Oracle 10g
SQL Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

ORACLE 10G SQL PROGRAMMING

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

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

Editor: Jan Waleri

Editorial Assistants: Dana Howell and 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 indexes to improve query performance.
- Use SQL*Plus to create formatted reports.
- Apply the basic theory behind relational database design.
- Contribute to all phases of database design and development.

- Use all aspects of subqueries.
- Apply Oracle's features for querying hierarchical data models.
- Use Oracle's Object-Relational Model.
- Create object types.
- Use Oracle's collection types in SQL.
- Select appropriate date-related datatypes for your applications.
- Use Oracle's regular expression SQL functions to perform pattern matching and string manipulation.
- Create and manage temporary tables.
- Establish goals in SQL tuning to improve performance.
- Use Oracle Database 10*g*'s tuning tools.
- Describe how indexes are used in RDBMSs, and use them effectively.
- Use the various analytic functions provided by Oracle to perform sophisticated analysis.
- Use SQL*Plus to format reports and extract data.

COURSE OVERVIEW

- **Audience:** This course is designed for database application developers.
- **Prerequisites:** Familiarity with relational database concepts.
- **Student Materials:**
  - Student workbook
- **Classroom Environment:**
  - One workstation per student
  - Oracle10*g*

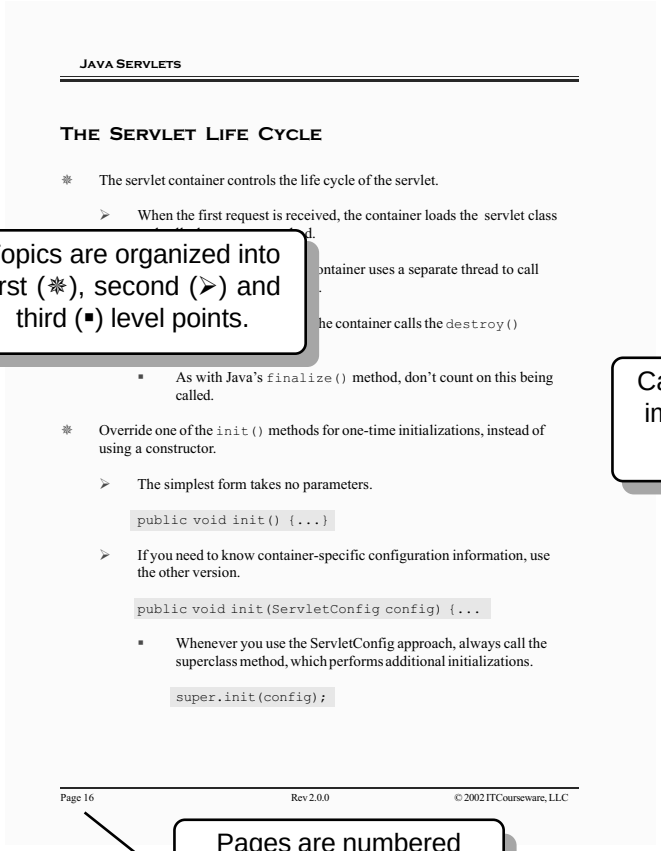
USING THE WORKBOOK

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

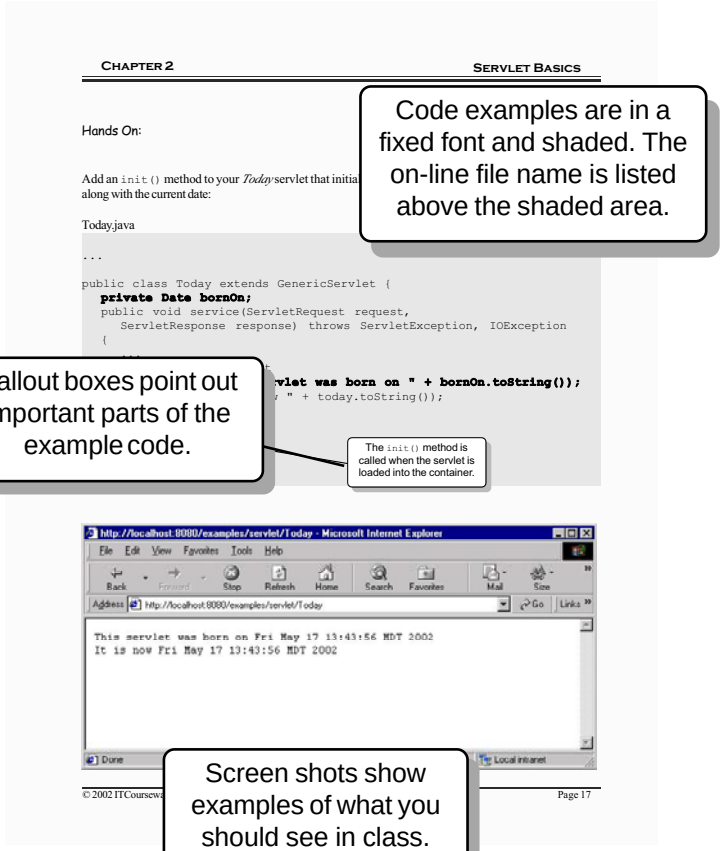
In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.



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



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.

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

SUGGESTED REFERENCES

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

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

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

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

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

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

An easy way to find it is to go to:

