* bytes. Must specify size.	4000 bytes
NVARCHAR2(*size*)	Variable-length character string for national character set with max size of *size* 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(*p*,*s*)	Number with precision of *p* and scale of *s*. Must be integer if only *p* 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(*size*)	Binary data of *size* 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(*fractional_seconds*)	All the information contained in date, plus *fractional_seconds* – the number of digits in the fractional part of the **SECONDS** datetime field.	*fractional_seconds* may be 0-9, default is 6.
TIMESTAMP (*fractional_seconds*) WITH TIME ZONE	Same as above, with timezone displacement value.	0-9, default is 6.
TIMESTAMP (*fractional_seconds*) 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 (*year_precision*) TO MONTH	Period of time in years and months. *year_precision* is number of digits in YEAR datetime field.	0-9, default is 2.
INTERVAL DAY (*day_precision*) TO SECOND (*fractional_seconds*)	Period of time in days, hours, minutes, seconds. *day_precision* is max number of digits in DAY. *fractional_seconds* is max number of digits in SECONDS field.	*day_precision* 0-9, default is 2. *fractional_seconds* 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 9*i*, 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.

VIRTUAL COLUMNS

- Starting in Oracle 11*g*, you can add virtual columns to your database tables.

```
column_name [datatype] [GENERATED ALWAYS] AS (expression) [VIRTUAL]
```

  - A *virtual column* is defined as the result of an expression, which may involve functions and other columns.

  - If a datatype is not specified, Oracle will determine the datatype of the column based on the result of the expression.

    phone.sql

```
ALTER TABLE person
  ADD full_phone AS ( '(' || area_code || ')' ||
                      SUBSTR(phone_number,1,3) ||
                      '-' || SUBSTR(phone_number,4) );
```

- The column does not consume any space for data, as the values for the rows are always calculated when retrieved.

- You cannot set the value of a virtual column through an **INSERT** or **UPDATE**.

- The expression on which the virtual column is based must satisfy several rules.

  - It may only refer to other non-virtual columns within the same table, and only if they have already been defined.

  - It may only use *deterministic* functions, those that return the same value for a given parameter list each time.

    - For example, **SYSDATE()** is non-deterministic as it returns a different date/time value every time you invoke it.

- Benefits to using virtual columns include:

  - Multiple applications do not have to calculate this value.

  - Statistics can be gathered on the column's expression, making queries faster.

You can **SELECT** a virtual column just like any other column.

get_phone.sql
```
SELECT id, lastname, firstname, full_phone
  FROM person
 WHERE state = 'WA';
```

You can also use the virtual column in a **WHERE** clause, even using functions and pattern matching with it, just like any other column.

get_phone2.sql
```
SELECT id, lastname, firstname, full_phone
  FROM person
 WHERE full_phone LIKE '(303)%';
```

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 8*i*):

```
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 10*g*, dropped tables are noted 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 operation, called a *Create Table As Select* (CTAS), 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 will 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 22 - MAINTAINING PL/SQL CODE

OBJECTIVES

- Control and manage privileges related to stored programs.
- Query the Data Dictionary for information about PL/SQL subprograms.
- Insert directives into programs for conditional compilation of code.
- Change warning levels for more information during compile-time.
- Manage dependencies and validation of stored programs.
- Use native compilation to increase PL/SQL execution performance.

PRIVILEGES FOR STORED PROGRAMS

- You must grant **EXECUTE** privilege on a subprogram to allow another user to call it directly.

```
GRANT EXECUTE ON subprogram TO user;
```

- By granting **EXECUTE** privilege, you are allowing the other user to:
  - Directly call the subprogram.
  - Compile other subprograms that call your subprogram.
- In order to work with stored procedures, functions, or packages, you must be granted at least one of the following privileges:
  - **CREATE PROCEDURE** — Create subprograms in grantees schema.
  - **CREATE ANY PROCEDURE** — Create subprograms in any schema.
  - **ALTER ANY PROCEDURE** — Create subprograms in any schema.
  - **DROP ANY PROCEDURE** — Delete subprograms in any schema.
  - **EXECUTE ANY PROCEDURE** — Execute subprograms in any schema.
- Triggers have a similar set of privileges by replacing **PROCEDURE** with **TRIGGER**.
  - **CREATE TRIGGER** — Create triggers in grantees schema.
  - **CREATE ANY TRIGGER** — Create triggers in any schema.
  - **ALTER ANY TRIGGER** — Enable, disable, and compile triggers in any schema.
  - **DROP ANY TRIGGER** — Delete triggers in any schema.

DATA DICTIONARY

- Source and compiled code is kept for all of your subprograms and packages in data dictionary tables.
  - The **ALL_** versions show information about objects that you can access.
  - The **USER_** versions show information about objects that you own.
- Stored procedure, function, and package code is kept in the **ALL_SOURCE** table.

get_source.sql
```
SELECT text
  FROM user_source
 WHERE name = 'HR3'
 ORDER BY type, line;
```

- The **ALL_OBJECTS** table maintains information about your stored code.

object.sql
```
SELECT status
  FROM user_objects
 WHERE object_name = 'HR3';
```

  - The status column indicates whether the object is currently **VALID**.
    - **INVALID** objects are recompiled the next time they are used.
- Trigger source code is kept in **ALL_TRIGGERS**.
  - The trigger specification is parsed and kept in various fields, while the code for the trigger is in the **trigger_body** field.

trigger.sql
```
SELECT trigger_name, trigger_body
  FROM user_triggers;
```

- **ALL_PROCEDURES** contains a listing of all functions and procedures, including those contained within packages.
  - Attributes are given for each subprogram, such as whether it is an aggregate.

The following program will re-create a given trigger's **CREATE** statement.

gettrig.sql

