Exam Manual

LearnSmart

(1Z0-051)

Oracle 11g

SQL Fundamentals I

This LearnSmart exam manual covers the most important concepts you need to master in order to successfully complete the Oracle 11g SQL Fundamentals 1 exam (1Z0-051). By studying this guide, you will become familiar with an array of exam-related content, including:

- Retrieving Data using the SQL SELECT Statement
- Restricting and Sorting Data
- Using Subqueries to Solve Queries
- Using the Set Operators
- And more!

Give yourself the competitive edge necessary to further your career as an IT professional and purchase this exam manual today!

Oracle Database 11g: SQL Fundamentals I (1Z0-051) LearnSmart Exam Manual

Copyright © 2011 by PrepLogic, LLC.

Product ID: 12045
Production Date: July 14, 2011

All rights reserved. No part of this document shall be stored in a retrieval system or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein.

Warning and Disclaimer

Every effort has been made to make this document as complete and as accurate as possible, but no warranty or fitness is implied. The publisher and authors assume no responsibility for errors or omissions. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this document.

LearnSmart Cloud Classroom, LearnSmart Video Training, Printables, Lecture Series, Quiz Me Series, Awdeeo, PrepLogic and other PrepLogic logos are trademarks or registered trademarks of PrepLogic, LLC. All other trademarks not owned by PrepLogic that appear in the software or on the Web Site (s) are the property of their respective owners.

Volume, Corporate, and Educational Sales

Favorable discounts are offered on all products when ordered in quantity. For more information, please contact us directly:

1-800-418-6789
solutions@learnsmartsystems.com

International Contact Information

International: +1 (813) 769-0920

United Kingdom: (0) 20 8816 8036

Table of Contents

Abstract 5
What to Know 5
Tips 6
1.0 Oracle Background 8
2.0 Data Retrieval Using SELECT 10
DESCRIBE 10
The Simple SELECT 12
3.0 Restricting and Sorting with SELECT 15
The WHERE Clause 15
The ORDER BY Clause 17
Substitution and Session Variables 18
Substitution Examples 19
4.0 Single-Row Functions 20
Single-Row Functions 22
Examples of Single-Row Functions 25
5.0 Conversion Functions and Conditional Expressions 28
Key Conversion Functions 29
Handling NULL Values 30
CASE Expressions and the DECODE Function 31
6.0 Group Functions 32
GROUP BY Clause 34
The HAVING Clause 34
The GROUP Functions 35
Key GROUP Functions 36
7.0 Joining Data Across Tables 39
ANSI Joins 40
Oracle Syntax Joins 41
8.0 Subqueries 42
Kinds of Problems Subqueries Can Solve 43
9.0 Set Operators 45
Example Set Operator Queries 46
10.0 Updating Data Through DML 47
INSERT 47
UPDATE and DELETE 48

MERGE ... 48
Example INSERT, UPDATE, DELETE, and MERGE Statements ... 48
TRUNCATE ... 49
Transactions ... 50
SAVEPOINTS ... 50
Transactions in Developer Tools ... 50
SELECT… FOR UPDATE ... 50
11.0 Creating Tables Through DDL ... **51**
Creating Tables ... 52
Modifying Existing Tables ... 55
Table Operations ... 56
Constraints ... 56
More on Foreign Key Constraints ... 58
12. 0 Creating Other Schema Objects ... **59**
Views ... 59
Synonyms ... 60
Sequences ... 61
Indexes ... 62
Practice Questions ... **64**
Answers & Explanations ... **69**

Abstract

This Exam Manual prepares you for the Oracle 11g certification exam #1Z0-051, "Oracle Database 11g: SQL Fundamentals I." Passing this exam plus the test entitled "Oracle 11g Administration I 1Z0-052" makes you an Oracle 11g Certified Associate (an "OCA").

This Exam Manual is your "Cliffs Notes" key to the test material. It is a short, highly-targeted summary of what's on the exam. It can't cover everything, but if you understand and memorize its contents, you will be very far along towards passing the exam.

The exam covers the SQL language in depth -- from an Oracle standpoint. Test topics include SQL language elements (operators, precedence, notation, and syntax), SELECT statements (keywords, selection, sorting, joins, subqueries, and CASE expressions), SELECT statement functions (single-row, group, and conversion), updating data (INSERT, UPDATE, DELETE, and MERGE statements), set operators, creating common objects (including tables, views, indexes, sequences, and synonyms), and basic table characteristics, such as data types and constraints.

What to Know

You need to know the SQL language very well to pass this pass. This includes operators, syntax, notation, special characters, and the like. Test questions will confront you with very minor syntax differences in SQL statements so you have to remember SQL and its syntax well enough not to become confused.

The SQL covered is that typically used by developers in their database work. This includes DML statements (SELECT, INSERT, UPDATE, DELETE, MERGE), transaction control statements (COMMIT, ROLLBACK, SAVEPOINT), and the DDL statements that manage the kinds of objects that developers deal with (CREATE / ALTER / DROP TABLE, INDEX, VIEW, SYNONYM, SEQUENCE).

The test does **not** cover the SQL statements used by database administrators to create or manage physical storage (CREATE / ALTER / DROP DATABASE, TABLESPACE), partitioned tablespaces, or TABLE storage parameters).

Know the SELECT statement, all its keywords and options, very well.

Understand Joins and how to perform them. Given a Join statement, be able to determine the result set.

Given a SELECT having a subquery, be able to determine its result set. Why does one use subqueries?

Understand the common uses of the INSERT, UPDATE, DELETE, and MERGE statements and their syntax.

What are transactions? How are they used and why are they important? Know why you issue COMMIT and ROLLBACK statements.

Know the difference between single-row and group functions and why each is used. Study the conversion functions too.

Know how to create and alter tables, views, indexes, synonyms, and sequences. What is each object and for what is it used?

What are Oracle 11g's data types and how do you declare them when creating tables? How and why does one apply constraints to tables?

Tips

- The test consists of multiple-choice questions with one correct answer. It also contains multiple-choice questions where you select 2 or 3 correct items out of a list of 5 or 6 possible answers.
- For multiple-answer questions, you must select every correct answer and none of the incorrect answers in order to get credit for correctly answering the question. There is no "partial credit."
- Be sure to answer every question -- there is no penalty for guessing.
- Read each question carefully and read all alternatives before picking an answer. Often the differences between answers are very slight and you don't want to pick a wrong answer just because you didn't read the question carefully.
- The answers to many questions will not be immediately evident. In this case it helps to use a process of elimination. Eliminate answers you know are wrong, and you'll often then be able to take your best guess from the remaining couple alternatives.
- You can always mark a question you're not sure about for later review.
- It is not unusual to find information in other questions that will later help you answer the questions you've marked for review.

To pass any Oracle exam, you need to do four things –

1. Verify Oracle Corp's exam requirements
2. Get hands-on experience with the database
3. Study a book written specifically to help candidates pass the test
4. Work with practice questions

Verify Oracle Corp's Exam Requirements:
You need to verify the exam requirements at Oracle Corporation's web site because the company reserves the right to change them at any time. Usually they only change the tests when they come out with a new version of the Oracle database, but on occasion they do make minor modifications to the exams between releases.

To view the exam requirements, go to Oracle's certification home page here. (Sometimes Oracle redirects incoming links so you may encounter a web page where you select your country first. Then select "Certification" from the menu on the resulting page. Make sure your browser accepts cookies.)

Hands-on Experience:
You can either get hands-on experience with Oracle 11g at work, or you can install the free version of Oracle database on your personal computer. The free version of Oracle is called "Oracle Express." You can download it to run on either Windows or Linux.

Download Oracle Express Edition from www.oracle.com and enter the words "oracle express download" in the Search Box.

Oracle Express Edition on your personal computer allows you to work with Oracle SQL and can provide sufficient experience to pass this exam. At the time of writing, 10g is the latest version of Oracle Express you can download – but this is perfectly fine for the purposes of this particular exam.

Books:
Search at Amazon online and you will find several books that are written specifically to help you pass this exam. Read reviewers' comments to select which you prefer. Or go to a bricks-and-mortar bookstore and peruse the books yourself. It is important to select a book you like and have confidence in… you will be spending a lot of time studying it.

Here are two popular certification books for this exam –

OCA: Oracle Database 11g SQL Fundamentals I Exam Guide: Exam 1Z0-051. By John Watson and Roopesh Ramklass (Osborne Oracle Press Series).

- OCA: Oracle Database 11g Administrator Certified Associate Study Guide: (Exams 1Z0-051 and 1Z0-052). By Biju Thomas (Sybex). This book covers both the exams you need to become an Oracle Certified Associate.
- The Oracle manual on which this test is based is the SQL Language Reference.

Practice Questions:
Practice questions get you ready to face the real exam. They give you a feel for what the tests are like, and they also help you conceptualize your knowledge of Oracle in terms of the kinds of questions you'll face in the real exam.

We recommend the practice tests available from LearnSmart. LearnSmart has earned a reputation for quality practice tests that prepare candidates well for certification exams.

Oracle Corporation offers ten free sample questions at their certification web site. These questions show you what the test questions look like -- but the sample is far too small to help you learn what you need to know to pass the exam.

1.0 Oracle Background

The Oracle **relational database management system**, or **RDBMS**, treats all stored data as **tables—**rows of data, in other words. Every table is defined as having one or more **columns**, each of which represents a particular kind of data or **attribute** stored in the table. The obvious things that should come to mind are the rows and tables of an Excel spreadsheet. In Excel, the intersection of a row and column is called a **cell**; in an Oracle table, we call this intersection a **column value** or **data element**.

While Oracle RDBMS may physically store data in various ways, when you retrieve the data it always appears as a relational table. You retrieve data by running a **query** against one or more tables. The data Oracle returns to you appears as a table and is called the **result set**.

Structured Query Language or **SQL** is the language used to query Oracle databases. SQL is standardized by international bodies such as the American National Standards Institute (ANSI) and the International Standards Organization (ISO). Oracle's SQL fully conforms to the ANSI:1999 standard and partially complies with the SQL:2003 standard endorsed by both ANSI and the ISO.

SQL is based on a mathematical foundation called **relational theory**. It implements relational algebra to operate upon tables and return the desired result sets. SQL is a set-oriented language, meaning that a single SQL statement can affect multiple database rows.

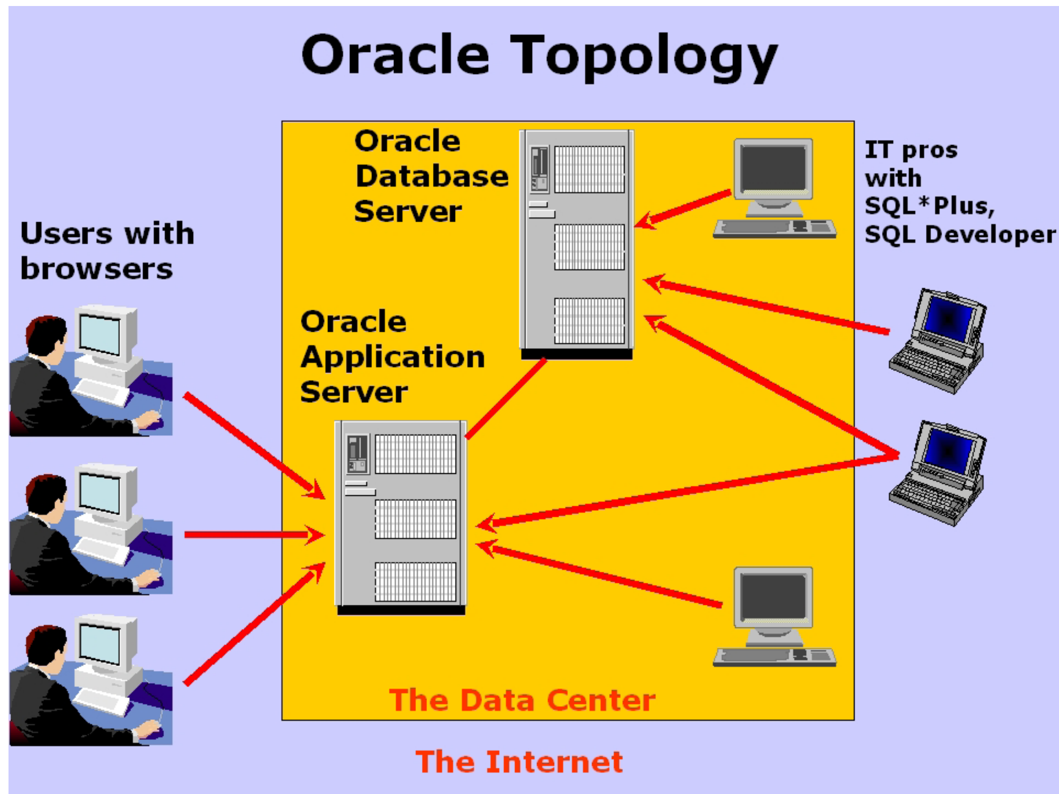
Oracle tables are often designed using **entity-relationship modeling**. Database design includes the **normalization** process to achieve **third normal form**, a design goal that ensures **data integrity** or data accuracy.

Though SQL is a powerful data manipulation language, it lacks many common programming features you might encounter in other languages, like flow of control statements (If-Then-Else, Do-While, etc). Therefore Oracle supplies the **PL/SQL** language, a full programming language built on top of SQL. You can develop complete applications in PL/SQL. Or, you can use traditional programming languages, like Java.

SQL*Plus and **SQL Developer** are two Oracle **client-based tools** you use to issue SQL queries and obtain results sets. The latter is designed to help programmers build complete applications. Many third-party tools are also available.

Whichever tool you use, you must create a **client connection** between your personal computer and the Oracle RDBMS to retrieve data. **Oracle Net** is the communications software that supports the communications connection between your personal computer and the Oracle database. Oracle Net is a layered protocol that runs on top of your underlying network protocol (such as TCP/IP for internet connections). Results sets are sent from Oracle to your computer using Oracle Net.

The diagram below shows how Oracle RDMS runs on a database server, while **Oracle Application Server** runs applications that connect to it. Users connect locally or remotely via their browsers to use the programs on the application server, which gets its data from the database server. Developers use client tools like SQL*Plus and SQL Developer either locally or remotely to develop and support applications. Oracle's **Grid Control,** based on its **Oracle Enterprise Manager** graphical user interface**,** manages the entire environment. An alternative to Grid Control is Oracle's **Database Control,** which manages a single database instead of several databases and their environment.



A **schema** is the group of all **database objects** (such as tables, indexes, etc) owned by a particular user. Two example or **demonstration schemas** come with Oracle and are used for tutorial purposes in the Oracle manuals and on this exam. They are **HR** (Human Resources) and **OE** (Order Entry). The demonstration schemas are usually created when you first install Oracle and create a database (their creation is an option in the **Database Configuration Assistant** tool). If the schemas were not created during install you can create them by running Oracle-provided scripts. It's easy to run the scripts but the details vary by platform so see the appropriate Oracle installation guide for details. Usually you'll find the scripts in a directory or folder labeled **demo** within your Oracle Home directory. You'll want to practice SQL statements on Oracle's two demo schemas. Exam questions continually refer to them.

For your reference while working with this Exam Manual, here are the table definitions for three key tables from the HR schema that the exam and this Manual use as examples.

Departments:

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

Employees:

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Job History:

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10
DEPARTMENT_ID		NUMBER(4)

2.0 Data Retrieval Using SELECT

The SELECT statement is the SQL statement used for retrieving data or **results sets**. SELECT statements perform read-only operations that do not change the table data upon which they operate. SELECT reads data from relational tables and reports results back in tabular format.

2.1 DESCRIBE

Use the DESCRIBE statement to discover the format of a table from Oracle's **data dictionary**. The data dictionary contains table definitions or **metadata** about tables. To learn the format of the HR schema's DEPARTMENTS table issue this DESCRIBE statement:

```
DESCRIBE  DEPARTMENTS;
```

Most SQL **keywords** like DESCRIBE have allowable abbreviations (DESC may be used instead of DESCRIBE). SQL statements and their keywords may be coded in either upper-, lower-, or mixed case. SQL is a **case-insensitive** language. But the data you store within the database is **case-sensitive**. It can be upper-case, lower-case, or mixed-case.

SQL statements are terminated by a semi-colon. (In SQL*Plus you'll often see the final SQL statement followed by a forward slash (/), which tells SQL*Plus to run the statement(s).)

In the DESCRIBE statement, the table name may optionally be preceded by the schema name. The table name *must* be preceded by the schema name if you are querying a table from a schema other than that to which you are connected. The example above only runs if you are connected as the user account HR and referring to the table HR.DEPARTMENTS.

These rules mean that, assuming you are connected to the HR schema, all these are valid variations on the previous statement:

```
DESC  DEPARTMENTS;
DESC  HR.DEPARTMENTS;
Describe  Departments;
desc  hr.departments;
```

The general format for the DESCRIBE statement is:

```
DESC[RIBE]  [schema_name].table_name ;
```

You replace the lower-case portion of the command with the proper information. The upper-case keywords are coded exactly as shown. Square brackets ([]) enclosing an item indicate it is optional. Here this means that you can code either DESCRIBE or DESC, and that the **schema_name** may be optional. SQL keywords are traditionally shown in uppercase (though they can actually be encoded in either upper-, lower-, or mixed case.) This syntax is the **conventional notation** used in the Oracle manuals. You should be able to read it for the exam.