http://tahiti.oracle.com/

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

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

CHAPTER 2 - RELATIONAL DATABASE AND SQL OVERVIEW

OBJECTIVES

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

REVIEW OF RELATIONAL DATABASE TERMINOLOGY

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

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

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

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

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

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

RELATIONAL DATABASE MANAGEMENT SYSTEMS

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

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

dictionary.sql

```
COL table_name FORMAT a40
COL comments FORMAT a50 wrapped
SET linesize 100

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.

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.
Oracle3	1983	Rollback and commit transactions supported. UNIX operating system supported.
Oracle4	1984	Read consistency.
Oracle5	1985	Client-Server model. Supported networking, distributed queries.
Oracle6	1988	Oracle Financials released. PL/SQL, hot backups, row-level locking.
Oracle7	1992	Triggers, integrity constraints, stored procedures.
Oracle8	1997	Object-oriented development, multi-media applications.
Oracle8*i*	1999	Support for the internet. Contained native Java Virtual Machine.
Oracle9*i*	2001	Over 400 new features — flashback query, multiple block sizes, etc. Ability to manipulate XML documents.
Oracle10*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.

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. Oracle10*g* adheres to SQL:1999 standards.

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

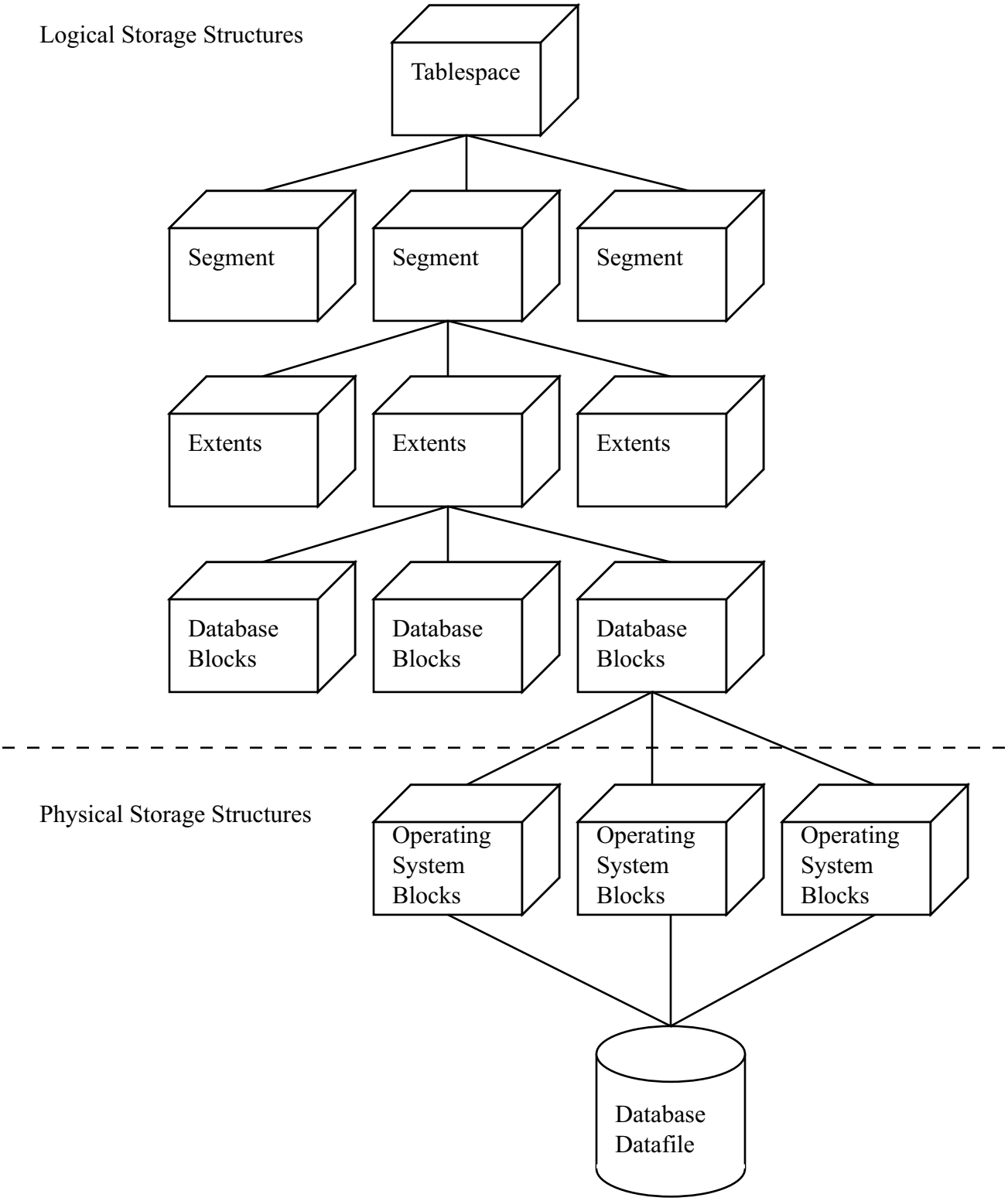
LOGICAL AND PHYSICAL STORAGE STRUCTURES

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 is 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).

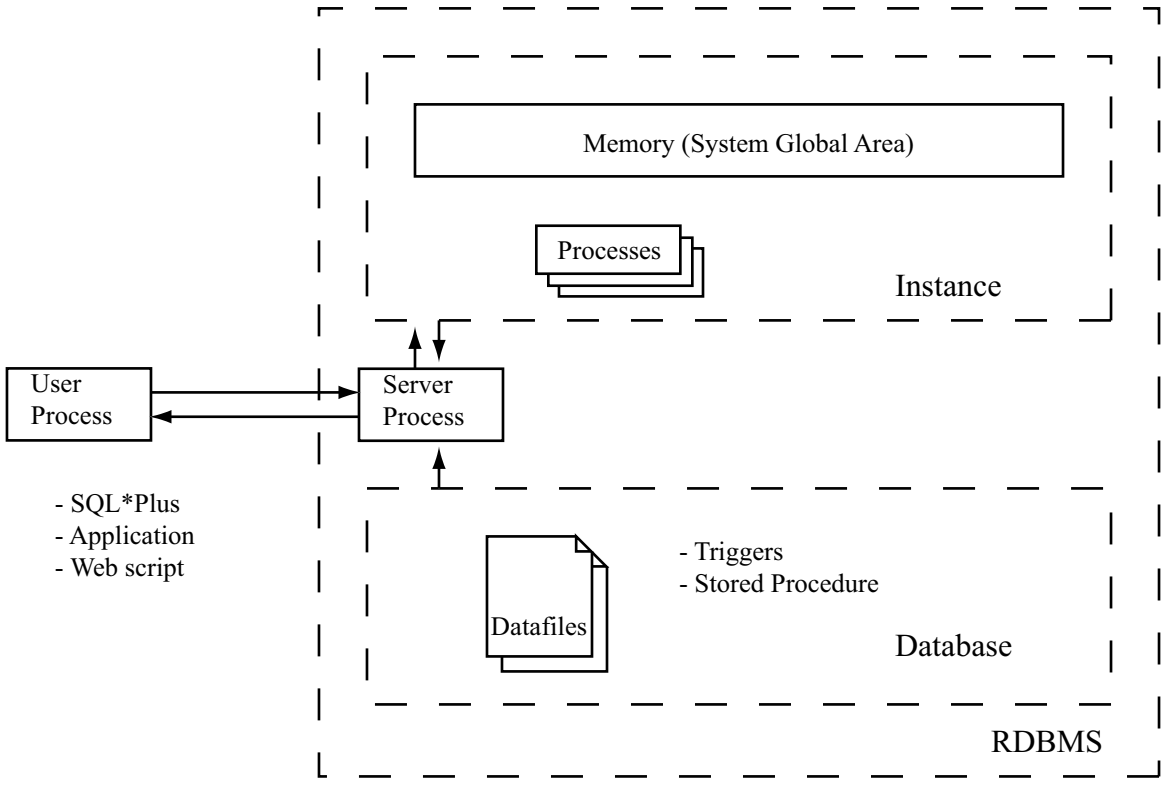
How can you execute SQL statements?

