ORACLE

SQL Certified Expert

And

Oracle PL/SQL Developer (OCA)

Exam Guides: 1Z0-047 and 1Z0-144

Todo:

-COMPLETAR PARTE PL/SQL

-HINTS

-EXPLAIN PLAN

-VISTA MATERIALIZADA

-FUNCIONES ANALITICAS

-CONCEPTO NORMALIZACION / DENORMALIZACION

-EJEMPLO FUNCIONAMIENTO SQL/PLUS

-COMO COPIAR UNA BBDD

-INSTALAR ORA.TSNAMES Y TODO ESO

-TRUCO DE INICIAR USUARIO CON OTRO ESQUEMA

-TRANSACCIONALIDAD: SERIALIZABLE

-INDICES SOBRE MISMAS COLUMNAS

What is next?".

- **Oracle Architecture** - Learn the Oracle RDBMS Architecture. You may not be a DBA, however learning the internals of the RDBMS will help you write more efficient SQL using the proper technics and features according to the business case. Learn how the background processes work, how undo and redo are generated. What is archivelog mode, what is row movement, row chaining and migration, etc.
- **Oracle Optimizer** - Gain a better and more in depth knowledge of the Oracle Optimizer. Learn the various Join methods and Access methods. Learn why it makes the choices it makes. Learn what are statistic, how to gather statistics, and why they are so important for the Cost Based Optimizer. Familiarise yourself with concepts such as index clustering factor, SELECTivity, and query transformation. Also, learn what hints are, how and when to use them. Finally, learn how to produce an explain plan and how to read it properly.
- **Data Concurrency and Consistency** - Make sure that you understand what is the **multiversion read consistency model** and how Read Consistency is achieved in Oracle. If you write applications for more than 1 concurrent user (you probably are) then if you do not understand the true essence of the multiversion read consistency model, in your application there will be bugs ready to happen.
- **Write Consistency** - This is an even more advanced topic that should be studied after you have a solid grasp of how Read Consistency works in Oracle. There are not many sources of information about this topic, but Tom Kyte's articles and answers in asktom cover pretty much the basic concept behind it. There are some rare but interesting things happening here and you will encounter terms such as **statement restart** and **mini-rollback**.
- **IOTs** - Learn how to create Index Organized Tables and when to use them. They may be extremely helpful in certain cases. They are easily created and there is no reason to remain unknown to the developer.

- **Materialized Views** - The materialized views are used mainly in Data Warehouses and Business Intelligence applications but can also be used in other cases as well. It's a nice feature to know.
- **Updatable join views** - You can apply DML on join views (views that the underlying SELECT statement joins tables) but certain conditions must be true in order for the DML to be successful. Learn the concept of **key preserved view**.
- **Analytic Functions** - I really wonder why Oracle prefers their SQL Experts to be familiar with Regular Expressions and not Analytics. I really do not get it. Anyhow, I believe that knowing how to properly use analytic functions has the potencial to make you write really powerful queries.
- **Table Partitioning** - When you are dealing with large data sets, partitioning can be life saver. Learn what is table partitioning and the various partitioning methods.
- **Table Compression -** Learn this extremely useful feature to save storage and increase the IO performance. Very good to know.
- **Result Cache** - A new feature in Oracle 11g that provides part of the memory in order to cache the result set of a query in order to get the same result set again later.
- **Locks and Deadlocks** - Learn the lock model. It is not easy but because the lock model is not easily understandable, it is an often trap for many developers. The better you understand it the better developer you will be.
- **Parallel Execution** - If your Oracle RDBMS has the capabilities to execute statements in parallel, then knowing how to properly use the parallel feature can improve the performance dramatically.
- **Direct-path Inserts (Append)** - Using the append hint you can insert data directly into the datafiles, skipping the SGA. This is called Direct-path insert. Very useful in ETL processes and when you want to load data in a table as fast as possible.
- **TKPROF** - A tool that every SQL guru should know how to use in order to troubleshoot mostly performance issues but also to reveal the internal processes of Oracle ie what Oracle does under the hood.

The next Oracle SQL features are not considered to be advanced but they have been left out of the 1Z0-047SQL Expert exam. Again, I do not understand the reason for this. I include them in this post because every Oracle SQL Expert should know about them.