```
SET HEADING OFF
SET NEWPAGE NONE
SET RECSEP OFF
SET VERIFY OFF
SET PAUSE OFF
SET LONG 4000

-- Prompt for the trigger name:
DEFINE trigger_name = &&trigger_name

-- Create an output file for the trigger source code:
SPOOL &trigger_name..sql

-- Trigger source code for &trigger_name
SELECT 'CREATE OR REPLACE TRIGGER ' || description
       || DECODE(when_clause,
                 NULL, '',
                 'WHEN (' || when_clause || ')'
                )
  FROM user_triggers
 WHERE trigger_name = UPPER('&trigger_name');
SELECT trigger_body
  FROM user_triggers
 WHERE trigger_name = UPPER('&trigger_name');
SELECT '/' FROM dual;

SPOOL OFF

PROMPT Trigger source code saved as &trigger_name..sql
UNDEFINE trigger_name


-- For reference...
-- user_triggers
--  Name                           Null?    Type
--   ------------------------------ -------- ----
--  TRIGGER_NAME                   NOT NULL VARCHAR2(30)
--  TRIGGER_TYPE                            VARCHAR2(16)
--  TRIGGERING_EVENT                        VARCHAR2(26)
--  TABLE_OWNER                    NOT NULL VARCHAR2(30)
--  TABLE_NAME                     NOT NULL VARCHAR2(30)
--  REFERENCING_NAMES                       VARCHAR2(87)
--  WHEN_CLAUSE                             VARCHAR2(4000)
--  STATUS                                  VARCHAR2(8)
--  DESCRIPTION                             VARCHAR2(4000)
--  TRIGGER_BODY                            LONG
```

PL/SQL Stored Program Compilation

- Prior to Oracle 9*i*, all PL/SQL subprograms were compiled into M Code.
  - Oracle stores your source code in the **USER_SOURCE** table.
  - Oracle also stores your M Code.
  - M Code is interpreted when you execute the subprogram.
  - Dependencies between objects are handled in the same manner as M Code.
- The advantage of M Code is its portability to various platforms.
  - The disadvantage of M Code is that it still must be interpreted to run on a specific platform.
- Oracle 9*i* Release 2 introduced the ability to natively compile your subprograms.
  - Oracle translates your PL/SQL code into corresponding C code and then uses a native compiler and linker to generate a shared library.
  - The shared libraries are then used whenever the subprogram is involved.
- Oracle 9*i* and 10*g* do not provide a native compiler; therefore, you must install a compiler compatible with the version of Oracle that you are using.
  - Once the compiler is installed into the Oracle environment, then native compilation is performed automatically.
- Starting with Oracle 11*g*, Oracle can compile PL/SQL directly into native code, without first converting it to C or using an external compiler.
  - Oracle stores the compiled native code in the SYSTEM tablespace, so external shared libraries aren't needed.

The advantage of using native compilation is improved performance for computation-intensive code. However, there are some potential disadvantages:

- You can't use debugging and tracing tools on natively compiled PL/SQL subprograms.
- Native compilation takes longer than normal compilation into M code.
- PL/SQL programs consisting mostly of SQL statements will probably not gain much performance.

CONDITIONAL COMPILATION

- Conditional compilation allows you to place directives within your PL/SQL code that are executed before your code is compiled.
  - Debugging code can be compiled into your procedures during development, and then removed from production code.
  - With the help of the **DBMS_DB_VERSION** package, you can insert version-specific code into your program.
- Selection directives **$IF**, **$THEN**, **$ELSE**, **$ELSIF**, and **$END** are used to test conditions.
  - Conditions tested must be able to be determined at pre-compile time and are called *boolean static expressions.*
- The **$ERROR** directive will cause a compile-time error to occur with the corresponding text returned.