Interactive SQL: All database vendors provide some sort of program you can run that allows you to connect to the database and then type in arbitrary SQL statements, displaying the results on your screen. The appearance and usage of these programs varies widely from vendor to vendor, but the idea is the same: a direct, interactive SQL session connected to your database. The SQL language itself is standard and usually consistent among different vendors.

Embedded SQL: Alternatively, you can place SQL statements in a program written in a programming language like C or Java. You must provide statements in the program that will connect your program to the database, create storage within the program for any data you retrieve from the database, check for errors, disconnect, and so on. The SQL statements are defined when you write the program, and compiled into the executable. The syntax and requirements for embedding SQL statements in a program are also standard and consistent among different vendors (and different programming languages).

Dynamic SQL: With a little more work, you can write a program that will generate SQL statements at runtime, perhaps based on user input.

Database Servers and Clients



DATATYPES

- Each column in a table has a specific, pre-defined 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.

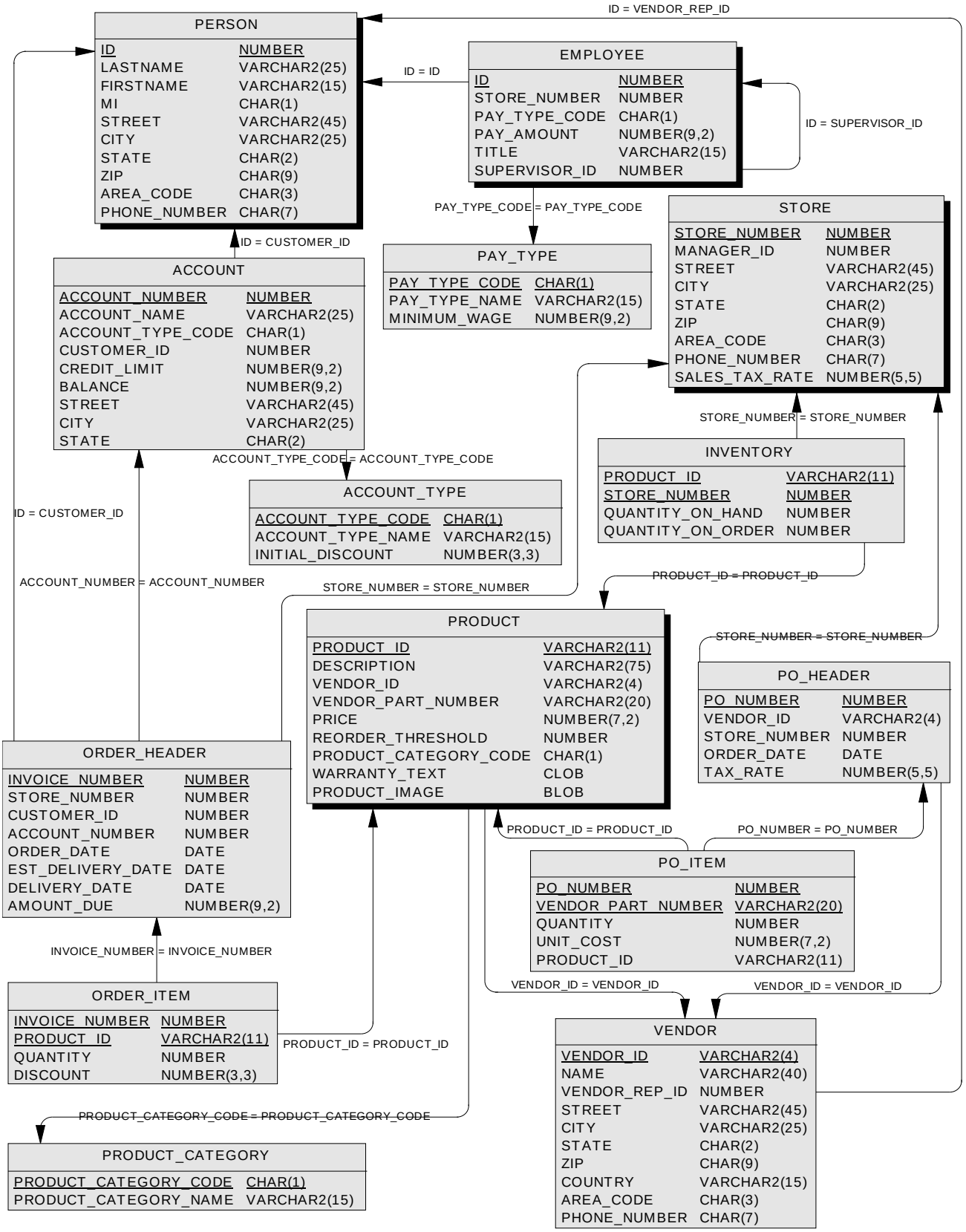
When a credit account is used for an order, the balance for the account must be updated to reflect the total amount of the invoice (including tax). The salesperson verifies (with a phone call) that the person is a valid user of the account.

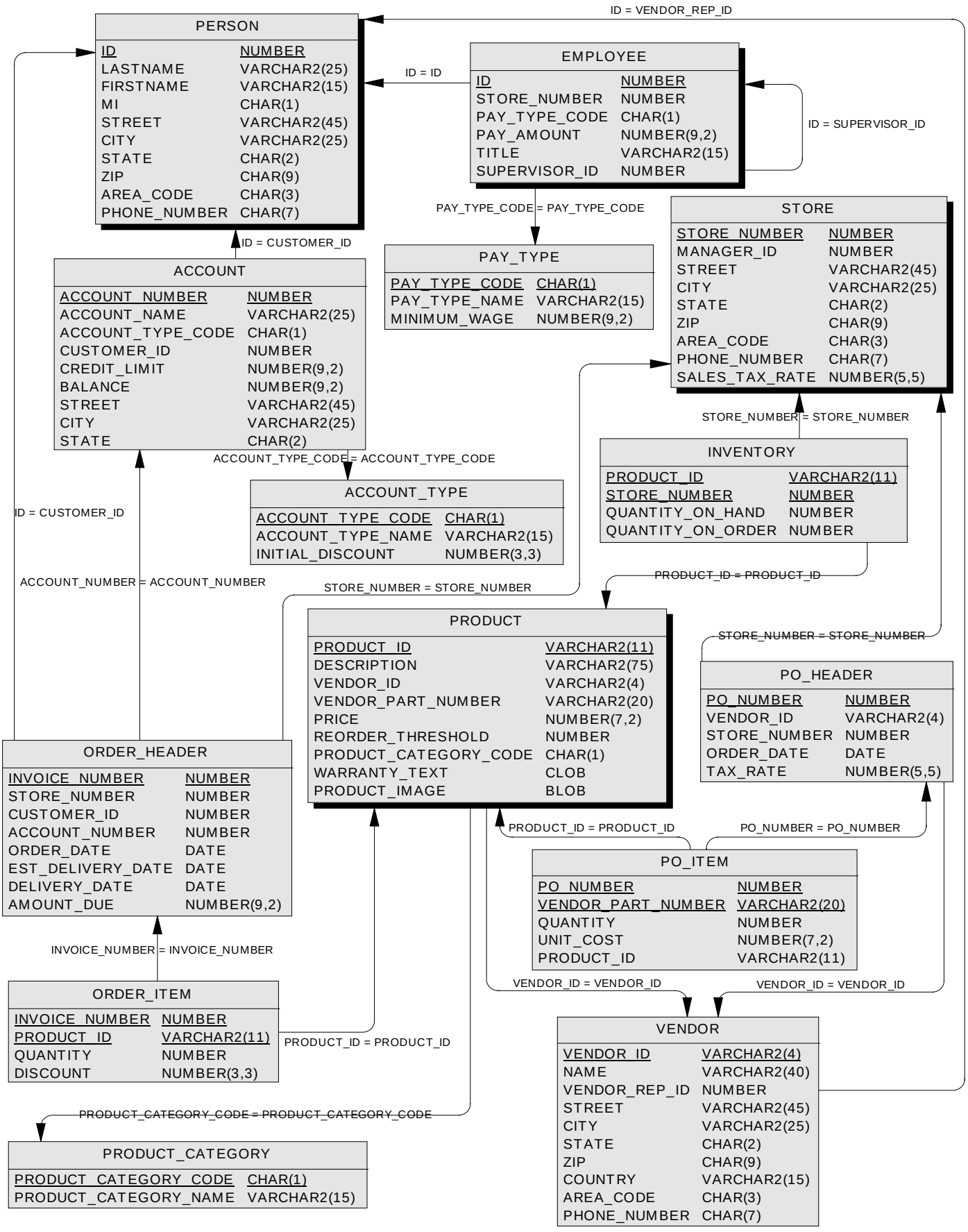
Each product we sell has a product ID and description. In addition, we keep track of the vendor from whom we purchase that product, the ID for that product, and the category ("Software," "Peripheral," or "Service"). We also store the address, phone, and sales rep ID for each individual vendor.

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
PRODUCT_ID = PRODUCT_ID
STORE_NUMBER = STORE_NUMBER
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 13 - HIERARCHICAL QUERIES

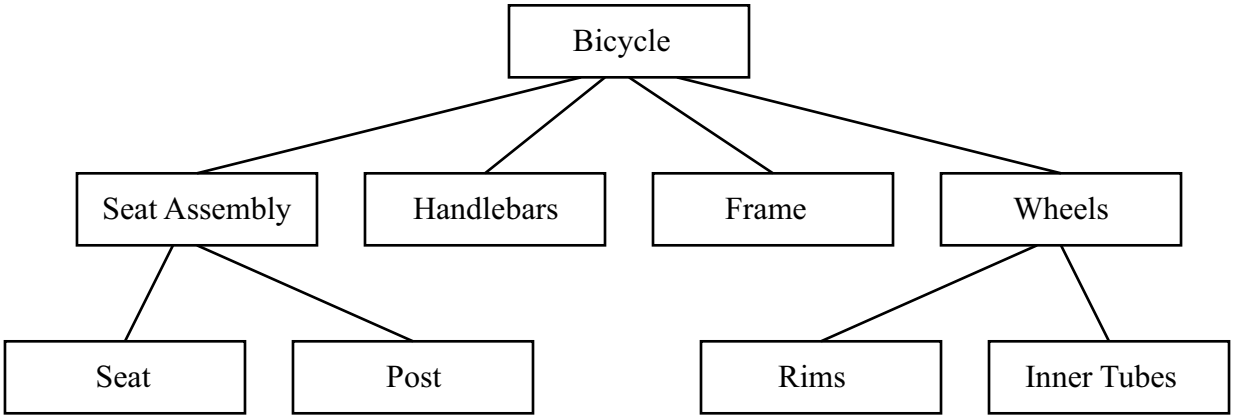
OBJECTIVES

- Establish hierarchical data models.
- Use SQL to create effective queries over hierarchical data.
- Use **SYS_CONNECT_BY_PATH** to produce useful hierarchical result sets.

HIERARCHICAL DATA

- Any data that can be classified on different levels is called *hierarchical data*.