1. **Invisible and Virtual Indexes** - During the development phase you can create new indexes at will. However, what happens after the application deployment? These technics can help in such cases. Easy to learn concepts that should be in the toolbox of every database developer.
2. **Virtual Columns** - Another new Oracle Database feature that can be handy in certain cases. Again, very easy to learn and to use.
3. **Sample** clause - In a query instead of accessing the whole table, you can instruct Oracle to access only a part of it. You can specify a percentage of the table using the sample clause.

The next 5 topics of the Oracle SQL are totally unknown to me at the moment. However, they are considered to be advanced and powerful features.

1. **Database Objects**
2. **Nested Tables**
3. **Partition Outer Join**
4. **Model clause**
5. **User-Defined Operators**

Introduction (C1)

The Exam: An Overview

- This chapter provides introductory material that is important to understanding preparing for the exam
- The 1Z0-047 Oracle Database SQL Expert exam, which is the subject of this exam guide, has 19 certification objective categories, of which ten are common to another exam, 1Z0-051 SQL Fundamentals I. While 1Z0-047 tests for all 19 of its categories, the exam tends to emphasize the nine areas that are unique to 1Z0-047 and not addressed on 1Z0-051
- The exam includes 70 questions and allows 120 minutes to complete them
- That's an average of less than two minutes per question

Define and Understand the Basics of the RDBMS

- A relational database consists of collections of data known as tables. A table could be a list of ship names and some statistics about each ship. Another table might be a list of employees who work on different ships. The "relational" aspect to a "relational database" has to do with the common information that "relates" two tables together—for example, the list of employees might include an entry for each employee's ship assignment, which would relate back to the list of ships and each ship's statistics
- A relational database management system, or RDBMS, is a system in which these relational tables and related objects can be created easily, using common functions to add, change, and remove data and database objects from the RDBMS

Define and Understand the Basics of SQL

- The Structured Query Language, or SQL, is the language used by programmers to interact with an RDBMS
- SQL statements can be used to create, alter, and drop database objects, such as tables
- SQL statements can add, change, and remove data from tables and other database objects

Understand the Oracle RDBMS and Oracle SQL

- Oracle Corporation released the first commercial RDBMS product. Today, Oracle is the industry leader in the RDBMS market
- The American National Standards Institute publishes a set of industry recognized standards for SQL. Oracle's implementation of SQL largely matches the ANSI standard but isn't 100 percent compliant. Oracle's competition is not fully compliant either
- Oracle's SQL*Plus command line interface is a great tool for entering and executing SQL commands from within any operating system platform

- Oracle’s SQL Developer tool is a great GUI for entering and executing SQL commands from within the Windows operating system

- The *SQL Language Reference Manual* is Oracle’s nearly 1,500-page manual that describes the Oracle implementation of the SQL language

Understand the Unique Role of SQL in Modern Software Systems

- SQL is most widely used fourth-generation language (4GL) in commercial use today

- SQL is the only language for interacting with the RDBMS. Any other programming language must use embedded SQL calls to interact with the RDBMS

- The constantly changing nature of databases makes them a tricky place to test software. If a SQL script is written and tested successfully today, it’s entirely possible that it may break down and produce erroneous information later on. The solution is that the script must not only be tested, but must originally be designed and written by a capable SQL developer who understands proper database design and is thoroughly versed in the RDBMS and SQL syntax

- The 1Z0-047 has been validated against Oracle database versions 10*g* and11*g*, so using either to prepare for the exam will be satisfactory

Using DDL Statements to Create and Manage Tables (C2)

Categorize the Main Database Objects (Tables, Constraints, Indexes, Roles, Sequences, Synonyms, Tables, Users, Views)

Database objects are the foundation of any database application.A database consists of one or more database objects. The following list shows objects that a database developer can create in the Oracle 11g RDBMS. Those marked withan asterisk are included on the exam.

Clusters **Constraints*** Contexts Database links Database triggers Dimensions Directories External procedure libraries **Indexes***	Index-organized tables Indextypes Java classes, etc. Materialized view logs Materialized views Mining models Object tables Object types Object views	Operators Packages Profiles Restore points **Roles*** Rollback segments **Sequences***	Stored functions/procedures **Synonyms*** **Tables*** Tablespaces **Users*** **Views***

The eight on the exam:

- Tables store data
- Constraints are rules on tables
- Views serve as a sort of “window” onto tables
- Indexes provide lookup support to speed queries on a table, like an index to a book
- Sequences are simple counter objects
- Synonyms are alternative names for existing objects
- Users are objects that own other objects
- Roles are sets of rights, or privileges, that can be granted to a user to give that user access to other objects

What Is a Schema?. Namespaces

A schema is a collection of certain database objects, such as tables, indexes, and views, all of which are owned by a user account. You can think of a “schema” as being the same thing as a user account, but there is a slight difference—the user account houses the objects owned by a user, and the schema is that set of objects housed therein.