Output from the DESCRIBE command lists the columns that comprise the targeted table. Each column is listed along with its **data type**, the kind of data that column contains:

```
SQL> desc hr.departments;
Name                                       Null?         Type
------------------------------------------ -----------   ----------------------------
DEPARTMENT_ID                              NOT NULL      NUMBER(4)
DEPARTMENT_NAME                            NOT NULL      VARCHAR2(30)
MANAGER_ID                                               NUMBER(6)
LOCATION_ID                                              NUMBER(4)
```

Oracle has a couple dozen data types. A few of the most common are:

Data Type:	**Used For:**	**Description:**
NUMBER	Numbers	NUMBER(p,s) where p is the precision and s is the scale. **Precision** tells how many digits NUMBER contains while **scale** is the number of digits to the right of the decimal place. e.g.: NUMBER(8,2) means 123456.78
VARCHAR2	Character strings	VARCHAR2(length) stores variable-length character data up to a maximum specified by its length. Maximum length is 4,000 bytes.
DATE	Dates	Date and Time. Includes Century, Year, Month, Day, Hour, Minute, and Second.
TIMESTAMP	Timestamps	The same information as a DATE but includes fractional seconds.

Some table columns are defined such that they are always required to contain data. They are defined as NOT NULL. **NULL** in Oracle refers to an absence of data. A **nullable column** is a column that is not required to contain data. If a column attribute does not contain any data it is often spoken of as "containing nulls." This is not strictly accurate – the column field contains nothing. Nulls means the absence of data.

Blank spaces do *not* count as NULL because they are present in the row and consume space. Thus setting a field to blanks can fulfill a NOT NULL definition for a character data type column. 0 (zero) is the same. It is a value that consumes space and it differs from NULL.

2.2 The Simple SELECT

The three key SELECT operations are:

Operation:	**Use:**
Projection	Restricts **attributes** (columns) in the result set
Selection	Restricts **tuples** (rows) in the result set
Joining	Combines selected rows from two or more tables into the result set

Here are some basic formats of the SELECT statement:

SELECT * | {[DISTINCT] column | expression [alias],…} FROM table ;
SELECT column [possibly more columns or expressions] [alias] FROM table ;
SELECT DISTINCT column [possibly more columns or expressions] [alias] FROM table ;

In these statement formats, SQL keywords are capitalized, while values you provide are in lower-case. The vertical bar (|) means "or" – select either the item to the right of the bar or to the left of it, but not both items. If one of those items is the default, it is underlined. Optionally coded items are surrounded by brackets ([ ]). This is the standard syntax from the Oracle manuals with which you should be familiar.

The asterisk (*) denotes "all columns" in a table, so this statement returns all columns from the named table in your schema:

```
SELECT * FROM my_table;
```

The DISTINCT keyword will eliminate duplicates in the result set. *If applied to more than one column, it returns only unique combinations of those columns.*

Let's look at an example using the HR.JOB_HISTORY table. First, here are the contents of the JOB_ID and DEPARTMENT_ID columns in the HR.JOB_HISTORY table:

```
SQL> select job_id, department_id from hr.job_history;

JOB_ID        DEPARTMENT_ID
----------    -------------
IT_PROG                  60
AC_ACCOUNT              110
AC_MGR                  110
MK_REP                   20
ST_CLERK                 50
ST_CLERK                 50
AD_ASST                  90
SA_REP                   80
SA_MAN                   80
AC_ACCOUNT               90

10 rows selected.
```

Now, look at these SQL queries on the HR.JOB_HISTORY table:

SQL Query:	**Number of Rows Returned**:
SELECT DISTINCT job_id FROM job_history;	8
SELECT DISTINCT department_id FROM job_history;	6
SELECT DISTINCT job_id, department_id FROM job_history;	9

The last query above returns more rows than the first two because its DISTINCT keyword eliminates only duplicate combinations of the two columns, JOB_ID and DEPARTMENT_ID. Look at the table data to make sure you understand how this works.

An **alias** is an alternate name for a column or expression. It's typically used to produce more user-friendly output, since aliases replace column names as column labels in query output. These two example statements show that the output will label the JOB_ID column as "Job Code" instead:

```
SELECT  employee_id, job_id  "Job Code"  FROM  hr.job_history;

SELECT  employee_id, job_id  AS  "Job Code"  FROM  hr.job_history;
```

The keyword AS is optional in denoting an alias. The presence or absence of the comma (,) between elements is what distinguishes columns from aliases in the **SELECT list** of the SELECT statement. The example alias "Job Code" must appear in double-quotes because it contains an embedded blank. The double-quotes also preserve the mixed case in "Job Code". If your alias does not contain an embedded blank, and you don't care that the alias will appear as an all upper-case column label in the output, then you can code the alias without the doublel-quotes.

SQL **expressions** and **operators** give SQL much of its power. Expressions usually perform operations on one or more column values. The operators are either arithmetic (for **numeric** values), concatenation (for **character string** values), or addition and subtraction (for **date** and **timestamp** values).

The **order of precedence** (or order of evaluation) for operators is, from highest to lowest:

Precedence:	**Operator Symbol(s):**	**Operation(s):**
1.	( )	Parentheses
2.	* /	Multiplication and division
3.	+ - ||	Addition, subtraction, and concatenation
4.	= < > <= >=	Equality and inequality comparison
5.	[NOT] LIKE, IS [NOT] NULL, [NOT] IN	Pattern, null, and set comparison
6.	[NOT] BETWEEN	Range comparison
7.	!= <>	Not equal to
8.	NOT	NOT logical operator
9.	AND	AND logical operator
10.	OR	OR logical operator

Chapter 3 covers the **comparison operators** in rows 4 thru 7 and the **logical operators** in lines 8 thru 10.

Operators of equal precedence are processed from left to right. If you need to dictate the order of expression evaluation, you can always use parentheses ( ) to indicate what to evaluate first, because they have the highest precedence.

An arithmetic operation with a NULL value always returns NULL. **Concatenation** is the joining together of character strings. A NULL value within any concatenation operation is just ignored.

Let's discuss example statements that illustrate these operators:

1.	SELECT 2 + 2 FROM dual;
2.	SELECT last_name || ',' || first_name AS person FROM employees;
3.	SELECT (4 + 4) * 8 FROM dual;
4.	SELECT employee_id, start_date, end_date, (end_date – start_date) + 1 days_employed FROM job_history;
5.	SELECT 'Employee of the year is:', last_name, first_name FROM employees;

1. This query returns the result of the arithmetic calculation (4), as indicated by the arithmetic operator (+). DUAL is a special table available in every Oracle database. It contains a single column called DUMMY and used in situations like this one, where you want to resolve an expression in a SQL statement and don't care what table is referred to in the required FROM clause.

2. This query lists everyone in the EMPLOYEES table. It uses the concatenation operator to splice together employee names, with the last name listed first. A comma and one blank follow each last name. *You* ***can't*** *assume any order to the output unless you add the ORDER BY clause to the SELECT statement* (see chapter 3):

   ```
   PERSON
   ----------------------------------------------
   Abel, Ellen
   Ande, Sundar                    (continued…)
   ```

3. This query returns the result of 64. The same query with the parentheses removed returns 36. Coded without any parentheses, the multiplication operation will occur first, due to its higher order of precedence than addition. With parentheses, the addition occurs first. This example demonstrates the importance of the precedence order for operators, and how to override it through use of parentheses.

4. This query lists employees and the number of days they have been employed. The column labeled DAYS_EMPLOYED by the alias shows the results of the "days employed" calculation. Notice how the calculation is applied to the values returned from the START_DATE and END_DATE columns. One day is added to the result of the calculation so that no employee shows 0 days employed. The column label DAYS_EMPLOYED will appear in upper-case – even though it's encoded in lower-case – because this column alias is not enclosed in double quotes to preserve its case.

5. This query shows use of a character string **literal**, a character string enclosed in single quotes. The output contains a series of lines with this literal, each with a different employee's name:

```
'EMPLOYEEOFTHEYEARIS:'   LAST_NAME        FIRST_NAME
------------------------------ -------------------- ---------------------------
Employee of the year is:   Abel             Ellen
Employee of the year is:   Ande             Sundar     (continued…)
```

How would you replace the goofy column heading for the literal with something more presentable? Use a column alias in the query.

3.0 Restricting and Sorting with SELECT

Selection restricts the set of rows returned to a query. The SELECT statement's WHERE clause accomplishes this. Only rows from the target table(s) that satisfy the WHERE clause are returned by the query. A SELECT statement without a WHERE clause returns every row in the table—potentially a performance issue for a very large table.

Rows in every table are always considered unordered. You cannot assume or count on rows being returned in any particular order unless you code the SELECT statement ORDER BY clause. Using the ORDER BY clause, you can sort the rows of your result set by one or more columns, and you can specify ascending or descending order for each sorted column.

This is the basic template for a SELECT statement with WHERE and/or ORDER BY clauses. As per standard Oracle notation, defaults are underlined:

```
SELECT  * | {[DISTINCT] column | expression [alias], …}
FROM  table
[WHERE  condition(s)]
[ORDER BY {col(s) | expr | numeric_pos} [ASC | DESC]  [NULLS FIRST | LAST]];
```

3.1 The WHERE Clause

WHERE clause conditions can test numeric, character, or DATE data values. WHERE clause conditions should be coded appropriate for the data type of the column(s) to be tested, but Oracle will perform **implicit data type conversions** if required whenever possible. For example, in the HR.EMPLOYEES the column SALARY is defined as NUMBER(8,2). Due to Oracle's implicit type conversions either of these two statements will yield the same result:

```
SELECT job_id, salary FROM employees WHERE salary > 15000 ;
SELECT job_id, salary FROM employees WHERE salary > '15000' ;
```

DATE values are considered equal when all their internal components are equal (century, year, month, day, hours, minutes, and seconds). Oracle's default date format is **DD-MON-RR**. RR is the year. If RR is between 50 and 99, Oracle returns the 20th century, otherwise it returns the 21st century. You can alternatively specify a four-digit year, YYYY. So the first two queries below are equivalent:

```
SELECT start_date FROM job_history WHERE start_date > '01-JAN-1998' ;

SELECT start_date FROM job_history WHERE start_date > '01-JAN-98' ;

SELECT job_id FROM job_history WHERE start_date = end_date ;
```

The last statement above points out that you can compare columns to each other in WHERE clauses. In fact, the terms *on either side* of the comparison coded within the WHERE clause can be columns, literal values, or expressions. For example, this statement contains expressions on both sides of the comparison operator in the WHERE clause:

```
SELECT employee_id FROM job_history WHERE start_date – 1 < end_date + 1 ;
```

Note that columns used in the WHERE clause do *not* have to be in the **SELECT list** (the list of columns appearing immediately after the SELECT keyword.) The above example shows this by coding both the START_DATE and END_DATE columns in the WHERE clause but not in the SELECT list.

The **comparison operators** used in WHERE clauses are like those used in many other programming languages. Either != or <> may be used to denote "not equal to":

```
=    !=    <>    <    >    <=    >=
```

WHERE clauses can include multiple conditions through use of the **Boolean** or **logical operators** AND, OR, and NOT:

Operator:	**Meaning:**
AND	Both WHERE conditions linked by the AND must be TRUE for the row to be returned
OR	Either WHERE condition linked by the OR can be TRUE and the row will be returned
NOT	A row must conform the logical opposite of the WHERE condition in order to be returned

Here's an example. Note that the WHERE keyword itself only appears once in the SQL statement; it is *not* coded for each condition tested. In this example rows must meet all three of the conditions in the WHERE clause to become part of the result set, since the conditions are linked by keyword AND:

```
SELECT distinct job_id FROM employees WHERE hire_date < '01-JAN-2000'
                    AND salary < 15000  AND department_id <> 50  ;
```

According to the operator precedence table given earlier in Chapter 2, the precedence of the logical operators from highest to lowest is: NOT, AND, OR. If you have a complicated WHERE clause with several conditions, remember that you can always use high-precedence parentheses to dictate the order in which the conditions are applied.

There are several more operators used in WHERE clauses:

Operator:	**Meaning / Example:**
BETWEEN	Tests if a column or expression value falls within an inclusive range. This example returns rows where the salary is greater than or equal to 10000 and less than or equal to 20000: WHERE salary BETWEEN 10000 and 20000 ;
IN	Tests if a column or expression value is a member of the specified set of literal values. This example returns rows where the salary is 10000, 20000, or 30000: WHERE salary IN (10000, 20000, 30000) ;
IS NULL	Tests if a column value is NULL. This example returns rows where no value has yet been assigned to the salary: WHERE salary IS NULL ;
IS NOT NULL	Applies NOT to IS NULL to return rows where there exists a valid non-null value for a column. Example: WHERE salary IS NOT NULL ;
LIKE	Applies a specified character pattern comparison to a character string. The underscore character (_) may be used to represent any one character while the percentage sign (%) may be used to represent any zero or more characters. Here are a few examples: WHERE first_name LIKE '%' ; Returns every non-null first_name WHERE email LIKE 'A%' ; Returns every email that starts with capital A WHERE email LIKE 'A_' ; Returns every email that starts with capital A and is followed by exactly one more arbitrary character WHERE email LIKE '%A%' ; Returns every email that contains a capital A

What if you have a LIKE comparison in which you actually want to use the percent symbol (%) or the underscore (_) as part of the pattern to compare? Use the ESCAPE keyword and you can denote that the symbol is literal and not to be interpreted as a special character. For example, to search the EMAIL column for the exact character string, "**AB%**", you would code:

```
SELECT email FROM employees WHERE email LIKE 'AB\%' ESCAPE '\' ;
```

The ESCAPE keyword identifies an **escape character**. Any single special character encoded immediately after this escape character in the LIKE pattern is treated by Oracle as the literal character encoded (and not as a special character). So in this case, the percent sign after the LIKE keyword is treated as an actual percentage sign, rather than as the special character that matches any zero or more arbitrary characters.

3.2 The ORDER BY Clause

For reference, here is the ORDER BY clause, again:

[ORDER BY {col(s) | expr | numeric_pos} [ASC | DESC] [NULLS FIRST | LAST]];

If a query contains both a WHERE clause and an ORDER BY clause, the ORDER BY goes last. *Like WHERE clauses, ORDER BY clauses may refer to column(s) not in the SELECT list.*

You can sort query results by more than a single column. In this case, the columns are applied to the sort from left to right (the leftmost column is the first-applied sort column and the rightmost is the last). Thus, we call the leftmost column the **major sort** and the rightmost the **minor sort**. Here's an example:

```
SELECT department_id, last_name, first_name FROM employees
        ORDER BY department_id, last_name DESC, 3 ;
```

This query lists employees sorted by department, and then within each department, by their last names in reverse alphabetical order. The **3** refers to the third item encoded in the SELECT list, in this case the column FIRST_NAME. So, this query's results are actually sorted by: ascending DEPARTMENT_ID, then LAST_NAME in descending order, and *then* ascending FIRST_NAME. Using a number like the 3 to refer to a column in the SELECT list is called a **positional sort parameter**. This query also demonstrates that different columns can each be independently sorted as ascending (the default) or descending. You can sort ascending on some columns and descending on others.

If you run this query with the sample data Oracle provides in the HR schema, you'll find that one employee has no department; it is NULL. This employee sorts last in the report, as the default is NULLS LAST. If you'd prefer to have NULL columns sort first, encode the NULLS FIRST keywords.

3.3 Substitution and Session Variables

So that you can save effort by re-using queries, Oracle allows you to substitute values for variables in SQL statements. This enables you to write SQL queries you store and re-use over and over. To make this possible, in a SQL query, a variable preceded by a single ampersand invokes **single ampersand substitution**. If the variable is not already defined for the session, Oracle prompts you to enter the required value.

Normally you'll see single ampersand substitution used to place values into variables. But it is technically possible to use it to substitute in a character string for any part of the SQL statement beyond the first word. For example, this is legal usage:

```
SELECT  &rest_of_statement ;
```

Or you could even code this ultimately-flexible SELECT statement:

```
SELECT       &select_clause
FROM         &from_clause
WHERE        &where_clause
ORDER BY     &order_by_clause ;
```

What if you wanted to reference the same substitutable input variable multiple times? Using single ampersand substitution would require you to enter that variable value each time Oracle encounters the same single ampersand variable in the SQL statement. **Double ampersand substitution** gives you a way around this problem. Coding a substitutable variable with two preceding ampersands tells Oracle to use the same value each time the same variable appears in the code.

A **session** is a user interaction with Oracle through a client tool like SQL*Plus or SQL Developer. A session ends when the user exits SQL*Plus (or similar tool)—either on purpose or because the user process abnormally terminated.