  - The values on a level depend on only a single value on the previous level.
- Examples of hierarchical data include:
  - A family tree database — A father can have a son and a daughter; the father has a parent of his own.
  - An organization chart — The President of the company is at the top of the tree; the corporation's Vice Presidents are on the next level, etc.
  - Manufacturing — A product is composed of many parts; each part can be made up of other parts.

The easiest way to understand hierarchical data is to see an example.



In the example above, we can see hierarchical data depicting the parts to create a bicycle. The bicycle is composed of a seat assembly, wheels, handlebars, and a frame. The seat assembly is composed of two other parts, as are the wheels. The handlebars and the frame have no components.

Notice that each level is defined by the level above it. The rims are components of the wheels. The seat assembly does not share components with anything else on its level.

bicycle.sql

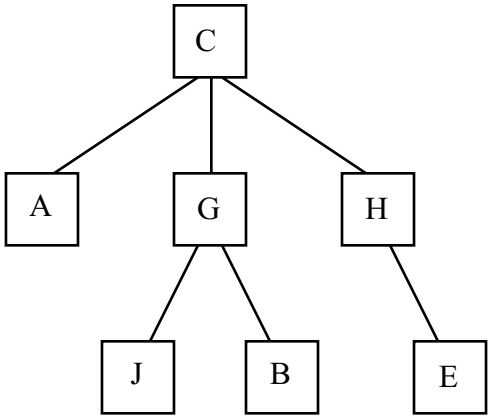
```
CREATE TABLE bicycle_parts (
   parent_part VARCHAR2(30),
   child_part VARCHAR2(30)
);
```

One common example used when explaining hierarchical data uses a family. Keep in mind that in an Oracle hierarchical query, you can only follow one parent hierarchy (either father or mother) at a time.

HIERARCHICAL TERMINOLOGY

- Each element in the hierarchy is a *node*.
- The top-most level of the hierarchy can have only one node, the *root* of the hierarchy.
- The root may have multiple paths, called *branches*, to the next level.
- A node on the next level can be reached by only one branch.
  - The node on the higher level is its *parent*.
  - The node on the lower level is called the *child*.
  - Only the root node has no parent.
- Any node without children is a *leaf node*.

Let's look at a sample hierarchy populated with simple values.



The node with value **C** is the *root node* of the hierarchy. The root node has three *children*, **A**, **G**, and **H**. Node **G** has two children, nodes **J** and **B**. Node **E** has one *parent*, node **H**. Nodes **A**, **J**, **B**, and **E** are *leaf nodes*.

HIERARCHICAL QUERY

- To create a hierarchical query, you will need two clauses in your SQL statement: **START WITH** and **CONNECT BY**.