Objects are either “schema”=owned by user account (tables, constraints, indexes, views, sequences, prívate synonyms) or “non-schema”=not owned (users, roles, public synonyms)

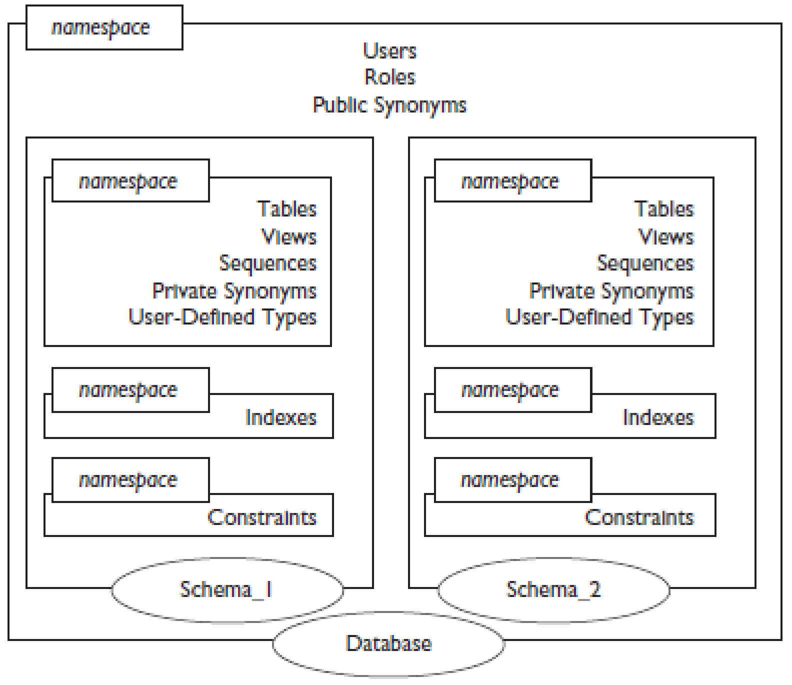
All database objects fall into one of two categories, or “types”. “Non-schema” objects cannot be owned by a user account. “Schema” objects are those objects that can be owned by a user account. “Non-schema” objects cannot be owned by a user account.For example, the USER object is a non-schema object.

Schema objects	Non schema objects

`Tables, Constraints, Indexes`
`Views, Sequences, Private Synonyms`
	`Users, Roles, Public Synonyms`

Create a Simple Table

- The CREATE TABLE statement is used to create a table
- You assign a name to a table by using the rules of naming database objects:
  - **Naming rules:**
    - The length of the name must be at least one character, and no more than 30 characters.
    - The first character in a name must be a letter.
    - After the first letter, names may include letters, numbers, the dollar sign ($),the underscore (_), and the pound sign (#), also known as the hash mark or hash symbol. No other special characters are allowed anywhere in the name.
    - Names cannot be reserved words that are set aside for use in SQL statements, such as the reserved words SELECT, CREATE, etc.
  - **Case sensitivity and double quotation marks:**
    - If a name is not enclosed in double quotation marks when it is created, then it will be treated as uppercase regardless of how it is created or referenced.
    - if a name is enclosed in double quotation marks, then it is case sensitive and must always be referenced with case sensitivity and with double quotation marks.
  - **Unique names and namespaces:**



So what happens if you try to create a database object with a name that matches the name of another database object that's already in the database?

- Objects that share a namespace must have unique names within that namespace.
- Objects in different namespaces are allowed to haveidentical names

Example: If you create a table in one schema called WORK_SCHEDULE, then you cannot create another table called WORK_SCHEDULE within that same schema. But you can do it in another schema, provided there isn't already a WORK_SCHEDULE in that schema.

  - **System-Assigned Names:** for example, when you create a table, and within the CREATE TABLE statement you optionally define an associated constraint, but without providing a name for the CONSTRAINT. The system will automatically generate a name for that constraint, a name that adheres to all the rules that we just reviewed.