When coding with single and double ampersand substitution, you need a way to manage the **session variables** to which they refer. Oracle provides several SQL client control commands for this purpose. *These are not SQL statements but rather Oracle-provided commands for controlling the behavior of the client tool SQL*Plus.* The semi-colon statement terminator, required for SQL statements, is optional with these SQL*Plus commands:

Command	**Use**
DEFINE;	List variables defined in your session and their values
DEFINE *variable=value;*	Creates a session variable and assigns it a value
UNDEFINE *variable;*	Removes a session variable (and its value)
SET VERIFY <u>ON</u> | OFF;	The default, VERIFY ON, causes Oracle to display the contents of substitutable variables both before and after the substitution
SET DEFINE <u>ON</u> | OFF;	Turning off DEFINE means that the client tool will not save session variables. By default DEFINE is ON

3.4 Substitution Examples

This first example shows entering a SELECT statement with single ampersand substitution for a single variable. Since that variable has not yet been defined in the session, Oracle prompts for its value. Here, the user entered **department_id** in response:

```
SQL> SELECT job_id, &colname FROM job_history ;
Enter value for colname:department_id
```

In this next example we first define the value of the variable **&colname** to our session by using the DEFINE command. When Oracle encounters the same SELECT statement as in the previous example, it replaces variable **&colname** with the session variable's value—it does not need to prompt the user to enter a value. The result of the query is the exact same as that of the previous example:

```
DEFINE colname=department_id ;
SELECT job_id, &colname FROM job_history;
```

In a new SQL*Plus session, we enter the query below. Oracle will prompt us twice for the value of **&colname** (one time for each of its two occurrences in the statement):

```
SELECT job_id, &colname FROM job_history WHERE &colname =  60 ;
```

If we prefer to be prompted only one time for **&colname**, we should use double ampersand substitution, as shown below. Oracle prompts us one time for a value for **&&colname** and then uses it in both substitutions. Assuming we enter the same string to Oracle's prompt, this query gives the exact same results as the one above:

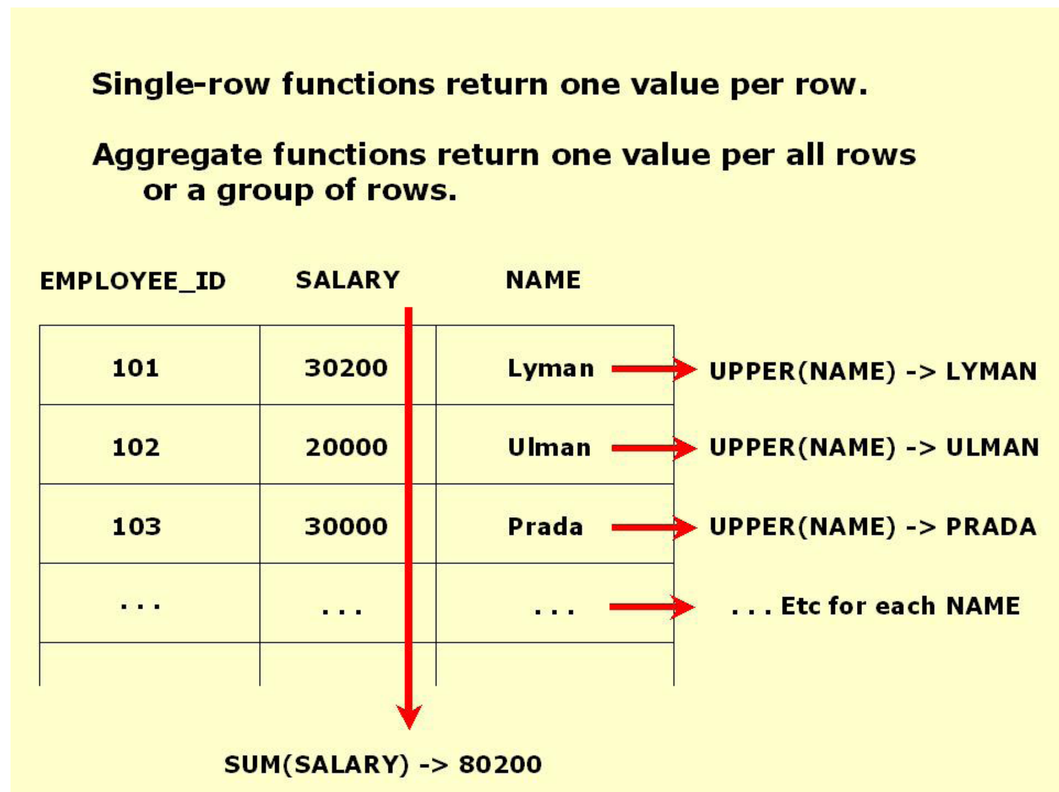
```
SELECT job_id, &&colname FROM job_history WHERE &&colname =  60 ;
Enter value for colname:department_id

(… the query result follows …)
```

4.0 Single-Row Functions

SQL **functions** operate on input values to return a result. They divide into two broad categories. **Single-row functions** operate on a single row and return a result, while **group functions** operate on a set of rows to return a result. Single-row functions are also called **scalar functions**, while group functions are also called **aggregate** or **multiple-row functions**.

Single-row functions return one value for each row in the result set. Aggregate functions return one value for a grouping of rows or the entire result set. For example, the UPPER function converts an input value to upper-case. It is a single-row function. Contrast this with the group function SUM, which can calculate the sum total for one column across all rows in the set.



All functions always return a single value of a predetermined data type. They return one result per execution.

Functions operate on three kinds of input data:

1. Character
2. Numeric
3. Date

Oracle's functions may be categorized as follows:

Function Group	**For**
Character functions (returning strings)	These functions take VARCHAR2 or CHAR argument(s) and return a character string. Examples: **UPPER** and **LTRIM**.
Character functions (returning numeric values)	Some character functions take a character argument but return a numeric value. Example: **LENGTH**.
Numeric functions	Return a single numeric value based on one or more arguments. Examples: **MOD** and **ROUND**.
Date and Time functions	Functions to manipulate date and time data, including date and time arithmetic. Examples: **ADD_MONTHS** and **NEXT_DAY**.
Conversion functions	These convert between standard data types. Examples: **TO_CHAR**, **TO_NUMBER**, and **TO_DATE**.
Group functions	Return a result per set of rows. Examples: **DISTINCT**, **SUM**, **COUNT**, **MAX**, and **MIN**.
Object reference functions	Manage **REF** values in **user-defined types** (UDTs). Examples: **DEREF** and **REFTOHEX**.
Analytical functions	Complex analytics.
Miscellaneous functions	A couple dozen functions that do not fit in any of the above categories. Examples: **USER**, **BFILENAME**, and **GREATEST**.

*The Exam does **not** cover Object Reference or Analytical functions.* It does cover all the others. This chapter summarizes the single-row functions, while the next chapter covers the conversion functions. Finally, chapter 6 details the aggregate or group functions.

The functions the exam covers are **built-in functions**. They come built-in as part of Oracle. Oracle also allows you to define your own functions, called **user-defined functions** or **UDFs**. UDFs are not covered on the exam.

4.1 Single-Row Functions

Scalar functions are probably used most often to reformat data. They may be encoded in SELECT lists, WHERE, and ORDER BY clauses.

They may be nested. That is, one function can process the output from another. In this example, the CONCAT function concatenates two literal character strings. The result is then converted to all upper-case by the UPPER function:

```
SQL> SELECT UPPER(CONCAT('Win ','the game!')) AS answer FROM dual;

ANSWER
-------------
WIN THE GAME!
```

When nested, the inner-most function executes first, and then its result is passed as input to the outer function. *If you encounter multiple levels of nested functions, all you have to do is resolve the inner-most one first, then the next inner-most one, and so on out to the outer-most function.* Each inner function, in turn, produces a single result that is then passed to any outer function that operates upon its result.

Single-row functions operate on input data of specific data types. If the input data they receive is not of the proper data type, Oracle attempts to convert the data to the proper data type, the **implicit conversion** concept we covered in chapter 3. This is as opposed to **explicit conversion**, data type conversions you direct by encoding any of the **conversion functions** discussed in the next chapter.

These two queries show an example of implicit conversion. The LENGTH function expects a character string input and returns the number of characters in the input string. If the input is either a numeric or date data type, Oracle first implicitly converts the input into a string type. Then the LENGTH function evaluates it. The first example query below feeds the LENGTH function a VARCHAR2 character value input. The second example also yields a valid result. But, note that the input to the LENGTH function was a salary, a NUMBER(8,2) data element with a value of **12000**, in this case:

```
SQL> SELECT LENGTH(last_name) FROM hr.employees WHERE employee_id=205;

LENGTH(LAST_NAME)
-----------------
          7

SQL> SELECT LENGTH(salary) FROM hr.employees WHERE employee_id=205;

LENGTH(SALARY)
--------------
          5
```

Here are Oracle's single-row functions:

Character Function	Returns
ASCII	ASCII equivalent for a character
CHR	The character for an ASCII value
CONCAT	Concatenates two strings
INITCAP	Capitalizes the first letter of each word in a string
INSTR	Numeric starting position of a substring
INSTRB	Same as INSTR but byte-count, not character count
LENGTH	String length in characters
LENGTHB	String length in bytes
LOWER	A string in lower-case
LPAD	Left-fills a string with a given fill character
LTRIM	Strips leading characters from a string
RPAD	Right-fills a string with a given character
RTRIM	Strips trailing characters from a string
REPLACE	Substring search and replace
SUBSTR	A substring (by numeric character position)
SUBSTRB	A substring (by numeric byte position)
SOUNDEX	A phonetic rendering of a string
TRANSLATE	Character search and replace
TRIM	Strips specified characters from string
UPPER	The string in all upper-case

Numeric Function	Returns
ABS	Absolute value
ACOS	Arc cosine
ASIN	Arc sine
ATAN	Arc tangent
ATAN2	Arc tangent via two inputs
BITAND	Bitwise AND on two inputs
CEIL	The next higher integer
COS	The cosine
COSH	The hyperbolic cosine
EXP	The base of natural log raised to a power
FLOOR	The next smaller integer
LN	The natural logarithm
LOG	The logarithm
MOD	The modulo (remainder) from a division operation
POWER	A number raised to a power
ROUND	A rounded number
SIGN	Sign indicator: positive, negative, or zero
SIN	The sine
SINH	The hyperbolic sine
SQRT	The square root
TAN	The tangent
TANH	The hyperbolic tangent
TRUNC	Truncates a number

Date Function	Returns
ADD_MONTHS	Adds months to a date
CURRENT_DATE	The current date
CURRENT_TIMESTAMP	Current date/time as a TIMESTAMP
DBTIMEZONE	The database's time zone
EXTRACT	A part of a date/time
FROM_TZ	A timestamp with time zone
LAST_DAY	The last day of the month
LOCALTIMESTAMP	Date/time in the session time zone
MONTHS_BETWEEN	Number of months between two dates
NEW_TIME	Date/time in new time zone
NEXT_DAY	The next day of the week beyond the given input
ROUND	Rounds a date/time
SESSIONTIMEZONE	The time zone of the session
SYS_EXTRACT_UTC	UTC for a timestamp in a time zone
SYSDATE	Current date/time
SYSTIMESTAMP	The current timestamp
TRUNC	Truncates a date to the degree specified
TZ_OFFSET	Offset from UTC for a time zone name

4.2 Examples of Single-Row Functions

Now we'll explore examples of the single-row functions that most commonly appear on the test. To simplify the examples, we've only provided the function clause. You can run and test any of these examples simply by constructing a query that refers to the dummy table DUAL. For example, to test the first example in the chart, just create a query containing the function clause like this:

```
SELECT  LENGTH(sysdate)  FROM  dual;
```

Explanations for why each example works the way it does are listed below the chart.

	Character Function Example	**Returns**
1.	length(sysdate)	9
2.	length('hi' || ' there')	8
3.	concat(1+2.14,' almost pi')	3.14 almost pi
4.	concat(concat('a','b'),'c')	abc
5.	substr('bob@aol.com',5,3)	aol
6.	substr('bob@aol.com',4)	@aol.com
7.	instr('bob@aol.com','@',1,1)	4
8.	instr('bob@aol.com','@')	4
9.	initcap('this is fun')	This Is Fun
10.	upper(lower('Why?'))	WHY?
11.	trim('@' from '@bob@aol.com@aol.com@@')	bob@aol.com@aol.com
12.	replace('this is fun','fun','bad')	this is bad

1. This example demonstrates nested functions. SYSDATE is a built-in Date function that returns the current date as a DATE data type. Oracle implicitly converts the DATE value to a character string in this default format: **29-APR-09**. This yields a character string length of 9.

2. This example shows how to concatenate two literal character strings using the concatenation operator (||). The result here is the string 'hi there' which contains 8 characters. (Note that the 2nd input argument contains a leading blank).

3. CONCAT takes two strings and concatenates them. If the arguments are not strings, it attempts to convert them prior to the concatenation operation. This example first resolves the numeric expression 1+2.14 and returns 3.14. It then implicitly converts this to the character string '3.14'. Finally, CONCAT splices this together with the string literal in the second input argument to produce the result shown.

4. This example shows function nesting. The inner-most CONCAT produces the string 'ab', while the outer function concatenates the letter 'c' to this to return 'abc.'

5 & 6. These statements illustrate SUBSTR. The first gives the subject string, then the starting charater position for the substring and its length. The second example shows that the implied length of the extracted substring, if not coded, is to the end of the subject string. **4** is the starting position for the substring to extract.

7 & 8. The first argument to INSTR is a subject string while the second is a search string to look for. The function returns the numeric position of the Nth occurrence of the search string it finds in the subject string, where Nth is specified by the 4th parameter (if any). The 3rd parameter is the position to start searching at in the subject string. In this case the first occurrence of the subject string @ is at character position four in the subject string. Example (8) just shows that the 3rd and 4th arguments default to 1 if not coded, so it returns the same result as example (7).

9. INITCAP capitalizes the first letter of every word in its input string.

10. The nested LOWER function converts the input string to lower case, then the outer UPPER function converts it to all upper-case letters.

11. TRIM can be used to trim off leading and/or trailing characters from a string. A common use is eliminating preceding and following spaces. Here it strips out occurrences of a specified character from both ends of a string. Related functions include LTRIM and RTRIM, which respectively remove leading and trailing occurrences of a specified character from a string. Conversely, the LPAD and RPAD functions left- and right- fill a string with occurrence(s) of a given character.

12. REPLACE replaces a target string within the subject string with a given replacement.

Here are example numeric and date functions. Full explanations follow the chart:

	Numeric and Date Function Examples	**Returns**
1.	round(13.48,1)	13.5
2.	trunc(13.42,1)	13.4
3.	mod(12,5)	2
4.	months_between('01-MAY-2009','01-JUL-2009')	-2
5.	months_between('01-JUL-2009','01-MAY-09')	2
6.	add_months('01-MAY-2009',2)	01-JUL-09
7.	next_day('01-MAY-2009','Friday')	08-MAY-09
8.	last_day('01-MAY-2009')	31-MAY-09

1. ROUND rounds off a number to the decimal precision indicated by the second input argument. If you do not code the 2nd input argument, ROUND will round off the number to the nearest whole number.

2. TRUNC truncates a number as per the decimal precision indicated by the second input argument. A whole number is returned if the 2nd argument is not coded.

3. MOD gives *the remainder* from dividing the 2nd argument into the 1st.

4 & 5. MONTHS_BETWEEN returns the number of months between the arguments. The ordering of the input arguments determines the sign of the result.

6. ADD_MONTHS adds the number of months indicated to the 1st argument.

7. NEXT_DAY returns the date on which the next day of the week specified occurs after the given input date.

8. LAST_DAY returns the last day of the month for the given month in date format.

5.0 Conversion Functions and Conditional Expressions

Recall that Oracle **implicitly converts** data types in input arguments to functions if required. This reduces syntax errors in your SQL statements. However, many times you'll want to **explicitly convert** data types using Oracle's conversion functions. This yields reliable data conversions for which you direct the result.

The kinds of implicit data type conversions Oracle will perform include: number to character, date to character, character to number, and character to date. Whether Oracle can successfully carry out implicit conversions depends on the value of the column and the data type required as the result.

The chart below lists Oracle's conversion functions. They are what you code when you want to explicitly direct conversions. While you should be familiar with what all the functions do, the exam questions concentrate on a small number of them. We'll give detailed examples of the most important ones following the chart.

Conversion Function:	**Usage:**
ASCIISTR	Converts characters to ASCII
BIN_TO_NUM	Converts bitstring to number
CAST	Converts datatypes
CHARTOROWID	Converts character to ROWID
COMPOSE	Converts to Unicode
CONVERT	Converts between character sets
DECOMPOSE	Decomposes a Unicode string
HEXTORAW	Casts a hex number to a raw
NUMTODSINTERVAL	Converts a number to interval day to second
NUMTOTMINTERVAL	Converts a number of interval year to month
RAWTOHEX	Casts a raw to hex
ROWIDTOCHAR	Casts a ROWID to character
TO_CHAR	Converts a date into a string
TO_DATE	Converts a string into a date
TO_DSINTERVAL	Converts a string to interval day to second

TO_MULTIBYTE	Converts single-byte to multibyte character
TO_NUMBER	Casts a numeric string to a number
TO_SINGLE_BYTE	Converts multibyte to single-byte character
TO_YMINTERVAL	Converts a string to interval year to month
UNISTR	Converts UCS2 Unicode

5.1 Key Conversion Functions

Three very useful conversion functions that illustrate how all the conversion functions work are TO_CHAR, TO_NUMBER, and TO_DATE. The exam concentrates on these.

TO_CHAR converts numbers or dates to character strings. It is often used to convert values for printing or display and has this general format:

```
TO_CHAR(number | date [,format] [,nls_parameter]),
```

The first parameter must either be a number or a date (or a value that can be implicitly converted into either a number or a date). The second parameter is an optional format specification that tells the function how to format the result it returns. Oracle supplies a long list of **format specifiers**—letters, digits, and special characters that provide masks that convert the data according to set rules. We can't list all the format specifiers here due to space concerns, and also because they are highly similar to those used in most programming languages (C++/C, Perl, COBOL, and others). See Oracle's *SQL Language Reference* manual for tables containing the format specifiers.

The third optional parameter refers to Oracle's **national language support** or **NLS** and can be safely ignored unless you are working with applications that provide cross-support for different spoken languages.

Here are the TO_DATE and TO_NUMBER templates:

```
TO_DATE(string [,format] [,nls_parameter]),
TO_NUMBER(string [,format] [,nls_parameter]),
```

Like TO_CHAR, TO_DATE and TO_NUMBER both have their own unique lists of format specifications. Oracle's *SQL Language Reference* manual lists them all.

Here are examples of these key functions:

Example:	Returns:	Note:
TO_CHAR(451,'0000999')	'0000451'	0 in format spec causes the display of leading zeroes
TO_CHAR(32.12,'99.9')	'32.1'	Last digit 2 is truncated by format spec precision
TO_CHAR(sysdate,'MON DDth, YYYY')	'MAY 15TH, 2009'	**DDth** specifier appends 'TH' to the result
TO_CHAR(sysdate, 'yyyy/mm/dd')	'2009/05/01'	Format spec formats the returned character string
TO_DATE('2009/05/01','yyyy/mm/dd')	01-MAY-09	2nd argument describes the format of the input
TO_DATE('20090501','yyyymmdd')	01-MAY-09	Dashes in output from default DATE format
TO_NUMBER('56','99')	56	As expected
TO_NUMBER('56','9999')	56	**9** in format spec means no leading zeroes
TO_NUMBER('56')	56	Only one argument is required

5.2 Handling NULL Values

What if you execute a series of nested functions, and one of them returns a NULL value? Special handling might be in order. Oracle provides four key functions for this purpose: NVL, NVL2, NULLIF, and COALESCE. Let's take a look at the template for each.