```
    SELECT column_list
      FROM table_name
START WITH condition
CONNECT BY condition;
```

- The **START WITH** clause specifies the root of the tree for this query.

- The **CONNECT BY** clause defines the relationship between the parent and the child.

  - The condition in the **CONNECT BY** clause contains the **PRIOR** keyword.

```
column_name = PRIOR column_name
```

```
PRIOR column_name = column_name
```

  - Place the **PRIOR** keyword with the child column.

Continuing with our bicycle example, we can see a table of bicycle parts.

bicycle_parts.sql

```
SELECT * FROM bicycle_parts;
```

```
PARENT_PART          CHILD_PART
-------------------- --------------------
                     BICYCLE
BICYCLE              SEAT ASSEMBLY
BICYCLE              WHEELS
BICYCLE              HANDLEBARS
BICYCLE              FRAME
BICYCLE              BRAKES
SEAT ASSEMBLY        SEAT
SEAT ASSEMBLY        POST
WHEELS               RIMS
WHEELS               INNER TUBES
BRAKES               PADS
BRAKES               CALIPER
BRAKES               LEVERS
BRAKES               CABLES
```

From this table, we can see that one child part has no parent, so this should be the root. We can use the **LPAD** function to help visualize the hierarchy (this is a commonly-used technique with hierarchical queries):

lpad_bicycle_parts.sql

```
SELECT LPAD(' ', 3*(LEVEL-1)) || child_part AS bicycle
  FROM bicycle_parts START WITH child_part = 'BICYCLE'
CONNECT BY parent_part = PRIOR child_part;
```

```
BICYCLE
------------------------------------------------------------
BICYCLE
   SEAT ASSEMBLY
      SEAT
      POST
   WHEELS
      RIMS
      INNER TUBES
   HANDLEBARS
   FRAME
   BRAKES
      PADS
      CALIPER
      LEVERS
      CABLES
```

HIERARCHICAL PSEUDOCOLUMNS

- The **LEVEL** pseudocolumn returns the tree level of the hierarchical query.
  - The root node is on level 1, the children of the root are on level 2, and so on.
  - The **LEVEL** column is most commonly used with the **LPAD** function to format the results so that they resemble a tree turned on its side.
  - To add *n* spaces per level, use the following format:

```
LPAD(' ', n*(LEVEL-1)) || column_name
```

- The **CONNECT_BY_ISCYCLE** pseudocolumn returns a **1** if the current row has a child which is also its ancestor, otherwise it returns **0**.
  - Use this to detect circular references within your hierarchy.
- The **CONNECT_BY_ISLEAF** pseudocolumn returns a **1** if the current row is a leaf node, or else it returns **0**.

We have already seen how we can use the **LEVEL** psuedocolumn along with the **LPAD** function to format our tree output. We can also query the level in our hierarchical query, as shown below:

level.sql

```
SELECT child_part, LEVEL
  FROM bicycle_parts START WITH child_part = 'BICYCLE'
CONNECT BY parent_part = PRIOR child_part;
```

```
CHILD_PART                LEVEL
------------------------  ----------
BICYCLE                        1
SEAT ASSEMBLY                  2
SEAT                           3
POST                           3
WHEELS                         2
RIMS                           3
INNER TUBES                    3
HANDLEBARS                     2
FRAME                          2
BRAKES                         2
PADS                           3
CALIPER                        3
LEVERS                         3
CABLES                         3
```

We can easily retrieve just the leaf nodes, using the **CONNECT_BY_ISLEAF** pseudocolumn in our **WHERE** clause.

leaf.sql

```
SELECT child_part
  FROM bicycle_parts WHERE CONNECT_BY_ISLEAF = 1
 START WITH child_part = 'BICYCLE'
CONNECT BY parent_part = PRIOR child_part;
```

```
CHILD_PART
--------------------
SEAT
POST
RIMS
INNER TUBES
HANDLEBARS
FRAME
PADS
CALIPER
LEVERS
```

SYS_CONNECT_BY_PATH

- **SYS_CONNECT_BY_PATH** is a special function which returns the path from the root to the given node, separated by a character of your choice.

```
SYS_CONNECT_BY_PATH(column_name, character)
```

- The column in the function can be of the **CHAR**, **VARCHAR2**, **NCHAR**, or **NVARCHAR2** datatype.

- The output from this function is the **VARCHAR2** datatype.

Returning to our bicycle example, we can use the **SYS_CONNECT_BY_PATH** function to show the path from the root to the given node.

path.sql

```
SELECT LPAD(' ', 3*(LEVEL-1)) ||
    SYS_CONNECT_BY_PATH(child_part, '/') AS path
  FROM bicycle_parts START WITH child_part = 'BICYCLE'
CONNECT BY parent_part = PRIOR child_part;
```