```
$ERROR
    'Not done writing my procedure, so this cannot
      compile yet!'
$END
```

  - The block won't compile until the **$ERROR** is removed.
- Inquiry directives provide access to **PLSQL_CCFLAGS** options and are accessed with **$$*varname*** syntax.

```
ALTER SESSION SET PLSQL_CCFLAGS = 'debug:true';
...
$IF $$DEBUG $THEN
  DBMS_OUTPUT.PUT_LINE('In ' || $$PLSQL_UNIT || ':' ||
                          $$PLSQL_LINE);
$END
```

  - Setting **debug** to **false** will cause all PL/SQL code using the directive to re-evaluate and recompile.

Oracle 9*i* R2 has support for conditional compilation, but it is turned off by default. In Oracle 10*g* R1 (10.1.0.4), conditional compilation is turned on by default, but can be turned off. Beginning with Oracle 10*g* R2, conditional compilation is always turned on.

The following program demonstrates one of the popular uses of conditional compilation, for debugging purposes.

debug.sql

```
CREATE OR REPLACE FUNCTION get_store_state
   (st_num store.store_number%TYPE) RETURN CHAR
AS
   st store.state%TYPE;
BEGIN
   $IF $$DEBUG $THEN
      DBMS_OUTPUT.PUT_LINE('In ' || $$PLSQL_UNIT || ':' || $$PLSQL_LINE);
   $END
   SELECT state
     INTO st
     FROM store
    WHERE store_number = st_num;
   RETURN st;
END;
/

ALTER SESSION SET PLSQL_CCFLAGS = 'debug:true';
ALTER FUNCTION get_store_state COMPILE;

DECLARE
   st VARCHAR2(30);
BEGIN
   st := get_store_state(7);
END;
/

ALTER SESSION SET PLSQL_CCFLAGS = 'debug:false';
ALTER FUNCTION get_store_state COMPILE;

DECLARE
   st VARCHAR2(30);
BEGIN
   st := get_store_state(7);
END;
/
```

COMPILE-TIME WARNINGS

- Oracle Database 10*g* introduced compile-time warnings to let you know if code might have a performance flaw or possible runtime error.

- There are three types of warnings:

  - **SEVERE** — Alerts for conditions that might produce wrong results.
  - **PERFORMANCE** — Alerts when a statement might affect code performance.
  - **INFORMATIONAL** — Neither severe nor a performance warning, such as code that may never be executed.

- You may enable and disable **PLSQL_WARNINGS** at the database, session, or object level:

```
ALTER SYSTEM SET PLSQL_WARNINGS = 'ENABLE:ALL';
                                  --enable all three types
ALTER SESSION SET PLSQL_WARNINGS =
   'ENABLE:SEVERE', 'ENABLE:PERFORMANCE';
ALTER PROCEDURE add_stock COMPILE PLSQL_WARNINGS =
   'ENABLE:ALL';
```

  - Oracle disables warnings by default.

  - You cannot turn on warnings for anonymous blocks.

- To display warnings, use the **SHOW ERRORS** SQL*Plus command or query the **USER_ERRORS** Data Dictionary view.

- Query the **USER_PLSQL_OBJECT_SETTINGS** Data Dictionary view to find the warning settings for the objects you own:

```
SELECT name, plsql_warnings
 FROM user_plsql_object_settings;
```


Use the **DBMS_WARNING** package to change warning settings programmatically from a PL/SQL block or other language (such as Java, C, etc...).

```
DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:ALL'   ,'SESSION');
DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:PERFORMANCE'   ,'SYSTEM');
```

add_stock_warn.sql

```
CREATE OR REPLACE PROCEDURE add_stock (
     snum inventory.store_number%TYPE,
     pid  inventory.product_id%TYPE,
     new_stock inventory.quantity_on_hand%TYPE )
AS
   qtyoh inventory.quantity_on_hand%TYPE := 0;
   no_parent_record EXCEPTION;
   PRAGMA EXCEPTION_INIT(no_parent_record, -2291);
BEGIN

  BEGIN
    SELECT quantity_on_hand INTO qtyoh
      FROM inventory
     WHERE store_number = snum AND product_id = pid;
  EXCEPTION
    WHEN no_data_found THEN
      BEGIN
         INSERT INTO inventory (store_number, product_id)
                VALUES (snum, pid);
      EXCEPTION
        WHEN no_parent_record THEN
            raise_application_error(-20005,
              'Invalid store or product ID.', false);
      END;
  END;

  IF (FALSE) THEN   -- generates warning for code that will never run
    qtyoh := qtyoh + new_stock;
  END IF;

  UPDATE inventory SET quantity_on_hand = qtyoh
   WHERE store_number = snum AND product_id = pid;