NVL(test_value, if_null)

NVL has two mandatory parameters of compatible data type. NVL tests the first argument. If it is NULL, NVL returns the value of its second argument IF_NULL. Otherwise it returns its first argument, TEST_VALUE.

NVL2(test_value, if_not_null, if_null)

NVL2 is just a variant of NVL. If the TEST_VALUE is NOT NULL, then NVL2 returns the 2nd argument. Otherwise it returns its 3rd parameter.

NULLIF(compare_term_1, compare_term_2)

NULLIF compares its two required parameters. If they are identical, then NULL is returned. If they are not identical then the 1st parameter is returned.

COALESCE(expr_1, expr_2, … expr_N)

COALESCE requires at least two parameters and may take many more if desired. It returns the first argument in its input list (read left to right), that is NOT NULL.

Here are examples of these key functions:

Example:	**Returns:**	**Reason:**
NVL(NULL,'ABC')	ABC	Returns the 2nd argument because the first evaluates to NULL
NVL(SUBSTR('A',5),'Bad')	Bad	Returns the 2nd argument because SUBSTR returns NULL since its *start* parameter is out of range for the given string
NVL2(NULL,'notN','N')	N	Returns the 3rd argument because the 1st argument evaluates to NULL
NVL2('1','no','yes')	no	Returns 2nd argument because the 1st argument evaluates to NOT NULL
NULLIF('abc','abc')	NULL	Returns NULL because the two arguments are identical
NULLIF('abc','xyz')	abc	Returns the 1st argument when the two arguments are not identical
COALESCE(NULL,'abc','def')	abc	Returns its 1st NOT NULL argument in the list

The chart above states that **NULLIF('abc','abc')** returns NULL. How can we verify this? (The result only shows up as a missing output on our display screen.) Here's one way:

```
SELECT  NVL(NULLIF('abc','abc'),'It was NULL!')  FROM dual;
```

This statement returns the string **It was NULL!**. This verifies that the first argument to the NVL function – our nested NULLIF function – returned NULL.

5.3 CASE Expressions and the DECODE Function

CASE expressions have two basic forms: the **simple CASE** and the **searched CASE**. First, here is an example of the simple CASE. It applies condition selection based on a series of WHEN clauses. This example figures out and indicates class membership of high school students based on the contents of the column CLASS_MEMBERSHIP. The **simple CASE**:

```
SELECT last_name, class_membership,
        CASE class_membership    WHEN 1  THEN 'Freshman'
                                 WHEN 2  THEN 'Sophomore'
                                 WHEN 3  THEN 'Junior'
                                 WHEN 4  THEN 'Senior'
                                  ELSE 'Unknown'  END  Classes
        FROM class_table ;
```

Here's an example of the **searched CASE**. In it, comparisons follow after the WHEN clause:

```
SELECT last_name, salary,
        CASE  WHEN  salary < 40000  THEN 'low'
                WHEN salary < 80000   THEN 'average'
                WHEN salary >= 80000 THEN 'high'  END Category
FROM employees    ORDER BY last_name ;
```

The DECODE function offers conditional logic, similar to CASE expressions, but coded into a function. DECODED requires *at least* three parameters, but can accept many more. Here is its template:

```
DECODE(expr1, comp1, if_true1 [, expr2, comp2, if_true2…]  [, if_false]
```

DECODE compares the 1st two input parameters. If they are equal, it returns the 3rd parameter, IF_TRUE1. If they are not equal, it proceeds to the next set of three parameters. If the first two of them are equal, it returns IF_TRUE2. If not, DECODE continues working its way through each set of three parameters, left to right, and returns the first IF_TRUEn value that matches two equal parameters.

If DECODE finds no parameter pairs that compare equally, it returns the optional parameter IF_FALSE. If IF_FALSE is not coded, it returns a NULL value.

Here's an example of the DECODE function directly from the Oracle manuals. It returns the name of one of four different cities depending on the value of the WAREHOUSE_ID. If WAREHOUSE_ID is not 1, 2, 3, or 4, then DECODE returns the string value 'Non domestic'.

```
SELECT product_id,
        DECODE (warehouse_id,      1, 'Southlake',
                                   2, 'San Francisco',
                                   3, 'New Jersey',
                                   4, 'Seattle',
                            'Non domestic') "Location"
FROM inventories
WHERE product_id < 1775
ORDER BY product_id, "Location";
```

6.0 Group Functions

Recall that **group** or **aggregate** functions return a single result for a set of rows. For example, SUM totals all values for a specified column and gives a single result. Group functions can also be applied to a group of rows that is some subset of the result set.

Group functions generally ignore NULL column values when calculating their results. Therefore developers sometimes code specific functions like NVL, NVL2, NULLIF, or COALESCE to handle them.

Whereas single-row functions can be nested way deeper than any practical use, group functions can only be nested two levels deep. The first two templates below are fine, but the third line is illegal and causes a syntax error:

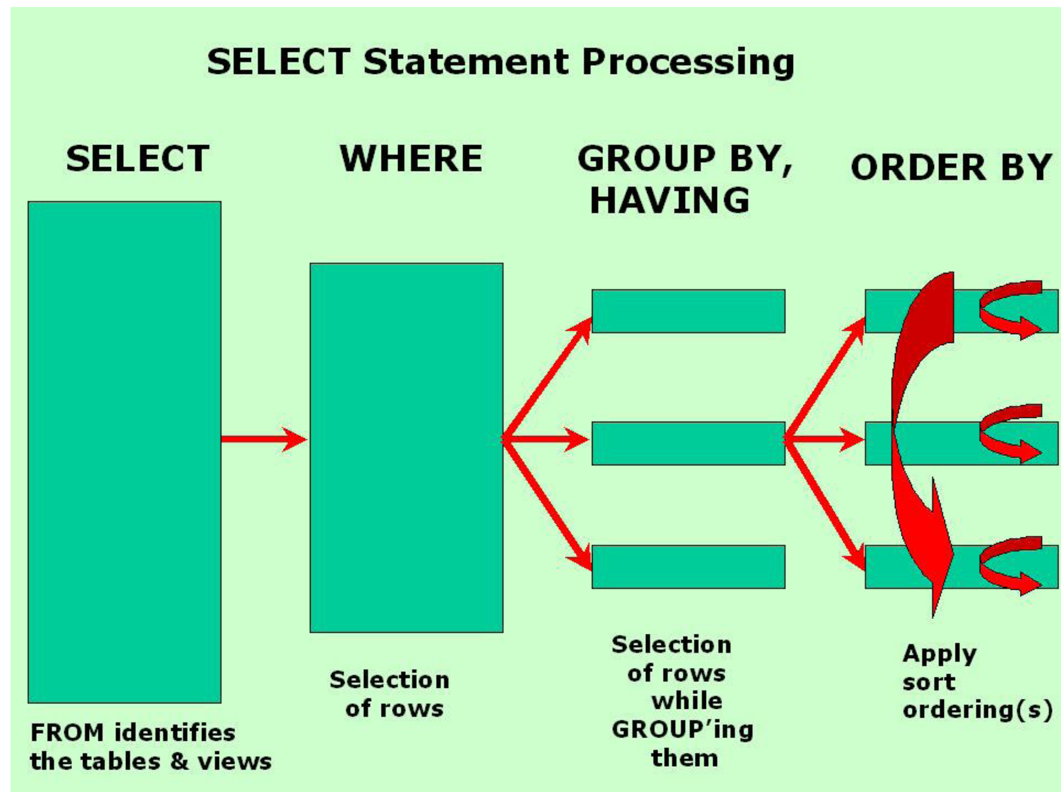
```
        GROUP_FUNC(group_column) = results ;
        GROUP_FUNC_1(GROUP_FUNC_2(group_column)) = results ;
NO!     GROUP_FUNC_1(GROUP_FUNC_2(GROUP_FUNC_3(group_column)))     NO!
```

When you have several keyword clauses in the same SELECT statement, the order in which they must occur is WHERE, GROUP BY and HAVING, ORDER BY. Here's the template:

```
SELECT {column | expr | group_function(column | expr) [alias] },…
        FROM table_name
        [WHERE condition(s)]
        [GROUP BY {column(s) | expr}]
        [HAVING group_condition(s)]
        [ORDER BY {column(s) | expr | num_pos}
                [ASC | DESC]  [NULLS FIRST | LAST]] ;
```

The order of the GROUP BY and HAVING keywords can be reversed – HAVING can come before GROUP BY. But both must occur *after* the WHERE clause and *before* the ORDER BY clause.

HAVING can only be coded if GROUP BY is present. It cannot be coded in the absence of a GROUP BY clause. The diagram below shows the parts the SELECT statement play in processing the query:



6.1 GROUP BY Clause

Several SELECT keywords are useful with group functions. The GROUP BY clause allows grouping data by specific column(s) or expression(s). It is often used to calculate statistics from a result set divided by different attributes. The column(s) driving the grouping is called the **grouping attribute**. The key rule to remember is this: *any item in the SELECT list that is not a group function must be a grouping attribute of the GROUP BY clause.* Otherwise a syntax error results. Here's an example of the GROUP BY clause that displays a count of the number of employees in each department, sorted by department:

```
SELECT COUNT(*), department_id
        FROM employees
        GROUP BY department_id
        ORDER BY department_id ;
```

Next, an example that violates the rule that all items in the SELECT list must be group functions or grouping attributes. Adding the LAST_NAME column violates the rule:

```
SQL> SELECT COUNT(*), department_id, last_name
        2       FROM employees
        3       GROUP BY department_id
        4       ORDER BY department_id ;
SELECT COUNT(*), department_id, last_name
                                *
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

6.2 The HAVING Clause

What if you want to restrict the rows within a group when using a GROUP BY clause? The WHERE clause will not work. It restricts entire results sets before GROUP BY is applied. So, SQL provides the HAVING clause. HAVING restricts group-level results in the same manner that WHERE restricts row-level results. It is therefore not allowable–and would not make sense–to code HAVING without a GROUP BY clause.

The following example statement restricts the GROUP BY results with the HAVING clause. It reports departments that have total salaries exceeding $100,000, and it shows what the total salary is for those departments:

```
SELECT department_id, SUM (salary) FROM employees
        GROUP BY department_id  HAVING SUM (salary) > 100000 ;
```

6.3 The GROUP Functions

Here is a list of Oracle's group functions. You won't need to be an expert on all of these, although you should be able to recognize them. After the chart we'll give examples of how to use the functions on which the exam focuses.

Group Function:	**Returns:**
AVG	Statistical mean
CORR	Coefficient of correlation of number pairs
COUNT	Number of non-NULL rows
COVAR_POP	Population covariance of number pairs
COVAR_SAMP	Sample covariance of number pairs
CUME_DIST	Cumulative distribution of values within groupings
DENSE_RANK	Ranking of rows in group (no skipping on ties)
FIRST	Modifies other aggregate functions to return expressions based on ordering of the second column expression
GROUP_ID	Group identifier to uniquely ID duplicate groups
GROUPING	0 for non-summary rows, 1 for summary rows
KEEP	Modifies other aggregate functions to return first or last value in a grouping
LAST	Modifies other aggregate functions to return expressions based on ordering of the secondary column expression
MAX	Largest value
MIN	Smallest value
PERCENT_RANK	Percentile ranking of the specified value
PERCENTILE_ CONT	The interpolated value that would fall in the percentile position using a continuous model
PERCENTILE_DISC	The interpolated value that would fall in the percentile position using a discrete model
RANK	Ranking of rows within an ordered group (skipping ties)

REGR_AVGX	Average x value in non-NULL (y,x) pairs
REGR_AVGY	Average y value in non-NULL (y,x) pairs
REGR_COUNT	Number of non-NULL (y,x) pairs
REGR_INTERCEPT	Linear regression of y intercept
REGR_R2	The linear regression coefficient of determination
REGR_SLOPE	The linear regression slope
REGR_SXX	Sum of the squares of the independent variable expression
REGR_SXY	Sum of the products of the independent variable expression and the dependent variable expression
REGR_SYY	Sum of the squares of the dependent variable expression
STDDEV	Standard deviation
STDDEV_POP	Population standard deviation
STDDEV_SAMP	Sample standard deviation
SUM	Sum of all values
VAR_POP	Population variance
VAR_SAMP	Sample variance
VARIANCE	Sample variance, or 1 for sample size of 1

6.4 Key GROUP Functions

Here are the templates of the group functions the exam usually asks about:

```
COUNT({ * | [DISTINCT | ALL] expr}) ;
AVG( [DISTINCT | ALL] expr) ;
SUM( [DISTINCT | ALL] expr) ;
MAX( [DISTINCT | ALL] expr) ;
MIN( [DISTINCT | ALL] expr) ;
VARIANCE( [DISTINCT | ALL] expr) ;
STDDEV( [DISTINCT | ALL] expr) ;
```

When DISTINCT is coded, only unique occurrences of the expression denoted by EXPR are included in the evaluation for each group. For example, COUNT DISTINCT will only count unique occurrences of EXPR for each group. ALL is the default, so by default all values are included in the group function. This includes any duplicates. Remember that, unless you specifically code to handle them through functions like NVL or NVL2, nulls are ignored in group function evaluations.

Here are a few usage examples of these group functions. Start with a simple COUNT of all rows in the EMPLOYEES table. Duplicates would be included, since we did not specify DISTINCT and the default for all the above group functions is ALL:

```
SQL> SELECT COUNT(*)  "Total Number"  FROM hr.employees;

Total Number
------------
       107
```

How many managers are there? Here we need unique MANAGER_ID's only:

```
SQL> SELECT COUNT(DISTINCT manager_id) "Managers" FROM hr.employees;

Managers
----------
18
```

What are the standard deviation and variance on salaries at the company?

```
SQL> SELECT STDDEV(salary) as "Stddev", VARIANCE(salary) as "Variance"
  2  FROM hr.employees;

Stddev     Variance
---------- ----------
3909.36575 15283140.5
```

What are the standard deviation and variance on the salaries within each department? This is a little trickier query because you want to apply the STDDEV and VARIANCE functions to all salaries within each department—not to all rows in the table as a whole. You'll want to group employees by department, since that is the level on which we want to apply the standard deviation and variance group functions. We'll add an ORDER BY clause to sort by department to make it easier to view the results:

```
SQL> SELECT department_id, STDDEV(salary) as "Stddev", VARIANCE(salary) as "Variance"
          2  FROM hr.employees
          3  GROUP BY department_id
          4  ORDER BY department_id;
```

```
DEPARTMENT_ID        Stddev      Variance
-------------      ----------   ----------
     10                    0            0
     20           4949.74747     24500000
     30           3362.58829     11307000
     40                    0            0
     50           1488.00592   2214161.62
     60           1925.61678      3708000
     70                    0            0
     80            2033.6847   4135873.44
     90           4041.45188   16333333.3
    100           1801.11077      3244000
    110           2616.29509      6845000
                           0            0
```

We've got the right grouping and group function outputs. But what about those departments that report 0 for the group functions? And how about that last row, where there is no reported DEPARTMENT_ID?

A quick query applying the COUNT function on each department shows that there is only one employee in each of the departments that returned 0 for standard deviation and variance. Those functions cannot compute meaningful numbers with only one value, so the 0 result makes sense:

```
SQL> SELECT department_id, COUNT(*)
2  FROM hr.employees
3  GROUP BY department_id
4  ORDER BY department_id ;

DEPARTMENT_ID   COUNT(*)
-------------  ----------
     10             1
     20             2
     30             6
     40             1
     50            45
     60             5
     70             1
     80            34
     90             3
    100             6
    110             2
                    1
```

And the missing DEPARTMENT_ID is NULL:

```
SELECT COUNT(*) FROM hr.employees WHERE department_id IS NULL;

COUNT(*)
----------
     1
```

So all we have to do to develop the final query is add a HAVING clause to eliminate any department that has only one employee. This clause also knocks out the row with the NULL DEPARTMENT_ID:

```
SQL> SELECT department_id, STDDEV(salary) as "Stddev", VARIANCE(salary) as "Variance"
2  FROM hr.employees
3  GROUP BY department_id
4  HAVING COUNT(*) > 1
5  ORDER BY department_id;

DEPARTMENT_ID        Stddev        Variance
-------------        ----------    ----------
      20             4949.74747      24500000
      30             3362.58829      11307000
      50             1488.00592    2214161.62
      60             1925.61678       3708000
      80              2033.6847    4135873.44
      90             4041.45188    16333333.3
     100             1801.11077        244000
     110             2616.29509       6845000
```

Now here's an example that shows ordering query results on more than one column. In this case DEPARTMENT_ID is the major sort order and EMPLOYEE_ID the minor (this could be useful for exception processing if accidentally there were more than one employee per EMPLOYEE_ID). Remember that to list EMPLOYEE_ID in the SELECT list for the output you have to include it in the GROUP BY clause. Failure to do so causes a syntax error.