```
PATH
------------------------------------------------------------
/BICYCLE
    /BICYCLE/SEAT ASSEMBLY
       /BICYCLE/SEAT ASSEMBLY/SEAT
       /BICYCLE/SEAT ASSEMBLY/POST
    /BICYCLE/WHEELS
       /BICYCLE/WHEELS/RIMS
       /BICYCLE/WHEELS/INNER TUBES
    /BICYCLE/HANDLEBARS
    /BICYCLE/FRAME
    /BICYCLE/BRAKES
       /BICYCLE/BRAKES/PADS
       /BICYCLE/BRAKES/CALIPER
       /BICYCLE/BRAKES/LEVERS
       /BICYCLE/BRAKES/CABLES
```

A common path separator character is the forward slash (/), but you can use any character you desire. Note that the first row contains only one value in the path, because it is the root node.

Processing Hierarchical Queries

- Oracle processes a hierarchical query in the following steps:

    1. Process joins, if present.

    2. From the join results, select the root row using the **START WITH** clause.

    3. Determine the child rows of the root row by evaluating the **CONNECT BY** clause.

    4. Determine the child rows in the third level, and continue the process until the tree is created.

    5. Evaluate any remaining **WHERE** clause expressions to further trim the result set.

LABS

❶ Oracle's HR sample schema includes an **EMPLOYEES** table. Each employee has an **EMPLOYEE_ID** and a **MANAGER_ID**, which is the **EMPLOYEE_ID** of that employee's manager (see the Sample Schemas in the Oracle Database online documentation for the rationale and diagrams of the sample schemas). Write a query, using **START WITH** and **CONNECT BY**, to list each each employee's last name and ID, along with their manager's ID, in hierarchical sequence starting with the employee with no manager ID.
(Solution: *startwith.sql*)

❷ Modify the previous query. Use **LPAD** to add spaces to the left of the employee's name so that each is indented according to the employee's level in the management hierarchy.
(Solution: *indenthier.sql*)

❸ Write a normal query (that is, without **START WITH** and **CONNECT BY**), using a self-join, to list the last name of each employee, along with the last name of each employees manager. Show the manager name first, the employee name second, with an arrow (' <-- ') concatenated between them indicating the employee reports to the manager.
(Solution: *selfjoin1.sql*)

❹ Modify the previous self-join query. Use **START WITH** and **CONNECT BY** to list the employees in hierarchical order.
(Solution: *selfjoin2.sql*)

❺ Modify the above query to add indentation showing the levels of the hierarchy.
(Solution: *selfjoin3.sql*)

❻ Write a query that uses **SYS_CONNECT_BY_PATH** to list, for each employee, their full management hierarchy (including themselves), separated by slashes.

a. Try changing the separator string to an arrow (as in the previous queries) — how does it look?
b. Use **LPAD** to add level indentation to the query.

(Solution: *connect_by_path.sql*)

❼ Modify the previous query to list the management hierarchy only for those employees who are not managers.
(Solution: *isleaf.sql*)

CHAPTER 20 - ORACLE ANALYTIC FUNCTIONS

OBJECTIVES

- Explain and use the analytic functions provided by Oracle.
- Use grouping sets to determine column summary rows.
- Use the **OVER**, **PARTITION**, and **ORDER BY** clauses in analysis.
- Apply various windowing techniques.
- Use windowing to produce sliding summaries.
- Show the ranking of results within a result set partition.
- Use the **ROLLUP** and **CUBE** clauses to generate subtotals and totals.
- Define rules for data models and apply them to query results with the **MODEL** clause.

ANALYTIC FUNCTIONS

- SQL's predicates and functions provide for very specific selection of rows into a result set, and manipulation of individual data values.
  - Aggregate functions (**SUM**, **MAX**, **COUNT**, etc.) can perform basic summary analysis over this result set.
  - Custom applications or third-party software often perform further analysis of the results.
  - You could perform some of this analysis using inline views, self joins and standard functions with the added cost in performance due to multiple passes through the data.
- Oracle's *analytic functions* compute aggregate values over a group of rows.
  - You can use these analytic functions to make calculations for sliding window calculations, lag analysis, lead analysis, first/last analysis, linear regressions, rankings, and percentiles.
  - These functions differ from the standard aggregate functions in that each group of records may produce multiple summary rows.
- You must define a *window* to establish the group of records over which the analytic function will perform its calculation.
  - The **OVER** clause, following the function name, defines the window.
    - The analytic function calculates a resulting row for each window of rows.
    - The window can be determined by a physical number of rows or a logical interval based on a value, such as time.

The analytic functions are performed after any joins, **WHERE**, **GROUP BY**, or **HAVING** clauses so the windows of data can be established. The only clause performed after the analytic function is the **ORDER BY** clause, so it is almost the last operation of the query. You can only use analytic functions in the **SELECT** and **ORDER BY** clauses.

The following is a list of analytic functions supported by Oracle:

AVG — Calculates average value of an expression.
CORR — Calculates the coefficient of correlation of number pairs.
COVAR_POP — Calculates the population covariance of number pairs.
COVAR_SAMP — Calculates the sample covariance of number pairs.
COUNT — Calculates the number of rows returned by the query.
CUME_DIST — Returns the cumulative distribution of a value in a group of values.
DENSE_RANK — Returns the numeric rank of a row in an ordered group of rows.
FIRST_VALUE — Returns the first value in an ordered set of values.
LAG — Provides access to a row at a given physical offset prior to a particular position.
LAST_VALUE — Returns the last value in an ordered set of values.
LEAD — Provides access to a row at a given physical offset beyond a particular position.
MAX — Finds the maximum value in an expression.
MIN — Finds the minimum value in an expression.
NTILE — Divides an ordered data set into a number of buckets.
PERCENT_RANK — Same as **CUME_DIST**, except range of values is between **0** and **1**.
PERCENTILE_CONT — Inverse distribution that assumes a continuous distribution model.
PERCENTILE_DISC — Inverse distribution that assumes a discrete distribution model.
RANK — Returns the rank of a value in a group of values.
RATIO_TO_REPORT — Returns the ratio of a value over the a set of values.
REGR_ ***Functions*** (Linear Regression) — Ordinary least squares regression line.
ROW_NUMBER — Creates a unique number for each row.
STDDEV — Calculates standard deviations.
STDDEV_POP — Population standard deviation.
STDDEV_SAMP — Cumulative sample standard deviation.
SUM — Calculates the sum of values of an expression.
VAR_POP — Returns the sum of values of expression.
VAR_SAMP — Calculates the sample variance of a set of numbers.
VARIANCE — Calculates the variance of expression.

OVER, PARTITION BY, AND ORDER BY

- To perform analytic functions across subgroups of records inside the result set you must establish partition boundaries within the result set.

- A **PARTITION BY** clause after the **OVER** keyword defines the partition boundaries.

  - The **PARTITION BY** keyword can include any non-analytic expression.

  - The analytic functions are applied to each group independently and then reset for the next group.

  - If the **PARTITION BY** clause is omitted from the **OVER** clause, then the entire result set is considered to be a single partition.

  - Use the **ORDER BY** clause to specify how data is sorted within each group.

    - The ordering is important, since functions are applied to the current row and the preceding rows within the partition.

The following example will list all of the employees, and create running totals for each department partition and for all of the records in the **employees** table.

sum_of_salaries.sql