END;
/
ALTER PROCEDURE add_stock COMPILE PLSQL_WARNINGS = 'ENABLE:ALL'
/
SHOW ERRORS
```

The PL/SQL Execution Environment

- When you execute a subprogram, Oracle allocates a shared area to hold the compiled code and a private area for session values.
  - The program is stored in the Library cache inside of the System Global Area (SGA).
  - The User Global Area (UGA) holds the local, global, and package variable values for the session.
- The SGA is used by an Oracle instance to hold shared memory containing data and control information.
  - Users share the data in the SGA and thus it is sometimes referred to as the "Shared Global Area."
  - The SGA is divided into different memory structures, including database buffers, redo logs, the Java pool and the shared pool.
  - No user data is stored here.
- The UGA contains data required to support a session, such as its variable values.

DEPENDENCIES AND VALIDATION

- Some Oracle database objects will be inherently dependent upon other database objects.

  get_dep.sql
  ```
  SELECT *
    FROM user_dependencies
  WHERE name = 'HR3';
  ```

  - A view is dependent on the structure of the underlying base tables.
  - A stored procedure or function is dependent on the tables that it works with and the other subprograms that it invokes.

- Modifying one database object could cause several other database objects to no longer run properly.
  - Modifying an object involves DDL (**CREATE**, **ALTER**, **DROP**), not DML (**INSERT**, **UPDATE**, **DELETE**) statements.
  - Any time an object is modified, all dependent objects are marked **INVALID**.
    - Object validation is kept in various Data Dictionary tables for different objects.

      object.sql
      ```
      SELECT status
        FROM user_objects
       WHERE object_name = 'HR3';
      ```

- Invalid objects will be automatically recompiled the next time they are accessed.
  - You can recompile all objects, invalid or not, belonging to a given schema using the **DBMS_UTILITY** package.

    ```
    DBMS_UTILITY.COMPILE_SCHEMA (schema => 'SCOTT');
    ```

  - You can recompile all invalid objects at once with the *utlrp.sql* script provided by Oracle.
    - This script is located in *$ORACLE_HOME/rdbms/admin* and must be run by a user with **SYSDBA** privilege.

To see invalidation in action, run the following scripts:

modify_table.sql

```
-- make title field longer
ALTER TABLE EMPLOYEE
  MODIFY title VARCHAR2(20)
```

get_invalid.sql

```
SELECT *
  FROM user_objects
WHERE object_name = 'HR3';
```

run_proc.sql

```
DECLARE
  mycurs SYS_REFCURSOR;
BEGIN
  OPEN mycurs FOR
    SELECT * FROM employee WHERE title = 'Manager';
  hr3.give_raises(mycurs);
  CLOSE mycurs;
END;
/
```

After running the last script, run *get_invalid.sql* again to see that the **HR3** package is once again **VALID**.

MAINTAINING STORED PROGRAMS

- Use **ALTER PROCEDURE**, **ALTER TRIGGER**, or **ALTER FUNCTION** to recompile a stored subprogram from the source code stored in the system catalog:

```
ALTER PROCEDURE procedurename COMPILE;
```

- Several compiler options can be specified after the **COMPILE** keyword.
  - **PLSQL_WARNINGS** is used to set the warning level.
  - **PLSQL_CCFLAGS** is used to set compiler flags.
  - **PLSQL_OPTIMIZE_LEVEL** is used to set the optimization level.
  - **PLSQL_CODE_TYPE** is set to **INTERPRETED** or **NATIVE**.
  - **PLSQL_DEBUG** is set to **true** to indicate that code should be compiled in interpreted mode for debugging purposes, regardless of the **PLSQL_CODE_TYPE** value.
  - **NLS_LENGTH_SEMANTICS** is used to switch **CHAR** and **VARCHAR2** semantics from **BYTE** to **CHAR**, making them synonymous with **NCHAR** and **NVARCHAR2**.
  - All of these options can be queried for conditional compilation.
- You can also use the **ALTER TRIGGER** command to disable, enable, or rename a trigger.

```
ALTER TRIGGER triggername DISABLE;
```

- Use **DROP PROCEDURE**, **DROP TRIGGER**, or **DROP FUNCTION** to drop a stored subprogram from the schema.

```
DROP PROCEDURE procedurename;
```

- Any **ALTER** or **DROP** will cause all dependent objects to be marked invalid.

LABS

❶ Demonstrate how subprograms can become invalid by doing the following:

Create a copy of the **pay_type** table called **pay_type_copy**:

```
CREATE TABLE pay_type_copy as
SELECT *
  FROM pay_type;
```

Create a function called **pay_type_count()** that returns the number of pay types in the **pay_type_copy** table. Create an anonymous block of code to test your **pay_type_count()** function.
(Solution: *valid.sql*)

❷ Now drop the **pay_type_copy** table and verify that your **pay_type_count()** function has been marked invalid in the Data Dictionary.
(Solution: *invalid.sql*)

❸ Try to run the **pay_type_count()** function and view the errors.
(Solution: *run_invalid.sql*)