```
SQL> SELECT department_id, employee_id
2  FROM hr.employees
3  GROUP BY department_id, employee_id
4  ORDER BY department_id, employee_id ;

DEPARTMENT_ID    EMPLOYEE_ID
-------------    -----------
      10             200
      20             201
      20             202
      30             114
      30             115          (continued…)
```

7.0 Joining Data Across Tables

Joins combine data from more than one table (or view) into the result set. Columns on which table(s) are joined must be of **compatible** data type (either identical or convertible). Data type conversions typically cause indexes not to be used in the join, which reduces performance. The minimum number of joins required for a set of tables is the number of tables minus 1. So to join 5 tables you need a minimum of 4 joins. To join 2 tables you need one join.

Oracle supports two entirely different syntaxes for joins. Oracle's own proprietary or "traditional" syntax is the most widely used, though it has been superseded by a newer syntax that fully conforms to the American National Standards Institute (ANSI) 1999 syntax. We'll start with the ANSI-1999 syntax and then proceed to the traditional syntax. You must know both for the exam.

7.1 ANSI Joins

Here are the basic operations and syntax for ANSI syntax join clauses. These are all for **equijoins**, meaning that tables will be joined on the basis of matching column values:

```
table_name   NATURAL   [INNER]   JOIN   table_name ;
table_name   [INNER]   JOIN   table_name  USING  column(s) ;
table_name   [INNER]   JOIN   table_name  ON  condition ;
```

Let's make this clearer with some examples.

The **natural join** means the join occurs based on all columns having the same name in both tables:

```
SELECT location_id, city, department_id
        FROM locations NATURAL JOIN departments ;
```

If you want to specify the columns that should be used for a join, code the JOIN…USING syntax. This allows you to exclude any identically-named columns you don't want participating in the join:

```
SELECT location_id, city, department_name
        FROM locations JOIN departments
        USING (location_id) ;
```

Regular joins are sometimes called **inner joins**. Only rows that match on join columns are returned. **Outer joins** return some rows with non-matching columns as well. Why might we do this? An example would be where you join EMPLOYEES and DEPARTMENTS tables. An outer join would allow you to retrieve employees who don't have departments (employees with NULL department values). You could see how this might be useful in exception processing.

The ANSI syntax uses the phrases LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL OUTER JOIN, instead of the traditional Oracle "+" symbol syntax for outer joins. LEFT OUTER JOIN (or LEFT JOIN) returns all matching rows, plus the unmatched rows from the table to the left of the join clause. Here's an example from Oracle's *SQL Language Reference* manual. It lists rows that match DEPARTMENT_IDs and also rows from the DEPARTMENTS table that have no match on the key in the EMPLOYEES table:

```
SELECT d.department_id, e.last_name
        FROM departments d LEFT OUTER JOIN employees e
        ON d.department_id = e.department_id
        ORDER BY d.department_id;
```

Remember that a FULL OUTER JOIN returns rows based on the matching condition, plus unmatched rows from the tables both to the left and right of the join clause. So this query returns all matched rows plus unmatched rows from both the tables:

```
SELECT d.department_id as d_dept_id, e.department_id as e_dept_id, e.last_name
        FROM departments d FULL OUTER JOIN employees e
        ON d.department_id = e.department_id
        ORDER BY d.department_id;
```

Note the use of **table aliases** in these two queries. A table alias is just a short abbreviation for the full table name to make it easier to write queries. The FROM clause assigns the table aliases to the two join tables. The table aliases are used as shorthand to quality columns. When column names are identical in the two tables you're joining, you need to make those column references **unambiguous** in the code—you have to tell Oracle which table any column that you are referring to belongs to.

Nonequijoins match rows from different tables based on an inequality between column values, rather than on equality like equijoins. Nonequijoins use the JOIN…ON syntax and are not very common.

Finally, the **cross join** or **Cartesian product** produces a result set with every possible combination of source and target tables joined together. The output is potentially huge and performance terrible—because multiplying the total number of rows for the two tables together tells you how many rows you'll get in the cross join results set. For example, with a 100,000 row table and a 200,000 row table, you'll get (100,000 * 200,000) = 20,000,000,000 rows. Wow! The ANSI keywords CROSS JOIN perform the Cartesian product.

Here's the template for a cross join using ANSI-1999 syntax:

```
SELECT  table_name_1.column,  table_name_2.column
        FROM  table_name_1
        CROSS JOIN table_name_2 ;
```

This example CROSS JOIN displays matching department number combinations and tries to limit results by adding a WHERE clause:

```
SELECT d.department_id, e.department_id, e.last_name
        FROM  departments d  CROSS JOIN  employees e
        WHERE d.department_id < 200
        ORDER BY d.department_id ;
```

7.2 Oracle Syntax Joins

The most common join is the **equijoin**, where tables are joined based simply on equality of values in designated columns (using the equality operators). Here is a simple equijoin linking the EMPLOYEES and DEPARTMENTS tables on the DEPARTMENT_ID:

```
SELECT a.department_id, b.last_name
        FROM departments a, employees b
        WHERE a.department_id = b.department_id ;
```

Outer joins include "normal" join results (inner join results) plus rows present in one table but not the other. An example is where a department exists but has no employees. Place the outer join operator **(+)** at the end of the name of the table deficient in data. Example:

```
SELECT a.department_name, b.employee_id, b.last_name
        FROM departments a, employees b
        WHERE a.department_id = b.department_id(+)
        ORDER by a.department_id ;
```

Self-joins relate a table to itself. They are useful in Bill Of Materials (BOM) and routing problems. They're sometimes referred to as **recursive joins**. This example returns a list of all employees whose salaries are more than double that of some other employee in the same department:

```
SELECT  a.department_id, a.last_name, a.salary, b.last_name, b.salary
        FROM employees a, employees b
        WHERE a.department_id = b.department_id
        AND a.salary > 2 * b.salary
        ORDER BY a.department_id, b.last_name ;
```

Hierarchical joins express hierarchical relationships (such as parent/child or part/sub-part relationships). They typically involve self-joins. START WITH identifies the root row of the hierarchy, CONNECT BY expresses the hierarchical relationship (identifying the parent by the PRIOR keyword), and the WHERE clause can limit the returned rows. Here's an example:

```
SELECT  last_name, employee_id, manager_id, job_id
        FROM employees
        START WITH job_id = 'AD_PRES'
        CONNECT BY PRIOR employee_id = manager_id ;
```

This query processes through the organizational reporting structure, starting at the top with the President (whose JOB_ID is **AD_PRES**). The query gives a list of employees that reflect the management reporting structure of the company. The connection between EMPLOYEE_ID and MANAGER_ID drives the query. Here is sample output:

```
LAST_NAME              EMPLOYEE_ID   MANAGER_ID     JOB_ID
---------------------  -----------   ----------     ----------
King                   100                           AD_PRES
Kochhar                101             100           AD_VP
Greenberg              108             101           FI_MGR
Faviet                 109             108           FI_ACCOUNT
Chen                   110             108           FI_ACCOUNT
Sciarra                111             108           FI_ACCOUNT
Urman                  112             108           FI_ACCOUNT
Popp                   113             108           FI_ACCOUNT
```

(…continued…)

8.0 Subqueries

A **subquery** is a nested query—a nested SELECT clause. The purpose of a subquery is to find and return data to a higher-level or enclosing SQL statement. Subqueries can be nested inside SELECT, UPDATE, INSERT, and DELETE statements.

This example SELECT statement lists employees who have higher than average salaries in the company. See how the indented or inner query calculates the average salary across the company, and that it returns that value to the outer query for purposes of the comparison:

```
SELECT last_name, salary FROM employees
        WHERE salary >
        (SELECT AVG(salary) FROM employees) ;
```

The subquery is termed the **inner query**, while the statement to which it returns its results is called the **outer query**. The inner query is resolved first, and its results are returned to the outer query for additional processing.

Since you can nest subqueries within subqueries, it sometimes makes sense to refer to the inner-most query and the outer-most query. When you have a complicated SQL statement with multiple nested subqueries, you can easily resolve it. Always remember this rule: however many nested subqueries a statement contains, *subqueries are always resolved from inner-most to outer-most.*

Single-row subqueries return return one row to their enclosing SQL statement. A **scalar subquery** is a special case of a single-row subquery; it returns one value (a single row with a single column). **Mutiple-row subqueries** return a multiple row result set. **Correlated subqueries** run the inner query once for each row in the table or view named in the outer SELECT statement.

Subqueries may be encoded in the:

- SELECT list used for column projection
- FROM clause
- WHERE clause
- HAVING clause

Subqueries are often coded to return from zero to many rows for use with keywords like **IN, NOT IN, ANY,** or **ALL**. Or, subqueries can return a single value for use with one of the comparison operators (eg: equals **=**).

8.1 Kinds of Problems Subqueries Can Solve

It is important to understand the different kinds of problems that subqueries are used to solve.

One typical use is where the subquery returns value(s) for comparison purposes. The example give above fits these criteria. The inner query returns value(s) that will be compared in the outer query. In the example below, the subquery determines the average percentage commission for all employees, then the outer query uses that single value for its WHERE clause. The example lists employees who make larger than average commissions:

```
SELECT employee_id, last_name, commission_pct  FROM employees
        WHERE commission_pct >
        (SELECT AVG(commission_pct) FROM employees) ;
```

Another use of subqueries is to generate data for DML statements (UPDATE, INSERT, and DELETE). Here the subquery is coded as a comparand in the WHERE clause of the driving data update statement. Here are two examples. These queries are useful because you may not know how many employees are in the lowest-number department:

```
/* Delete employees whose department is the lowest in the DEPARTMENTS table */
DELETE FROM employees
        WHERE department_id =
        (SELECT MIN(department_id) FROM departments) ;

/* Raise employees salaries by 10% for those whose department number is lowest */
UPDATE employees  SET salary = salary + (salary * 0.10)
        WHERE department_id =
                (SELECT MIN(department_id) FROM departments) ;
```

Note that you cannot encode a subquery in the INSERT statement's VALUES clause.

Subqueries can be used to generate a table from which to SELECT. This usage is called an **inline view.** It is where you code the subquery in the FROM clause of the outer query. The goal is to simply a complicated SQL statement by removing JOIN logic and condensing the query. Since inline views are tough to grasp, look at these two equivalent statements. They both return the same result, the number of employees in the employees table. But the second statement does it by an inline view:

```
SELECT  COUNT(DISTINCT employee_id)  FROM  employees ;

SELECT COUNT(*)
        FROM (SELECT DISTINCT employee_id FROM employees) ;
```

Here's a more complicated and useful inline view:

```
SELECT employee_id, salary, ROUND(salary / tot_salary * 100,0) percent
    FROM employees,
        (SELECT SUM(salary) tot_salary FROM employees) ;
```

What does this query do? Remember, the key is to resolve the inner-most query first. Determine what it returns to the outer query, and you can decode the statement. Here, the inner query is calculating the total salary for all employees, which it labels **tot_salary**. The outer query uses this number to calculate what percentage each employee's salary is of the total amount this company pays in salaries.

Another common use of subqueries is to return a results set to which the outer query applies one of the keywords IN, NOT IN, ALL, ANY, or SOME. This query lists information for all employees who work at departmental locations **1500** or **1600**:

```
SELECT department_id, last_name, employee_id
    FROM employees
    WHERE department_id IN
        (SELECT department_id FROM departments
        WHERE location_id = 1500 OR location_id = 1400)
    ORDER BY department_id, last_name ;
```

Star transformation is another use for subqueries. This example reports from data warehouse tables that are designed in the "star formation" popular in data warehouse applications. The example reports from the large sales table and uses subqueries to determine which information to list:

```
SELECT order_number, purchaser_number FROM sales
    WHERE prod_code IN
        (SELECT prod_code FROM products WHERE product = 'harness')
    AND offer_num IN
        (SELECT offer_num FROM offers WHERE special_offer = 'TV') ;
```

Yet another use for subqueries is to project columns that they will populate. In this usage the subquery is encoded as part of the SELECT list itself. This example figures out what the maximum commission paid would be if the highest salaried employee also had the highest commission rate:

```
SELECT (SELECT MAX(salary) FROM employees) *
    (SELECT MAX(commission_pct) FROM employees) / 100
    AS "MAX COMM RATE"
FROM dual;
```

Finally, subqueries can be used in correlated subselects. *A **correlated subquery** executes the inner SELECT one time for each row in the table or view named in the outer SELECT.* So the sub-SELECT is reevaluated for each row retrieved by the outer SELECT.

This example lists the employees in each department whose salaries are greater than that of the average salary for their department;

```
SELECT department_id, employee_id, salary FROM employees s
    WHERE salary >
        (SELECT AVG(salary) FROM employees
        WHERE department_id = s.department_id)
    ORDER BY department_id, employee_id ;
```

In the correlated subquery, the outer SELECT is assigned a table label or **correlation name**. In this example it is the letter **s**. The column name S.DEPARTMENT_ID is then used to correlate the columns of the outer-level SELECT with those of the sub-SELECT.

Correlated subqueries are sometimes easier to write than alternative ways of solving the same problem. But since the inner query repeatedly executes for each row of the outer query, they typically perform poorly.

9.0 Set Operators

Set Operators combine the results sets from two or more SELECT statements.
This is called a **compound query**. Here are the set operators for Oracle SQL:

Set Operator:	**Use:**
UNION	Returns all rows from both queries, no duplicates, sorted.
UNION ALL	Returns all rows from both queries, including duplicates. Unsorted by default.
INTERSECT	Returns distinct rows returned by both queries, sorted.
MINUS	Returns distinct rows returned by the first query but not by the second query, sorted.

By default, sorting occurs across all the columns, as listed left to right. The exception is UNION ALL which does not sort at all by default. You would therefore normally code an ORDER BY clause for a UNION ALL operation.

The order in which tables are coded in SELECT statements affects the results for MINUS but not for the other set operations. There is no order of precedence among the set operators themselves, so if you code more than one set operator in a query, parentheses are recommended to dictate the order of resolution.

Here are some set operator examples that show how the commands work**.** Assume table **One** contains rows A, B, C, and table **Two** contains rows A and B:

Example:	**Result:**
One UNION Two	A, B, C
One UNION ALL Two	A, A, B, B, C (in any order)
One INTERSECT Two	A, B
One MINUS Two	C
Two MINUS One	NULL set

In writing the queries, it is fine if different column names are used in the corresponding columns for the tables involved. But the data types of the corresponding columns must be of the same data type group. That is, Oracle will accept two different numeric types for a corresponding column across tables, but not a numeric type and a character type, for example. The number of selected columns must match across the components of the compound query.

9.1 Example Set Operator Queries

In Oracle's demonstration HR schema, the HR.JOB_HISTORY table has ten rows of data:

SQL> select * from job_history;

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	DEPARTMENT_ID
-----------	---------	---------	----------	-------------
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

First we'll create a duplicate of the JOB_HISTORY table in our own default schema:

```
CREATE TABLE job_history AS SELECT * FROM hr.job_history;
```

Both the new JOB_HISTORY and HR.JOB_HISTORY tables now have the exact same table definitions and the same ten data rows shown above. While it's unlikely you'll often perform set operations on duplicate tables, doing so is very instructive in showing how set operations work.

First, perform a UNION of the two tables:

```
SELECT * FROM job_history UNION SELECT * FROM hr.job_history;
```

The result is the same ten rows seen above, but sorted. Since UNION eliminates duplicates, we get ten rows back (instead of twenty). The rows are sorted by the columns as listed, from left-most on to the right. So the output is sorted by EMPLOYEE_ID, then by START_DATE within EMPLOYEE_ID, etc. Here is that result set:

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	DEPARTMENT_ID
-----------	---------	---------	----------	-------------
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
102	13-JAN-93	24-JUL-98	IT_PROG	60
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	17-SEP-87	17-JUN-93	AD_ASST	90
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90
201	17-FEB-96	19-DEC-99	MK_REP	20

Now, try a UNION ALL with the two identical tables:

```
SELECT * FROM job_history UNION ALL SELECT * FROM hr.job_history;
```

Since UNION ALL does not eliminate duplicates, we get twenty rows in the result set (two of each row). The rows are in no particular order, as UNION ALL does not sort by default. We could add an ORDER BY clause if sorted output is important (as it usually is).

INTERSECT reports the rows that occur in both tables without duplicates:

```
SELECT * FROM  job_history INTERSECT SELECT * FROM hr.job_history;
```

Thus this reports back ten rows. They are sorted by columns, left-most to right-most. In this particular case, where we are operating on two duplicate tables, the output happens to be the exact same we received from the initial UNION query. (Of course, since most compound queries don't work with duplicate tables, this is an unusual result.)

Finally, let's try a MINUS operation. MINUS shows the difference between two tables (what rows are in the first table that are not in the second). Since we are operating on duplicate tables, this query returns no rows:

```
SELECT * FROM  job_history  MINUS  SELECT * FROM  hr.job_history;
```

Usually MINUS gives entirely different results if you switch the positioning of the tables in the query. "Table A MINUS table B" does not equal "table B MINUS table A." But in this specific case, where the tables JOB_HISTORY and HR.JOB_HISTORY contain exact duplicates rows, swapping the table positions in the query will also return no rows:

```
SELECT * FROM  hr.job_history  MINUS  SELECT * FROM  job_history;
```

10.0 Updating Data Through DML

SQL statements are often divided into:

- **Data Manipulation Language** or **DML** statements that work with data
- **Data Definition Language** or **DDL** statements that define the underlying objects like tables, indexes, views, and tablespaces
- **Data Control Language** or **DCL** statements that manage security
- **Transaction Control Language** or **TCL** statements to control transactions

Oracle's DML statements are SELECT, INSERT, UPDATE, DELETE, and MERGE. SELECT is officially a DML statement, though in informal usage, most Oracle professionals wouldn't include it because it does not update data; the other four DML statements are used to add or change data in the database. DDL statements include CREATE, ALTER, DROP, RENAME, TRUNCATE, and COMMENT. These manage tables and other database objects. DCL statements include GRANT and REVOKE for security, while the TCL statements include COMMIT, ROLLBACK, and SAVEPOINT for transactional control.

10.1 INSERT

Use an INSERT statement with a VALUES clause to insert a row into a table. You only need to specify a **column list** (field names) if you do not supply a value for every column.

You may use an INSERT with a subquery to insert multiple rows. Subqueries must return rows that match the number of columns in the target table's column list and have **compatible** (ie, convertible) data types. This example inserts (potentially) many rows into the EMPLOYEES table as read from the HUMAN_TAB table:

```
INSERT INTO employees  SELECT * FROM human_tab ;
```

When inserting into a view, the view should have been defined with the WITH CHECK OPTION to ensure inserted rows must meet the criteria of the view's defining query.

As with UPDATE, the data INSERT statements attempt to place into a table must meet any **constraints** on that table. Constraints are business rules you apply to a table when you create the table or alter its definition. The next chapter explains everything about constraints.

When coding the INSERT statement inside a program, use the RETURNING clause to return inserted values to variables (including ROWIDs). Use PARTITION to insert data into a specific table partition. Both clauses also apply to the UPDATE statement.

Note that it is often faster to "bulk load" tables through Oracle utility programs like SQL*Loader or Datapump than it is to insert rows by the SQL INSERT statement.

10.2 UPDATE and DELETE

UPDATE and DELETE statements can affect zero, one, or many rows of data in one statement. Just like the INSERT statement, UPDATE and DELETE operations must respect any table constraints to succeed.

Sometimes an UPDATE or DELETE will target one row, based on a primary key given in the statement's WHERE clause. Other times the WHERE clause will be less restrictive and many rows will be updated or deleted by one SQL statement.

UPDATE or DELETE statements coded without a WHERE clause update or delete *all rows* in the table. This is called an **unqualified** UPDATE or DELETE. Unqualified UPDATEs and DELETEs are very powerful and should not be issued in error!

10.3 MERGE

The MERGE statement both updates and inserts rows into a table. Rows that have matching keys are updated; other (new) rows are inserted. The MERGE statement clauses WHEN MATCHED and WHEN NOT MATCHED are used to indicate whether rows are updated or inserted, depending on the match criteria.

10.4 Example INSERT, UPDATE, DELETE, and MERGE Statements

Here are examples of DML statements that update data:

```
INSERT INTO emp VALUES('BOB',19836,32000,1,'295','HIRED') ;
INSERT INTO emp (ename, emp_id) VALUES ('BILL', 29283) ;
INSERT INTO emp  SELECT * FROM human_tab WHERE id IN (17,22,113) ;
UPDATE emp SET salary = salary * 1.2  WHERE deptno = 29 ;
DELETE FROM emp WHERE emp_status = 'FIRED' ;
DELETE emp WHERE emp_status = 'FIRED' ;        -- FROM is not required
DELETE emp ;                                   -- Deletes all table rows !
```

10.5 TRUNCATE

An alternative to the DELETE statement to delete all rows from a table is the TRUNCATE command. TRUNCATE is a DDL statement that removes all data from a table—as opposed to DELETE, which is DML. TRUNCATE is fast, leaves the table definition intact (including dependent constraints and indexes), and reclaims space. But, since it's DDL, you can **not** issue a ROLLBACK to recover from an errant TRUNCATE. In contrast, since the DELETE statement is DML, DELETE logs changes and offers the chance to ROLLBACK data until it is committed to the database. Here's the format for TRUNCATE:

```
TRUNCATE TABLE  table_name ;
```

If you need to very quickly delete all the rows from a large table, yet keep the table definition in the data dictionary, TRUNCATE is your answer. Just remember that TRUNCATE is a DDL statement, so the change will be immediately auto-committed—there is no logging and rolling back the statement if you deleted the rows by mistake.

10.6 Transactions

A **transaction** is a set of one or more SQL DML statements, all of which will either succeed (and are applied to the database), or do not succeed (in which case none of the statements in the transaction are applied to the database). The purpose of dividing data updates into transactions is to ensure the integrity of the data.

A log-in session starts a transaction as soon as it issues any INSERT, UPDATE, or DELETE statement. Whatever changes you make in that session are invisible to other database users or programs until you COMMIT them to the database.

A transaction is ended **explicitly** by the user when he issues a SQL **COMMIT** or **ROLLBACK** statement. Or it may be ended **implicitly**, by external events like ending the client session or the database going down. *Issuing a DDL or DCL statement implicitly ends a transaction with a COMMIT.* So if you issue a series of DML statements, then follow them by a DDL or DCL statement, all the preceeding DML changes are committed to the database.

A **commit** makes pending data changes permanent to the database. A **rollback** ensures any pending changes not yet committed are discarded and not applied to the database. You can issue explicit COMMIT or ROLLBACK commands yourself to control transactions.

Remember that while DDL and DCL commands issue an implicit commit, DML commands (SELECT, INSERT, UPDATE, DELETE) do **not** cause an implicit commit. They are thus eligible for rollback until a COMMIT statement is issued.

For example, say you perform these actions in order:

1. Create a new table
2. INSERT 2 rows into that table
3. GRANT the SELECT privilege to PUBLIC on that table

If the database crashes after step (3), all 3 actions were successfully applied to the database because of the implicit commit that occurs after any DCL statement (the GRANT). If the database were to crash after step (2), you have a new table due to the implicit commit after the DDL statement (CREATE TABLE) in step (1), but no rows are in it because the INSERT statement is DML and is not implicitly committed.

10.6.1 SAVEPOINTS

Oracle supports **savepoints**, intermediate points within transactions. You can rollback only part of the transaction by issuing a ROLLBACK TO SAVEPOINT statement. Savepoints provide a convenient way to break up large transactions into individually recoverable pieces.

To create a savepoint, simply issue this command:

```
SAVEPOINT savepoint_name;
```

To later roll back to this savepoint, issue:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

To commit the work in the savepoint permanently to the database, issue the standard COMMIT command:

```
COMMIT ;
```

10.6.2 Transactions in Developer Tools

Most developer tools–including SQL*Plus and SQL Developer–by default follow the standard SQL behavior. They start a transaction implicitly when you issue a DML statement, and the transaction ends when you issue a COMMIT or ROLLBACK (or if you exit the tool or if your session abnormally terminates).

You can change this behavior so that a COMMIT or ROLLBACK automatically occurs *after every DML statement*. This state is called **autocommit**. In SQL*Plus set autocommit on or off by issuing this statement:

```
SET AUTOCOMMIT  ON  |  OFF ;
```

In SQL Developer, to set autocommit on, go to **Tools | Preferences** and you'll see this option under Expanded Database and Worksheet parameters.

10.6.3 SELECT… FOR UPDATE

Now that you understand how transactions work and their purpose, we can resolve an issue many developers are surprised to encounter. What if you program logic like this:

```
SELECT  column_list  FROM  table_name ;
/* Present data to the user for their update here            */
/* The user inspects the data and decides to update it… */
UPDATE  column-list  FROM  table_name ;
```

The problem is that some other user might update the data you presented to your user in the first SELECT statement, while your user is still viewing the data and considering whether to update it. So by the time your user launches the UPDATE statement, the data in the table is no longer consistent with what they viewed on their screen!

You can solve this problem by replacing the first SELECT statement by one with a FOR UPDATE clause:

```
SELECT  column_list  FROM  table_name  FOR UPDATE ;
```

This locks the data for your exclusive update option until you release it to others by issuing a COMMIT or ROLLBACK command or end the transaction implicitly. The benefit is that your UPDATE statement is guaranteed to work against the exact same data that was retrieved by your initial SELECT…. FOR UPDATE statement. The cost is that other users and programs are prevented from updating the data until you issue your COMMIT or ROLLBACK.

You can see that this update issue is a subtle problem of database **locking**. Locking by Oracle database protects **data integrity**, the accuracy of the data. Most of this locking is transparent to users and programs; it is automatic and they are not even aware of it. But the example above shows that sometimes developers do need to be aware of locking and how to code programs that work as expected.

Locking is based on the four underlying principles, called **ACID test** due to their initials. Databases like Oracle that ensure ACID principles guarantee data integrity:

A is for **Atomicity**	**Atomicity** means that all parts of a transaction must complete, or else none of them complete
C is for **Consistency**	**Consistency** means that the results of a query must be consistent with the state of the database at the time the query started
I is for **Isolation**	**Isolation** means that an incomplete transaction must not be visible to any user or program except the one currently working on it
D is for **Durable**	**Durability** means that once a valid transaction completes (is committed), the database must never lose it

11.0 Creating Tables Through DDL

Each **user** who connects to the Oracle database has his or her own **schema**, the set of database objects they own. These database objects might include:

- Tables- for physically storing data
- Views- alternative ways of viewing table data
- Indexes- physical structures that organize data and speed data access
- Synonyms- aliases for tables or views
- Sequences- a construct that generates unique numbers

Oracle's **data dictionary** is a special set of objects that keep track of all the other objects. You could query the data dictionary to find out the names of the tables in your schema, for example. When you issue the DESCRIBE command, it retrieves information about the table you specify from the data dictionary to display it to you. The data dictionary view named USER_OBJECTS lists basic information about all the objects in your schema.

Your schema is also by default your **namespace**; that is, the area in which names must be unique. Thus each table you create must have a unique name within your schema or namespace. A table and a view could not have the same name if they exist in the same schema.

All schema objects following the same naming rules. They must be:

- 1 to 30 characters long (except that database link names may be up to 128 characters)
- Start with a letter of the alphabet
- Contain only letters, digits, the underscore (_), dollar sign ($), or pound sign (#)

You cannot use a reserved word (like SELECT) for an object name. Object names are converted to all upper-case by default. So under normal conditions you would expect all object names in the data dictionary to be upper-case.

The SQL used to create and maintain objects is referred to as **Data Definition Language** or **DDL**. This contrasts to the **Data Manipulation Language** or **DML** statements we have worked with in previous chapters (SELECT, INSERT, UPDATE, and DELETE). The terms DDL and DML provide a quick way to distinguish between SQL statements that are used to manipulate data versus those they are used to create and maintain the underlying objects that facilitate storage and use of that data.

11.1 Creating Tables

There are several kinds of tables in an Oracle database. The exam only covers **relational tables**, the row-and-column tables we have worked with thus far.

Relational tables themselves come in a variety of internal organizations, such as heap tables, partitioned tables, clustered tables, etc. *From the standpoint of data retrieval and manipulation, you use the same SQL to manipulate the data, regardless of the underlying table type.* The exam only covers standard relational tables stored in the default form, called **heap tables**. All the DDL in this chapter create and alter heap tables.

Let's create an example table that is similar to the HR.JOB_HISTORY table. First find out the structure of the HR.JOB_HISTORY table:

```
SQL> desc hr.job_history;
 Name                                      Null?          Type
 ----------------------------------------- -----------    ---------------

EMPLOYEE_ID                               NOT NULL       NUMBER(6)
START_DATE                                NOT NULL       DATE
END_DATE                                  NOT NULL       DATE
JOB_ID                                    NOT NULL       VARCHAR2(10)
DEPARTMENT_ID                                            NUMBER(4)
```

Now that we know the column names, their data types, and whether they can be nulled (not entered), we can reverse-engineer the CREATE TABLE statement to create a table just like it:

```
CREATE  TABLE  JOB_HISTORY
        (EMPLOYEE_ID      NUMBER(6)                NOT NULL,
        START_DATE        DATE                     NOT NULL,
        END_DATE          DATE                     NOT NULL,
        JOB_ID            VARCHAR2(10)             NOT NULL,
        DEPARTMENT_ID     NUMBER(4)        ) ;
```

Notice that all the column definitions together are enclosed in a single set of parentheses. Don't forget the final right parenthesis at the end of the column definitions. And remember, too, the single semi-colon that terminates the entire SQL statement.

This table is created in my own schema, as defined by my user id. If my user id is **designer**, my full table name in the form SCHEMA.TABLE_NAME would be DESIGNER.JOB_HISTORY. Remember that names are by default in all upper-case in the dictionary.

If my user id were HR, this CREATE TABLE statement would attempt to create a table named HR.JOB_HISTORY. Since a table with this name already exists in that namespace, the statement would return an error and create nothing.

One key parameter that is missing in the above table definition is DEFAULT. This permits you to declare default data value that is inserted into a column if the input data does not specify a value. This example ensures that START_DATE and END_DATE are never NULL by inserting the current date, SYSDATE, into them if the user does not explicitly provide dates for insertion. The example also shows setting JOB_ID to a default character string and DEPARTMENT_ID to a default numeric value:

```
CREATE  TABLE  JOB_HISTORY
        (EMPLOYEE_ID      NUMBER(6)          NOT NULL,
        START_DATE        DATE               DEFAULT SYSDATE,
        END_DATE          DATE               DEFAULT SYSDATE,
        JOB_ID            VARCHAR2(10)       DEFAULT 'DBA',
        DEPARTMENT_ID     NUMBER(4)          DEFAULT(60)  ) ;
```

To create tables you need to know the allowable data types. Here are Oracle's key data types. The ones you need detailed knowledge of for the exam are VARCHAR2, NUMBER, and DATE. The others you just need be familiar with:

Datatype:	**Use:**
CHAR(size)	Fixed length string up to 2000 bytes. Default and minimum SIZE is 1. Blank-padded to fit.
NCHAR(size)	Same as CHAR but for Unicode data. **Unicode** is a multibyte character set that can represent characters used in any human language.
VARCHAR2(size)	Variable-length string up to 4000 bytes. You **must** specify SIZE (minimum is 1).
VARCHAR(size)	Use VARCHAR2 instead.
NVARCHAR2(size)	Like VARCHAR2 for Unicode data.
NUMBER(p,s)	Number w/ P as Precision and S as Scale. Defaults to P=38 and S as null when not specified.
NUMERIC, DECIMAL, DEC, INTEGER, INT, FLOAT	Use NUMBER instead.
DATE	Date and Time (stored internally in 7 bytes). Includes Century, Year, Month, Day, Hour, Minute, and Second.
TIMESTAMP	Like DATE, but can store fractional seconds up to a precision of 9 digits.
TIMESTAMP WITH TIME ZONE	Like TIMESTAMP, but stores a time zone displacement.

TIMESTAMP WITH LOCAL TIME ZONE	Like TIMESTAMP, but stores the time as a normalized form of the database time zone.
INTERVAL YEAR TO MONTH	Stores a time interval as years and months.
INTERVAL DAY TO SECOND	Stores a time interval as days, hours, minutes, and seconds.
ROWID	Uniquely IDs rows (used by Oracle internally).
UROWID	Stores logical rowids for index-organized tables or non-Oracle tables (used by Oracle internally).
LONG	Character data of variable length up to 2 Gig. One allowed per table. **Recommended to use CLOB instead of LONG.**
CLOB	Character large object, up to 4 Gig.
NCLOB	Same as CLOB for Unicode data.
BLOB	Binary large object, up to 4 Gig.
BFILE	Pointer to an external LOB file that can be up to 4 Gig.
RAW	Binary info up to 2,000 bytes.
LONG RAW	Raw binary data up to 2 Gig. **Recommend to use BLOB instead.**

After we created our own JOB_HISTORY table similar to HR.JOB_HISTORY table, we might want to load the data from HR.JOB_HISTORY into our table. This requires two steps:

1. Creating the table
2. Loading the table with data

An alternative is to create the table and load it in a single statement. This technique creates the new table via a subquery and is often called the CREATE TABLE AS statement.

This SQL creates a new JOB_HISTORY table with the exact same definitions as the HR.JOB_HISTORY table and then loads all rows from the original HR.JOB_HISTORY table into the new table:

```
CREATE  TABLE  job_history
        AS  SELECT  *  FROM  hr.job_history ;
```

Perform a SELECT * query against the new JOB_HISTORY table and you'll see it contains the exact same data as the HR.JOB_HISTORY table. DESCRIBE statements show both tables have the same table definition and structure. Later in this chapter we'll discuss table constraints (like primary and foreign keys, etc). But for now note that while CREATE TABLE AS duplicates data types, NOT NULL, DEFAULT, and CHECK constraints into the new table, it does **not** duplicate primary key, column uniqueness, or foreign key constraints. This is because these three constraints would require index creation.

You can get fancier with table creation subqueries by applying any arbitrary WHERE clause to the subquery. The effect is to restrict what rows are loaded into the new table. This statement only loads rows into the new table that are from department 60:

```
CREATE TABLE job_history
    AS SELECT * FROM hr.job_history WHERE department_id = 60 ;
```

You can also manipulate the SELECT list in the subquery to change the design of the new table. For example, you can specify only a subset of the columns in the original table to come over to the new table. This example creates a new table that has only two of the original column definitions, which are loaded with the relevant data:

```
CREATE TABLE job_history
    AS SELECT employee_id, department_id FROM hr.job_history ;
```

This example performs a group function in creating a new table that contains the average sales from the original SALES_TABLE:

```
CREATE TABLE new_sales_table
    AS SELECT avg(sales_quantity) Average_Sales  FROM sales_table;
```

AVERAGE_SALES is the name of the new column that contains the average sales. It is the only column in the new table and it has exactly one row in it, which contains the sales average.

11.2 Modifying Existing Tables

Once a table has been created, you may modify the columns in almost any manner desired.
Here are examples:

Create a comment on a column by:

```
COMMENT ON COLUMN my_tab.column1 is 'THIS IS MY COMMENT' ;
```

To add a column:

```
ALTER TABLE table_name ADD (new_column column_definition) ;
```

For example:

```
ALTER TABLE job_history ADD (prior_position VARCHAR2(20)) ;
```

To drop a column:

```
ALTER TABLE table_name DROP COLUMN column_name ;
```

To modify a column's definition:

```
ALTER TABLE table_name MODIFY (column_name new_column_definition) ;
```

For example:

```
ALTER TABLE job_history MODIFY (department_id NUMBER(9) DEFAULT 8) ;
```

To rename a column:

```
ALTER TABLE table_name RENAME COLUMN old_name TO new_name ;
```

To mark a column as unused:

```
ALTER TABLE table_name SET UNUSED COLUMN column_name;
```

Marking a column as UNUSED makes it appear as if the column does not exist to SQL queries. For example, SELECT statements would not retrieve that column, and INSERT and UPDATE statements would not refer to that column. Marking a column as UNUSED is also a quick operation. In contrast, dropping columns is usually time-consuming because Oracle restructures each row in the table to remove the column's data. So it is often best to set unused columns as UNUSED rather than dropping them. Then at a later time when processing time is available, issue this command to finally drop the unused columns:

```
ALTER TABLE table_name DROP UNUSED COLUMNS;
```

Another consideration in the above statements is the role of NULL columns. Oracle will not let you alter a column to be NOT NULL if any existing rows are NULL for that column. Similarly, it does not make sense to try to add a NOT NULL column to an existing table, as that would be incompatible with the initial state of the newly-defined column.

11.3 Table Operations

Create a comment on a table by:

```
COMMENT ON TABLE my_tab IS 'THIS IS MY COMMENT' ;
```

Rename a table by:

```
RENAME old_table_name TO new_table_name ;
```

Drop a table by the DROP TABLE statement:

```
DROP TABLE table_name;
```

Dropping a table removes the table definition from the data dictionary and eliminates all table data. The DROP TABLE only succeeds if it has exclusive access to the table (that is, if no transactions are active on the table).

Also there must not be any foreign key constraint that refers to the table: no other table must depend on this one as its "parent." You must add the keywords CASCADE CONSTRAINTS if there exist any constraints on the table. The DROP statement permanently eliminates the table and its data, as well as any constraints, indexes, triggers and privileges on that table.

11.4 Constraints

Constraints are rules or restrictions that apply to tables and their columns. They encode **business rules** in the database (instead of in your program logic) for simplicity, accuracy, universality, and data integrity.

When a SQL statement that changes data runs against a table, it must not violate the constraint rules on that table. If it does, it fails (by default).

This summarizes Oracle's five allowable constraints:

Constraint:	**Keyword:**	**Use:**
Primary Key	PRIMARY	One allowed per table to uniquely identify rows. Index may be automatically created. Primary keys are UNIQUE and NOT NULL.
Uniqueness	UNIQUE	As many per table as desired. Index may be automatically created. Unless you add NOT NULL to this, UNIQUE columns could have multiple null indicators.
NOT NULL	NOT NULL	Requires a column to contain some value.
Check	CHECK	Imposes value restrictions for columns.
Foreign Key	FOREIGN KEY… REFERENCES	Used for **Referential Integrity** (RI). Refers back to a unique or primary key in the parent table. The table containing the foreign key is dependent on the table it points to (often called its **parent**).

Key constraints can be defined over multiple columns. These are referred to as **composite keys**.

There are two main ways to specify constraints when you first define a table. You can place a constraint on the same line as the column definition to which it refers, or you can place constraints on their own lines. This example shows both techniques:

```
CREATE TABLE my_employee_table
        (EMPLOYEE_ID  NUMBER(6)    PRIMARY KEY,
        SALARY        NUMBER(8,2) CHECK (salary BETWEEN 10000 and 1000000),
        TAX_INCREASE NUMBER(6)
                CONSTRAINT tax_check CHECK (TAX_INCREASE > 100),
        DAYS_WORKED      NUMBER(9)          NOT NULL,
        PREV_DAYS                NUMBER(9)          CONSTRAINT prev_here NOT NULL,
        SECONDARY_ID     NUMBER  UNIQUE,
        SEC_ID_TOO               NUMBER  CONSTRAINT sec_uniq UNIQUE,
        EMP_ID_REF               NUMBER(6)          UNIQUE,
        CONSTRAINT unq_sal_inc UNIQUE (salary,tax_increase)  )  ;
```

This example also shows that you can leave constraints unnamed, or you can give them specific names through use of the CONSTRAINT keyword. Adding a constraint name is useful when you violate the constraint, because the error message will contain the constraint name. It's a good documentation practice to name constraints.

Usually constraints are coded on the same line as the column to which they refer, but the last line of code above shows that you can also place constraints at the end of the table definition. Where you see this most frequently is in the definition of foreign key constraints.

How many indexes would automatically be created as a result of the above CREATE TABLE statement? Five—every PRIMARY KEY and UNIQUE constraint forces an index to be automatically created if you do not create the required index(es) explicitly yourself.

Foreign keys relate one table to another. The table containing the foreign key constraint points to its parent table and is dependent on it. These two CREATE TABLE statements illustrate the point. The second table is dependent on the first through its foreign key:

```
CREATE TABLE supplier
    (SUPPLIER_ID      NUMBER(8)         NOT NULL,
    SUPPLIER_NAME     VARCHAR2(30)      NOT NULL,
    CONSTRAINT supplier_pk PRIMARY KEY (supplier_id) ) ;

CREATE TABLE products
    (PRODUCT_ID       NUMBER(8)         NOT NULL,
    SUPPLIER_ID       NUMBER(8)         NOT NULL,
    CONSTRAINT  fk_supplier
        FOREIGN KEY (supplier_id)
        REFERENCES supplier(supplier_id) ) ;
```

The column in the parent table that the foreign key points to or **references** must already be defined (though there are ways to defer it, ultimately every row in the table containing the foreign key must refer to a valid parent table row). The two related columns do **not** have to have the same name in the two tables but they do have to have the same data type.

You can use ALTER TABLE to add or drop constraints on existing tables. Examples:

```
ALTER TABLE tab ADD (CONSTRAINT cname PRIMARY KEY (c1));
ALTER TABLE tab ADD (CONSTRAINT cname UNIQUE (c1));
ALTER TABLE tab ADD (CONSTRAINT cname CHECK (c3=c4));
ALTER TABLE tab MODIFY (c1 NOT NULL);
ALTER TABLE tab DROP CONSTRAINT cname ;
ALTER TABLE tab DROP PRIMARY KEY ;
```

Constraints may also be **enabled** (activated) or **disabled** (turned off) by use of the ALTER TABLE statement with the ENABLE or DISABLE keywords. Examples:

```
ALTER TABLE table_name DISABLE CONSTRAINT constraint_name ;
ALTER TABLE table_name ENABLE CONSTRAINT constraint_name ;
```

11.5 More on Foreign Key Constraints

Foreign key constraints involve two tables in a **parent-child relationship**. The foreign key definition occurs in the table definition of the child table. This key corresponds to some key in the parent table.

This leads to issues you have undoubtedly run into if you've ever programmed SQL that updates related tables. If you try to INSERT a row into the child table and its foreign key does not already exist in the parent table, the INSERT will fail. How about if you try to delete a parent row on which child row(s) are still dependent?

In the REFERENCES clause in the foreign key definition, you can encode either:

```
ON DELETE SET NULL   |   ON DELETE CASCADE
```

The former sets the foreign key column(s) in the child table to NULL if you delete the row in the parent table on which they are dependent. The latter causes the deletion of the parent row to cascade to all its dependent child rows. So deleting the parent row deletes all rows in child table(s) that depend on it. Omit both clauses, and Oracle will not let you delete a parent table row that still has any related dependent rows in any child table.

12.0 Creating Other Schema Objects

Along with tables, the exam covers other schema objects likely to be used by the typical programmer or developer. These are views, synonyms, sequences, and indexes. Objects concerned with the actual physical storage of data within the database–such as tablespaces or clusters–are *not* included.

12.1 Views

Views are "virtual tables," **stored queries** that can be treated like tables in SQL. Their definitions are stored in Oracle's data dictionary, but they do not consume physical storage on disk. Views are internally just stored SELECT statements that run against tables.

Views can be defined upon tables or other views and view definitions may be nested (although too much nesting costs performance). Creating a view does **not** create a new table… a view is simply a stored query into an existing table. The purposes of views are to:

- Give users a different view of data
- Simplify queries for users
- Enforce security (example: hide certain columns from user access)
- Reduce errors
- Help performance

The basic syntax to create views is:

```
CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW
        [schema.]view_name [(alias[,alias]…)]
AS subquery
        [ WITH CHECK OPTION [CONSTRAINT constraint_name]]
        [ WITH READ ONLY [CONSTRAINT constraint_name]] ;
```

The *subquery* describes the SELECT statement that defines the view.

Use CREATE OR REPLACE to replace the view definition with a new one, if the view already exists. Since you can not adjust the column definitions in a view once it has been created (unlike a table), you must drop and recreate a view to redefine its attributes. The OR REPLACE option makes this possible in a single SQL statement.

Use the WITH CHECK OPTION to ensure that updates through the view are also selectable from the view (to ensure constraints apply all the way through when basing one view on other(s)). This option is not enabled by default. Use READ ONLY to make the view non-updatable.

The FORCE option creates the view even if the table upon which it is defined does not exist. But remember that you can not use the view until the underlying conditions are met. NOFORCE is the default; in this case you can only create the view if its underlying table is already defined.

Here is an example view definition. It creates a view with three columns, all based on the EMPLOYEES table. A WHERE clause means this view will only show employees whose DEPARTMENT_ID is 20:

```
CREATE VIEW emp_view AS
        SELECT last_name, first_name, employee_id
        FROM employees
        WHERE department_id = 20;
```

Can you update data using a view? In general, you can with **simple views** but you can not with **complex views**.

Views are not updatable by INSERT and UPDATE if they join detail tables in complex fashion, use functions, or perform aggregation. Specifically, you can not INSERT or UPDATE through a view if it:

- Contains DISTINCT, GROUP BY, CONNECT BY, or START WITH
- Contains computed expressions in the column list
- Does not contain all NOT NULL columns in its underlying table (and these must be assigned values by the updating statement)
- Have nested table columns or flattened subqueries
- Contains CAST or MULTISET expressions
- Uses the set operators

The rules for updatability of **join views** (views based on more than one table) are complicated. I recommend just querying the data dictionary table called USER_UPDATABLE_COLUMNS to see what columns in a join view can be updated. An Oracle feature called INSTEAD OF triggers can sometimes be used to get around some of the restrictions on updatable views.

You can **not** change a view by the ALTER VIEW statement. Instead re-create the view without loss of granted privileges by issuing CREATE OR REPLACE VIEW statement.

The main use of the ALTER VIEW statement is to recompile an existing view. The compilation or recompilation process ensures that the view is validly based on its underlying table(s) and that it is ready to be used. Use the DROP VIEW statement to eliminate a view.

Object views are views created upon traditional relational tables to make them appear like object tables. Object views contain the key identifying phrases OF and WITH OBJECT IDENTIFIER. This is example creates an object view, based on relational table EMP and the type definition EMP_T:

```
CREATE VIEW emp_obj OF emp_t
        WITH OBJECT IDENTIFIER (employee_id) AS
        SELECT employee_id, last_name, job_id FROM employees
        WHERE job_id='SH_CLERK' ;
```

12.2 Synonyms

Use a **synonym** as shorthand or alias for an object name. You can refer to an object simply by coding its synonym. Synonyms provide:

- Shorter, simpler naming
- Location transparency for remote queries
- Insulation of programs from changes to object names

Database Administrators create PUBLIC synonyms for use by everyone, while users can create their own **private synonyms** for their own sole use. Synonyms can include the final parameter @DATABASE_LINK, representing a **database link**—a reference to a remote object on another computer. Syntax for CREATE, ALTER, and DROP SYNONYM are:

```
CREATE [PUBLIC] SYNONYM synonym_name FOR object_name;
ALTER SYNONYM synonym_name COMPILE;
DROP [PUBLIC] SYNONYM synonym_name;
```

Like views, synonyms become invalid if the underlying object upon which they are based is dropped. The ALTER… COMPILE statement recompiles and revalidates a synonym.
Here's an example of creating a synonym:

```
CREATE SYNONYM emp FOR hr.employees ;
```

This example creates a PUBLIC synonym everyone can use that creates a shorthand name for a remote table residing on a different computer:

```
CREATE PUBLIC SYNONYM remote_emp
   FOR hr.employees@remote.site.com ;
```

12.3 Sequences

Sequences are a mechanism for generating unique integer values. They are sometimes used to generate primary keys.

In SQL, the Oracle-generated sequence values are accessed through the pseudo-columns CURRVAL and NEXTVAL. For example, if you create a sequence named ORDER_SEQUENCE, you could access the next generated value from that sequence and use it in an INSERT statement like this:

```
INSERT INTO orders_table (order_number, order_date, customer_id)
        VALUES (order_sequence.nextval, sysdate, '1505') ;
```

CURRVAL and NEXTVAL can be used in the:

- Select list of SELECTs
- VALUES clause or subquery select list of INSERTs
- SET clause of an UPDATE

but can **not** be used in:

- A subquery, inline view, view, or snapshot
- The WHERE clause
- A CHECK constraint
- The DEFAULT column value in CREATE/ALTER TABLE
- SELECTs with UNION, INTERSECT, or MINUS operators
- With DISTINCT, GROUP BY or ORDER BY clauses

Here is the basic syntax to create a sequence. Default keywords are underlined:

```
CREATE SEQUENCE [schema.]sequence_name
        [ INCREMENT  BY number ]
        [ START WITH number ]
        [ MAXVALUE number |  NOMAXVALUE]
        [ MINVALUE number |  NOMINVALUE]
        [ CYCLE | NOCYCLE]
        [ CACHE number | NOCACHE ]
        [ ORDER | NOORDER]  ;
```

Control the generated values by keywords START WITH, INCREMENT BY, MINVALUE, MAXVALUE, CYCLE, and ORDER. CACHE speeds access by caching the specified number of values held in Oracle' memory area, the **System Global Area** or **SGA**, but also means values are lost when the database is shut down (or goes down). This results in a gap in your sequence. *Generated sequence numbers are guaranteed to be unique and ordered but they are not guaranteed to be without any gaps.* CACHE default value is 20.

Here is the example CREATE SEQUENCE statement used for the orders insertion example of NEXTVAL above:

```
CREATE  SEQUENCE  order_sequence ;
```

Since this sequence assumes the defaults, it will start with 1, increment by 1, and have no minimum or maximum value. 20 sequence values will be generated and stored in cache memory.

This sequence specifies that values start at 1,000 and increment by 10 each time. The NOCACHE keyword reduces performance since pre-generated keys are not kept ready in memory, but also tends to reduce sequence number gaps:

```
CREATE SEQUENCE  parking_stickers_seq
        START WITH     1000
        INCREMENT BY   10
        NOCACHE
        NOCYCLE ;
```

12.4 Indexes

The purposes of indexes are to:

- Speed data access
- Avoid expensive sorts
- Enforce constraints (eg, uniqueness or primary key constraints)

Indexes are implicitly created for you by Oracle when you define unique or primary key constraints, or you can explicitly create them by issuing the CREATE INDEX statement.

There are two basic types of index:

- B-tree
- Bitmap

Unless you specify otherwise, indexes are created as b-tree by default. These work best with **high cardinality data** (a wide range of values) and on big tables. Indexes may be ascending (ASC the default) or descending DESC. To enforce key uniqueness within the index, specify keyword UNIQUE. Specifying UNIQUE does **not** preclude the possibility of multiple NULL entries for columns that permit nulls. Indexes containing more than one column are called **composite indexes**.

Create a bitmap index by keyword BITMAP. BITMAP indexes are recommended for **low-cardinality data** (data having few different values, e.g., gender). They save space and work fast. They are probably not so good if you frequently update the index column values. They are definitely not good for data with many different values, ie high-cardinality data.

Function-based indexes allow you to create an index using a SQL function as long as that function is **deterministic** (always gives the same results for the same inputs, however or whenever used).

Here is the basic index creation statement:

```
CREATE [UNIQUE | BITMAP] INDEX [schema.]index_name
    ON [schema.]table_name (column [,column…] ) ;
```

To DROP an index just:

```
DROP [schema.]index_name ;
```

Here are a few index creation examples. This creates a b-tree index:

```
CREATE INDEX emp_mgr_ix
    ON employees (manager_id);
```

This creates a unique, composite index spanning the EMPLOYEE_ID, LAST_NAME, and FIRST_NAME columns:

```
CREATE UNIQUE INDEX emp_comp_ix
    ON employees (employee_id, last_name, first_name);
```

This statementt creates a bitmap index. Assumedly the column LIST_PRICE has a limited range of possible values:

```
CREATE BITMAP INDEX product_bm_ix
    ON product_information_part(list_price);
```

Practice Questions

Chapter 1

1. A schema is:
   - ❍ A. The collection of all the tables owned by a user
   - ❍ B. The collection of all database objects owned by a user
   - ❍ C. The collection of all data owned by a user
   - ❍ D. The collection of all programs and data owned by a user

1. Which statements are true (check all that apply):

   - ❍ A. SQL is a set-oriented language
   - ❍ B. SQL does not have flow-of-control statements (like IF or DO-WHILE)
   - ❍ C. SQL*Plus and SQL Developer provide ways to run SQL
   - ❍ D. SQL statements can not be placed with PL/SQL programs

Chapter 2

1. When does a simple SELECT statement alter table data?

   - ❍ 1. When does a simple SELECT statement alter table data?
   - ❍ A. When the statement logic indicates an update
   - ❍ B. When a CHANGE keyword or subclause is encoded
   - ❍ C. When the FOR UPDATE clause is coded
   - ❍ D. When required by the issuing program
   - ❍ E. Never

2. Which statement will label the output column with this word in all upper-case: LAST

   - ❍ A. SELECT last_name as 'last' FROM employees;
   - ❍ B. SELECT last_name as 'LAST' FROM employees;
   - ❍ C. SELECT last_name as last FROM employees;
   - ❍ D. SELECT last_name as "last" FROM employees;

Chapter 3

1. Choose the proper statement about this SQL query:

```
SELECT distinct job_id FROM employees
        WHERE hire_date < '01-JAN-2000'
        AND WHERE salary < 15000  AND WHERE department_id <> 50  ;
```

   - ❍ A. The query will return rows matching the hire_date, salary and department_id conditions
   - ❍ B. The query will not run because the "not equals" syntax <> is incorrect and must be !=
   - ❍ C. The query will not run because of the WHERE keywords
   - ❍ D. The query will not run because all WHERE conditions must be specified on the same line

2. Which statement is true?

- ❍ A. The default for DEFINE is OFF and its purpose is to allow you to define session variables
- ❍ B. The default for DEFINE is ON and its purpose is to allow you to define session variables
- ❍ C. The default for DEFINE is ON and its purpose is to remove session variables
- ❍ D. The default for DEFINE is ON and its purpose is to show the values of session variables immediately prior to and after their substitution

Chapter 4

1. Why would you want to explicitly convert a value between data types rather than having Oracle do it for you implicitly?

- ❍ A. You don't know which data type Oracle will convert the value to.
- ❍ B You always get more accurate results when you explicitly convert values yourself.
- ❍ C. Performance. Explicit conversions are faster.
- ❍ D. You have greater control and reliability, and explicitly document the conversion when you code it yourself.

2. What is the difference between the CONCAT function and the concatenation operator?

- ❍ A. The CONCAT function can also perform string replacements
- ❍ B. The CONCAT function can take any number of string arguments
- ❍ C. The concatenation operator can take any number of string arguments
- ❍ D. The concatenation operator can also perform string replacement

Chapter 5

1. The NULLIF function:
   - ❍ A. Compares two parameters and returns NULL if they are equal and the 1st parm otherwise.
   - ❍ B. Tests its 1st parameter and returns the 2nd parm if the first is NULL. Otherwise it returns the 1st parm.
   - ❍ C. Tests its 1st parameter. If it is NOT NULL, the 2nd parm is returned, otherwise the 3rd parm is returned.

2. What will this return: TO_CHAR(TO_DATE('01-MAY-09','DD-MON-RR'),'Day')

- ❍ A. The day of the week for 01-MAY-09 (which happens to be 'Friday')
- ❍ B. The day of the week for 01-MAY-09 in all caps (which happens to be 'FRIDAY')
- ❍ C. The ERROR: ORA-01756: quoted string not properly terminated
- ❍ D. You can't run this clause because data types are incompatible

Chapter 6

1. Which two templates reflect the correct order for SELECT statement keywords?

- ❍ A. SELECT … FROM … WHERE … ORDER BY …GROUP BY … HAVING
- ❍ B. SELECT … FROM … WHERE .. GROUP BY .. HAVING .. ORDER BY
- ❍ C. SELECT … FROM … WHERE .. GROUP BY … ORDER BY … HAVING
- ❍ D. SELECT … FROM … WHERE .. ORDER BY … HAVING … GROUP BY
- ❍ E. SELECT … FROM … WHERE …HAVING … GROUP BY … ORDER BY
- ❍ F. SELECT … FROM … WHERE …HAVING… ORDER BY

2. What will be the result of executing this statement:

```
SELECT  MAX(SUM(AVG(salary)))
        FROM hr.employees
        GROUP BY department_id  ;
```

- ❍ A. The query returns the sum total of all salaries for each department grouped by department id
- ❍ B. The query returns the maximum sum total of the average salaries for the department
- ❍ C. The query returns which department has the largest total payout in salaries
- ❍ D. ORA-00935: group function is nested too deeply

Chapter 7

1. If you want to retrieve all rows that match between two tables and also the rows from one table that do not have matches in the other, what kind of join do you want to perform?

- ❍ A. Full Outer Join
- ❍ B. Cartesian Join
- ❍ C. Equijoin
- ❍ D. Nonequijoin
- ❍ E. Left or Right Outer Join
- ❍ F. Cross join

2. What is the purpose of table aliases?

- ❍ A. To tell Oracle which tables to join
- ❍ B. To rename tables in output reports
- ❍ C. To rename tables in a way that corresponds to related column aliases
- ❍ D. To help quality ambiguous columns with short table abbreviations

Chapter 8

1. Which statement will delete staff whose department number is that of the lowest department number in the ORG table?

- ❍ A.
```
DELETE FROM staff
        WHERE dept =
                (SELECT MIN(deptnumb) FROM org) ;
```

- ❍ B.
```
DELETE FROM staff
        WHERE dept IN SOME
                (SELECT MIN(deptnumb) FROM org) ;
```

- ❍ C.
```
DELETE * FROM staff
        WHERE dept =
        (SELECT MIN(deptnumb) FROM org) ;
```

- ❍ D.
```
DELETE * FROM staff s
        WHERE salary >
        (SELECT AVG(salary) FROM staff
        WHERE dept = s.dept)
ORDER BY dept, id ;
```

2. In a correlated subquery:

- ❍ A. The single-row subquery is executed only once, and its result is passed to the outer query
- ❍ B. The subquery is executed one time for each row in the parent query
- ❍ C. The subquery is executed once per retrieval of all rows or a group of rows
- ❍ D. None of the above

Chapter 9

1. Which statement is true?

- ❍ A. UNION ALL sorts results and removes duplicates
- ❍ B. UNION ALL neither sorts results nor removes duplicates
- ❍ C. UNION neither sorts results nor removes duplicates
- ❍ D. UNION sorts results but does not remove duplicates
- ❍ E. UNION does not sort results but does remove duplicates

2. You have three tables named A, B, and C. You want to remove all data from A and also remove its data dictionary definition. You want to remove all rows from B as quickly as possible but retain its data dictionary definition. You want to remove all rows from C but require logging of any changes and for possible rollback. What do you do:

- ❍ A. DROP table A, DELETE with no WHERE clause for tables B and C
- ❍ B. DROP table A, TRUNCATE table B, and DELETE with no WHERE clause for table C
- ❍ C. TRUNCATE can accomplish all three requirements with the proper parameters
- ❍ D. DROP table A, DELETE without a WHERE clause for table B, and TRUNCATE table C

Chapter 10

1. You issue the following commands while in SQL*Plus:

```
SQL>  update employees set salary = 100000 where employee_id= 101 ;
SQL>  create table my_employee_table as select * from employees ;
SQL>  delete from employees;
<<<  Poof!  You're session ends as your clumsy cube-mate trips over
        your computer's power cord and spill his hot coffee on you! >>>
```

Assuming all statements are valid and all required objects are available, what is the result of these statements?

- ❍ A. The UPDATE, CREATE TABLE, and DELETE have all been applied to the database
- ❍ B. The UPDATE and CREATE TABLE have been applied to the database but not the DELETE
- ❍ C. None of the three statements (UPDATE, CREATE TABLE, DELETE) have been applied to the database because you did not get to issue a COMMIT
- ❍ D. The UPDATE and the DELETE have been applied to the database because they are DML statements but the CREATE TABLE DDL statement was not

2. The SELECT with FOR UPDATE clause:

- ❍ A. Locks table rows so that you can later update them
- ❍ B. Immediately updates the rows specified
- ❍ C. Permits you to control whether other users can update the table rows
- ❍ D. Is no longer supported in Oracle 11g

Chapter 11

1. Which statement is true about columns defined with the UNIQUE attribute in table definitions?

- ❍ A. The column may not have rows with NULL values
- ❍ B. In the entire table there can be one row with a NULL value for that column
- ❍ C. In the entire table there can be multiple rows with NULL values for that column
- ❍ D. None of the above statements is true

2. Why do most people use the VARCHAR2 data type instead of CHAR?

- ❍ A. VARCHAR2 is variable-length, so it saves space, and it permits columns up to 2,000 bytes
- ❍ B. CHAR was not supported by older releases of Oracle database
- ❍ C. VARCHAR2 is variable-length, so it saves space, and it permits columns up to 4,000 bytes
- ❍ D. VARCHAR2 is variable-length, so it saves space. CHAR handles binary data

Chapter 12

1. For better performance with a large table with high-cardinality data, you should create which kind of index?

- ❍ A. A bitmap index
- ❍ B. A bitmap index with a composite key on all relevant retrieval columns
- ❍ C. A b-tree index
- ❍ D. A function-based index

2. The fastest, best way to change columns in a view is to:

- ❍ A. DROP then re-create the view
- ❍ B. Use the ALTER VIEW statement
- ❍ C. Use the CREATE OR REPLACE VIEW statement
- ❍ D. Create a new view the way you want it to look and leave the old one unchanged

Answers & Explanations

Chapter 1

1. ANSWER: B

A schema consists of all database objects "owned" by a user (all the database objects associated with a particular *user id* or *user account*). This includes tables, views, indexes, synonyms, and other objects later chapters will cover. Answer A is incorrect because a schema includes objects owned by the user beyond just tables, such as views and indexes. Answer C is incorrect because a schema refers to objects owned by the user (eg: tables, views, indexes), not data owned by a user. Answer D is incorrect because a schema refers to database objects owned by the user, not programs and data.

2. ANSWERS: A, B, C

SQL is a set-oriented language, meaning that one SQL statement can affect multiple rows of data in the database. SQL lacks flow-of-control statements, which is why you need languages into which you can embed SQL to create complete applications (like PL/SQL and Java). SQL*Plus and SQL Developer are client tools that are specifically designed to help you run SQL statements. Answer D is incorrect because you can embed SQL statements within PL/SQL programs.

Chapter 2

1. ANSWER: E

A SELECT statement is a read-only operation. It never changes data within tables. You can embed a SELECT statement within an enclosing INSERT, UPDATE or DELETE statement, but it is the INSERT, UPDATE or DELETE that changes the data (not the embedded SELECT statement.) Answer B is incorrect because the SELECT statement does not have a CHANGE keyword. Answer C is incorrect because the Oracle SELECT statement FOR UPDATE clause does not update data. It merely locks data for your update later on through an UPDATE statement (see chapter 10 for full details on this). Answer D is incorrect because a SELECT statement is a read-only operation.

2. ANSWER: C

The lack of any quotation marks means the column alias will appear in all upper-case in the output. Answers A and B will not run, since aliases can not appear inside of single quotation marks. Answer D will run because it uses double-quote marks around the alias, but it will list the alias in all lower-case letters as last. Double-quote marks ensure no translation to all upper-case letters occurs.

Chapter 3

1. ANSWER: C

A SELECT statement can only have one WHERE keyword, and this one incorrectly has three. Answer A is incorrect because the query will not run because it has more than one WHERE keyword. Answer B is incorrect because <> and != are equivalent and coding either is valid. Answer D is incorrect because SQL statements can span lines, as long as they are broken apart between and not within words, and as long as they are terminated by a semi-colon.

2. ANSWER: B

B is correct because the default for DEFINE is ON. DEFINE has two uses, one of which is the allow you to define session variables (the other is to list current session variables and their values). Answer A is incorrect because the default for DEFINE is ON. Answer C is incorrect because UNDEFINE is used to remove session variables, not DEFINE. Answer D is incorrect because VERIFY is used to display the values of session variables before and after their substitution, not DEFINE.

Chapter 4

1. ANSWER: D

When converting a data type yourself you have more control and can explicitly state the data type of the result. Answer A is incorrect because you can definitely determine which data type Oracle will convert the value to if you look it up in the Oracle documentation. Answer B is incorrect because you could possibly get more accurate results than Oracle, but this depends on what you code and the situation. Answer C is incorrect in that performance for conversions is not appreciably different whether you specify it or Oracle does it implicitly.

2. ANSWER: C

A big advantage to the concatenation operator over the CONCAT function is that you can splice multiple input strings together into one string in one command. Answer A is incorrect because the CONCAT function can only concatenate strings, it can not perform string replacements (use the REPLACE function for that). Answer B is incorrect because the CONCAT function always takes exactly two input arguments. Answer D is incorrect because the concatenation operator can not perform string replacements, it only performs concatenation.

Chapter 5

1. ANSWER: A

The NULLIF function has two mandatory input parameters. It compares them and returns NULL if they are identical and the 1st parm otherwise. Answer B is incorrect because it is the definition of how the NVL function works, not the NULLIF function. Answer C is incorrect because it is the definition of how the NVL2 function works, not the NULLIF function.

2. ANSWER: C

The DD-MON-RR parameter is missing a right single quote mark, so the syntax error results. Answer A is incorrect because of the syntax error. If there were no missing single quote this answer would be correct. Answer B is incorrect due to the syntax error. Answer D is incorrect because all the data types in the clause are compatible.

Chapter 6

1. ANSWERS: B, E

These two orderings are correct. Answers A, C, and D are incorrect because the ORDER BY must always go last among SELECT clauses (not HAVING or GROUP BY). Answer F is incorrect because HAVING can never be coded in a SELECT statement without a GROUP BY.

2. ANSWER: D

The maximum allowable nesting of **group functions** is two levels. Since the first line of this statement nests group functions three levels deep, it fails with the message indicated. Remember that **single-row functions** do not have this limitation and can be nested as deeply as you like. Answers A, B, and C are incorrect because this statement fails with the syntax error message given in Answer D.

Chapter 7

1. ANSWER: E

A left or right outer join retrieves all matching rows plus rows from one of the tables that do not have matches in the other. A is incorrect because full outer joins retrieve all matching rows plus all rows from both tables that do not have matches in the other. Answer B is incorrect because a Cartesian join produces all combinations of rows from both tables. Answer C is incorrect because an equijoin retrieves matching rows from the tables only. Answer D is incorrect because an nonequijoin match column values based on an inequality condition. Answer F is incorrect for the same reason Answer B is incorrect. Cartesian joins and Cross joins are the same thing.

2. ANSWER: D

Table aliases are a short-hand way to refer to tables. Usually they are used to qualify columns so that those references are unambiguous since join statements refer to multiple tables. Answer A is incorrect because the FROM clause tells Oracle which tables to join. Answer B is incorrect because you don't use table aliases to rename tables in output reports. Answer C is incorrect because table aliases have no relationship to column aliases.

Chapter 8

1. ANSWER: A

Answer A is correct because the inner query returns the lowest department number from the ORG table, and then the outer query deletes all staff with this department number. Answer B is syntactically incorrect because there is no IN SOME operator in Oracle SQL. IN and SOME are individual keywords that are used separately. Answer C is syntactically incorrect because DELETE does not take an asterisk as an operand. It should be DELETE FROM STAFF not DELETE * FROM STAFF. Answer D is incorrect for the same reason as Answer C. It also is incorrect because it returns an average salary from the subquery, not a lowest department number.

2. ANSWER: B

Answer B is correct because a correlated subquery executes (evaluates) the subquery one time for each parent row. Answer A is incorrect because it refers to the behavior of regular **single-row subqueries**, not correlated subqueries. Answer C is incorrect because it describes **group functions**, not correlated subqueries. Answer D is incorrect because Answer B is correct.

Chapter 9

1. ANSWER: B

This answer is correct because UNION ALL does not sort results and it does not remove duplicates. Answer A is incorrect because UNION ALL does not sort results nor does it remove duplicates. Answer C is incorrect because UNION sorts results and removes duplicates. Answer D is incorrect because UNION removes duplicates. Answer E is incorrect because UNION sorts results.

2. ANSWER: B

DROP removes a table and its data, TRUNCATE quickly removes data but leaves the data dictionary definition intact, and DELETE without a WHERE clause deletes all rows from a table while logging changes for possible rollback. Answer A is incorrect because DELETE with no WHERE clause can not handle the requirements for table B. Answer C is incorrect because TRUNCATE can only accomplish the objectives for table B. Answer D is incorrect because DELETE without a WHERE clause is not as fast as TRUNCATE for the requirements of table B, and because TRUNCATE'ing table C ensures no logging and no possible ROLLBACK.

Chapter 10

1. ANSWER: B

Answer B is correct. The CREATE TABLE issues an implicit COMMIT that applied it and the prior UPDATE statement changes to the database. The DELETE was not applied to the database because no explicit or implicit COMMIT occurred after it was issued. Instead an involuntary end-of-session occurred which prevented the DELETE change from being committed to the database. Answer A is incorrect because the DELETE was not committed to the database. Answer C is incorrect because the CREATE TABLE implicitly committed itself and the prior UPDATE changes to the database. Answer D is incorrect because whether they changes were committed to the database is not a mere function of DML versus DDL. As correct Answer B shows, the sequence of statements is also critical to determining what changes are committed to the database.

2. ANSWER: A

SELECT with FOR UPDATE locks table rows until you issue a COMMIT or ROLLBACK. Before you do you typically update those rows. Answer B is incorrect because the SELECT with UPDATE statement itself does not update any data. Answer C is correct, in that you are locking out other users from updating the data until you issue a COMMIT or ROLLBACK. But it is not your main goal when issuing this statement. Answer A gives the true purpose. Answer D is incorrect because 11g supports this SQL statement.

Chapter 11

1. ANSWER: C

The UNIQUE attribute means all values inserted into the table for that column must be unique. But it does not prevent multiple rows from being inserted with NULL for that column. Answer A is incorrect because UNIQUE does not prevent insertion of rows with NULL for the column defined as UNIQUE. Answer B is incorrect because UNIQUE does not prevent insertion of multiple rows with NULL for the column. Answer D is incorrect because answer C is correct.

2. ANSWER: C

VARCHAR2 is variable-length so it saves space, and it permits a maximum column length of 4,000 characters. Answer A is incorrect because VARCHAR2 permits up to 4,000 bytes, not 2,000 (which is the limit for CHAR columns). Answer B is incorrect because CHAR is supported on older releases of Oracle. Answer D is incorrect because CHAR handles character data, not binary data.

Chapter 12

1. ANSWER: C

A b-tree index is recommended for improving performance for high-cardinality data in large tables. **High-cardinality** data are rows that have a large range of different values. Answer A is incorrect because bitmap indexes best improve performance for low-cardinality data. Answer B is incorrect because bitmap indexes best improve performance for low-cardinality data. Answer D is incorrect because function-based indexes are useful when you want to access data by applying some function to the data; they have nothing to do with data cardinality and the size of the table.

2. ANSWER: C

The CREATE OR REPLACE VIEW statement is specifically designed for this purpose, to help you easily change view definitions. Answer A is incorrect because it will accomplish the task but it is not as easy or efficient as Answer C. Answer B is incorrect because the ALTER VIEW statement can *not* change column definitions in views. Answer D is incorrect because it never actually changes the column in the original view. This approach would force you to give the new view a different name, and you would leave old unused views around.