```
  SELECT last_name, department_id , salary,
         SUM(salary) OVER (PARTITION BY department_id
                                ORDER BY last_name)
                     AS "Dept Salary Total",
         SUM(salary) OVER (ORDER BY department_id, last_name)
                     AS "Company Salary Total",
         ROW_NUMBER() OVER (PARTITION BY department_id
                                  ORDER BY last_name)
                      AS "Seq"
    FROM hr.employees
ORDER BY department_id, last_name;
```

The following example will demonstrate various ways of using **PARTITION BY** and **ORDER BY** in analytic queries. The first three columns just list the category, id, and list price of products, respectively. The fourth column will keep a running total of the product list prices with a particular category id. The fifth column will display the total for the list prices in a particular category. The last column will keep a running total of the categories within the query, since there is no partition.

partitions.sql

```
SELECT category_id, product_id, list_price,
       SUM(list_price) OVER (PARTITION BY category_id
                                  ORDER BY product_id) AS  "sum 1",
       SUM(list_price) OVER (PARTITION BY category_id
                                  ORDER BY category_id) AS "sum 2",
       SUM(list_price) OVER (ORDER BY category_id) AS "sum 3"
  FROM oe.product_information;
```

WINDOWING

- You may find there are times when the window for the analytic function needs to slide with the current row of the result.
  - For example, if you want the total of the last two sales with the current sale.
- The column value of the analytic function is based on the previous rows of the partition and is referred to as the *window*.
  - By default, the window is from the beginning of the partition to the current row.
- You can define windows based on the offset of **ROWS** from the current row.

sliding_orders.sql

```
  SELECT sales_rep_id,
         TO_CHAR(order_date,'Mon DD, YYYY') AS "DATE",
         order_id, order_total,
         SUM(order_total)
           OVER (PARTITION BY sales_rep_id
                     ORDER BY order_date ROWS 2 PRECEDING)
            AS "SLIDING"
    FROM oe.orders
ORDER BY sales_rep_id, order_date;
```

- You can define windows based on ranges of values from the current row.

The following example will select all the orders, but will also return the average **order_total** inside a 14-day window.

orders.sql

```
SELECT sales_rep_id,
       TO_CHAR(order_date,'MON DD,YYYY') AS "DATE",
       order_total,
       AVG(order_total)
         OVER (ORDER BY order_date
               RANGE BETWEEN INTERVAL '7' DAY PRECEDING
               AND INTERVAL '7' DAY FOLLOWING)
         AS "RUNNING AVG"
  FROM oe.orders
```

ROLLUP

- Aggregate functions are very useful for producing summary totals, but do not produce summaries at multiple levels.
  - Oracle8*i* introduced **ROLLUP**, an extension to the **GROUP BY** clause, which generates multi-level summaries for each aggregate column along the non-aggregate columns of the **SELECT** list.
  - This new notation also provides a grand summary for all the aggregate columns.
- **ROLLUP** creates subtotals for aggregate columns, from detailed levels to a final grand total.
  - You provide **ROLLUP** with an ordered list of grouping columns.
  - First, the aggregate values for the columns specified in the **GROUP BY** clause are calculated.
  - Next, progressively higher-level subtotals are calculated, moving from right-to-left through the grouping column list.
  - Finally, a grand total result is created.

rollup1.sql

```
  SELECT department_id, manager_id, job_id, SUM(salary)
    FROM hr.employees
GROUP BY ROLLUP(department_id, manager_id, job_id);
```

- If there are certain columns that you wish to be excluded from the rollup process, then just place them outside the **ROLLUP** clause.

To exclude columns from the summary rows produced by **ROLLUP**, just exclude the column names from the **ROLLUP** clause in the **GROUP BY** clause. The **department_id** field will be excluded from all summaries in the following **SELECT** example:

rollup2.sql

```
   SELECT department_id, manager_id, job_id, SUM(salary)
     FROM hr.employees
GROUP BY department_id, ROLLUP(manager_id, job_id);
```

Some of the subtotal grouping can be combined together in the **ROLLUP** clause by enclosing them in parentheses. These are referred to as *composite columns*. The following example will create a summary line for each unique **department_id/manger_id/job_id** combination and every unique **department_id/manger_id** combination, but not for each **department_id**. A final total line will be produced.

rollup3.sql

```
   SELECT department_id, manager_id, job_id, SUM(salary)
     FROM hr.employees
GROUP BY ROLLUP((department_id,manager_id), job_id);
```

CUBE

- You are limited with **ROLLUP** in that it will only produce subtotals along the listed fields in right-to-left order.
  - **ROLLUP** cannot be used to generate summary information across multiple combinations of the non-aggregate columns.
- You can use **CUBE** to create summaries for a specified set of columns and all of their possible permutations, thus providing the additional cross-tabulation values.
- If the **CUBE** clause is given four fields, then the resulting output will produce 16 (2^4) types of groupings.

cube1.sql

```
  SELECT department_id, manager_id, job_id, SUM(salary)
    FROM hr.employees
GROUP BY CUBE(department_id, manager_id, job_id);
```

- The **CUBE** clause also allows composite columns to eliminate unwanted summary combinations.

cube2.sql

```
  SELECT department_id, manager_id, job_id, SUM(salary)
    FROM hr.employees
GROUP BY CUBE((department_id,manager_id), job_id);
```


Columns that are not included in the grouping combinations will return a **NULL** value. The **GROUPING** function can be used in conjunction with **DECODE** or **CASE** to provide values for columns when they are not included in the summary. **GROUPING** will return a value of **1** if the column is not included in the summary and **0** if it is included. This allows you to provide an alternate value for total rows and allows the user to distinguish between total rows and rows with **NULL** values.

The following example will return the word "total" for any columns that are not included in the groupings on a summary row:

grouping.sql

```
   SELECT DECODE(GROUPING(department_id), 1, 'total', 0, department_id)
            AS dept,
          DECODE(GROUPING(manager_id), 1, 'total' ,0, manager_id)
            AS manager,
          DECODE(GROUPING(job_id), 1, 'total', 0, job_id)
            AS job,
          SUM(salary) AS salary
     FROM hr.employees
GROUP BY CUBE(department_id, manager_id, job_id)
```

GROUPING SETS

- When you are using **ROLLUP** and **CUBE** to produce summary rows, the column combinations for the summary rows are determined by the **GROUP BY** clause.
  - This is a problem if you want to specify which columns to summarize and which to exclude.

group_sales.sql

```
  SELECT sales_rep_id, order_status,
         SUM(order_total) AS "Order Total"
    FROM oe.orders
GROUP BY GROUPING SETS (sales_rep_id, order_status);
```

- **GROUPING SETS** are an extension of the **GROUP BY** clause and allow you to specify multiple groupings of rows within one **SELECT** statement.
- **GROUPING SETS** were created to avoid using the **UNION ALL** set operator and gain query efficiency.
  - Efficient aggregation is gained, since aggregates you do not need are excluded.
  - The groupings are computed for each grouping specified in the **GROUPING SETS** clause and then the results of individual groupings are combined using the **UNION ALL** set operation.
    - Since a **UNION ALL** is being performed duplicate rows may be created.

The following example will produce salary summary rows for each **department_id**, each **manager_id**, and each **department_id/manager_id/job_id** combination.

grouping_sets1.sql

```
   SELECT department_id, manager_id, SUM(salary)
     FROM hr.employees
GROUP BY GROUPING SETS((department_id, manager_id, job_id),
                       department_id,
                       manager_id);
```

The same results could have been achieved using three separate **SELECT** statements with **UNION ALL** clauses between them, but would suffer from less efficiency.

RANK

- There are two techniques you can use to find where a particular row falls within an ordered data set.
  - The first, and very awkward, way is to use an in-line query in the **FROM** clause and then return the **ROWNUM** column in the parent query.
  - The second way is to use the **RANK** function, which returns an incremented number ranging from 1 to the number of rows in the data set.
- The **RANK** and **DENSE_RANK** functions allow you to rank items within a data set or partition.
  - The ranking is based on where a particular row falls within a partition based on the **ORDER BY** clause.
  - The **PARTITION BY** clause can be used to define where the partition breaks will occur.
    - If no **PARTITION BY** clause is provided, then the entire result is considered the partition.
  - The ranking is reset back to **1** at each partition break.

rank1.sql

```
SELECT job_id, employee_id, commission_pct,
       RANK() OVER (PARTITION BY job_id
                          ORDER BY commission_pct)
       AS rank
  FROM hr.employees
 WHERE commission_pct IS NOT NULL;
```


DENSE_RANK is the same as **RANK**, except that if any of the rows have equal values it does not increment the rank values for each of the duplicates.

rank2.sql

```
SELECT job_id, employee_id, commission_pct,
       RANK() OVER (PARTITION BY job_id
                        ORDER BY commission_pct)
       AS rank,
       DENSE_RANK() OVER (PARTITION BY job_id
                             ORDER BY commission_pct)
       AS "dense rank"
  FROM hr.employees
 WHERE commission_pct IS NOT NULL;
```

MODELING

- The **MODEL** clause lets you create multi-dimensional arrays from a query result, and then apply various rules (similar to what you might find in a spreadsheet) to calculate new values.
  - The rules are formulas and vary from basic arithmetical expressions to complicated equations using recursion.
- You avoid transferring large amounts of data between various modeling environments by defining the models within the database engine.
- The **MODEL** clause maps columns of a query into three groups to define a multi-dimensional array.
  - The rules of the **MODEL** clause are applied to each grouping defined by the *Partition* columns.
  - The *Dimension* columns are used to define cells within the partition.
  - The *Measures* are the cells that are identified by the Dimension.
- You can think of the query as a mechanism to populate a spreadsheet and the rules defined in the model are used to manipulate the spreadsheet.

model1.sql

```
SELECT rep, year, total
  FROM year_orders
 MODEL
    PARTITION BY (order_mode)
    DIMENSION BY (rep, year)
    MEASURES (total)
    RULES upsert
      ( total[159,1999] = 3000,
        total[163,1999] = total[163,1998] + total[163,1]
      );
```


These examples use the **YEAR_ORDERS** view:

view.sql

```
CREATE OR REPLACE VIEW year_orders (rep, year, order_mode, total)
  AS SELECT sales_rep_id,
            TO_CHAR(order_date, 'YYYY'),
            order_mode,
            SUM(order_total)
       FROM oe.orders
   GROUP BY sales_rep_id,
            TO_CHAR(order_date, 'YYYY'),
            order_mode;
```

MODEL CLAUSES

- There are certain limitations to the use of the **MODEL** clause:
  - The **MODEL** clause only returns the table data as a data set, whether or not it has been modified.
  - The **MODEL** clause does not update the table of the parent **SELECT** statement.
- The **RETURN UPDATED ROWS** clause makes the query return only those rows that are updated or inserted by the model rules.
  - The default is **RETURN ALL ROWS**.
- Each rule, or all the rules collectively, can have a clause to specify how cells should be treated.
  - **UPSERT —** Updates the cells or creates new ones if they do not exist.
  - **UPSERT ALL —** Same as **UPSERT**, except it allows a broader set of rule notation for cell insertion.
  - **UPDATE —** Only updates existing cells.
  - **IGNORE NAV —** Numeric cells that are null treated a **0**.
  - **KEEP NAV —** Keeps cell values null that were not available.

The following **SELECT** statement has two rules to update the result with new totals for rep 159 in year 2005 and rep 300 in year 2005.

model2.sql

```
SELECT rep, year, total
  FROM year_orders
 MODEL
    RETURN UPDATED ROWS
    PARTITION BY (order_mode)
    DIMENSION BY (rep, year)
    MEASURES (total)
    RULES upsert
       ( total[159,2005] = 3000,
         total[300,2005] = 4000
      );
```

The **SELECT** statement below has two rules that will update the results and one rule that will insert a new row if one does not exist.

model3.sql

```
SELECT rep, year, total, order_mode
  FROM year_orders
 MODEL
    RETURN UPDATED ROWS
    PARTITION BY (order_mode)
    DIMENSION BY (rep, year)
    MEASURES (total)
    RULES UPDATE
      ( total[159,2005] = 3000,
        total[163,1999] = 5000,
        UPSERT total[300,2005] = 4000
      );
```

LABS

❶ Create a query to list all the **order_date**, **order_id**, **order_total**, and the **total** for all the orders on the same day.
(Solution: *order_date_total.sql*)

❷ Query the database to rank the **sales_rep_id** in order of their total sales for 1999 in descending order.
(Solution: *rank_1999.sql*)

❸ Retrieve the **product_id**, **warehouse_id**, and **quantity_on_hand** from the **inventories** table and get **quantity_on_hand** totals for each product.
(Solution: *products_on_hand.sql*)

❹ List the **customer_id**, **gender**, **territory**, **credit_limit**, and the average **credit_limit** for each gender in that territory from the customers.
(Solution: *gender_terr_avg.sql*)

❺ Create a query to return the **employee_id**, **department_id**, and **salary** from the **employees** table. If the employee is in department 50, then add $20 to the return value of the **salary**.
(Solution: *dept_50_bonus1.sql*)

❻ Modify the previous query so if any **salary** field is null it will still reflect the $20 bonus amount.
(Solution: *dept_50_bonus2.sql*)

❼ Modify the previous query so the employees' **commission_pct** field will be returned as **.4**.
(Solution: *dept_50_bonus3.sql*)