- You also assign names to the table's columns using the same rules
- All tables have at least one column

Review the Table Structure

```
CREATE TABLE cruises(
cruise_id NUMBER,
cruise_type_id NUMBER,
cruise_name VARCHAR2(20),
captain_id NUMBER NOT NULL,
start_date DATE,
end_date DATE,
create_date DEFAULT SYSDATE NOT NULL,
status VARCHAR2(5) DEFAULT 'DOCK',
CONSTRAINT cruise_pk PRIMARY KEY (cruise_id) );
```

- The DESC command can be used to display a table's structure: it isn't a SQL statement; it's a SQL*Plus statement that is unique to Oracle. (Some other product vendors have since implemented DESC in their own unique SQL products.)
- The structure includes the table name, table columns, data types, and optional constraints
- *it is allowed to use SYSDATE in the default VALUE, as well as other functions (see create_date above)*

List the Data Types That Are Available for Columns

- Each column must be assigned a data type

Datatype character (CHAR(n), VARCHAR2(n))

- **CHAR(n):** character fixed size (max 2000). It pads any remaining unused space with blanks to ensure that the length of your value will always equal the value of n, example if CHAR(5) then value 'A' will be stored and retrieved as 'A   '
- **VARCHAR = VARCHAR2(n):** variable character (max 4000)

Datatype numeric (NUMBER(n))

- **NUMBER(n,m):** Accepts numeric data, including zero, negative, and positive numbers, where n specifies the "precision", which is the maximum number of significant digits (on either side of the decimal point), and m is the "scale", meaning the total number of digits to the right of the decimal point. EXAMPLE: number(5,2) denotes that you can store a number like 999.99. If NUMBER(3,2), trying to insert 10.56 will throw an error

  Note: If a precision is not specified, the column stores values as given. If no scale is specified, the scale is zero.

Datatype DATE (DATE, TIMESTAMP(N), TIMESTAMP(N) WITH TIME ZONE, TIMESTAMP(n) WITH LOCAL TIME ZONE, INTERVAL YEAR(n) TO MONTH, INTERVAL DAY(n1) TO SECOND(n2))

- **DATE[1]:** fields stored include year, month, date, hour, minute, and second. It does not include any time zone component. So if you copy a row that contains "04-SEP-2010 15:20" from a database in London accross a database link to a database in New York the value will be stored as "04-SEP-2010 15:20"

[1] Oracle defines in parameter NLS_DATE_FORMAT the default format for DATE when they are retrieved/entered from text, the NLS_DATE_FORMAT for UK and US ='DD-MON-RR'. Example: if column AS_OF of table T is DATE, then the statement UPDATE T SET AS_OF='18-MAR-12' updates the column AS_OF to a DATE having year=2012, Month = March and Year=2012

- **TIMESTAMP(n):** an extension of DATE, the value for n specifies the precision for fractional seconds. It defaults to 6. For example, `'26-JUN-02 09:39:16.78'` shows 16.78 seconds. The fractional seconds precision is 2 because there are 2 digits in '`78`'.

- **TIMESTAMP(n) WITH TIME ZONE:** A variation of TIMESTAMP that adds either a time zone region name, or an offset for time zone. It stores the time zone along with the time so the data is both unambiguous and presented in a consistent way to clients independent of the location of the client. So in our example the value would be “04-SEP-2010 15:20 +01:00” in both London and New York databases and for both London and New York users. The time zone reflects British summer time.

Note: you cannot change the database time zone if the database already contains any tables with columns of the TIMESTAMP WITH LOCAL TIME ZONE datatype. To know whether there are already tables with TIMESTAMP WITH LOCAL TIME ZONE, you can query the following view:

```
SELECT OWNER, TABLE_NAME, COLUMN_NAME, DATA_TYPE
FROM DBA_TAB_COLUMNS
WHERE DATA_TYPE LIKE '%LOCAL TIME_ZONE%'
ORDER BY OWNER, TABLE_NAME, COLUMN_NAME;
```

To change the session time zone:

```
ALTER SESSION SET TIME_ZONE = 'America/Los_Angeles';
```

- **TIMESTAMP(n) WITH LOCAL TIME ZONE:** differs from TIMESTAMP WITH TIME ZONE in that it does not store the time zone but stores the time normalized to UTC time. When queried, it presents its data in the user’s local time zone. It uses an offset from UTC time. If you copy a value of “04-SEP-2010 15:20 +1:00” from a database in London across a database link to a database in New York the value in both databases will be visible as “04-SEP-2010 15:20 +1:00” to a London client (or more precisely to any client with a session time zone of GB or similar) as before. However, the value in both databases will be seen as “04-SEP-2010 11:20 -04:00” to any client with a session time zone of “America/New_York” or similar. This may be a bit confusing as the same data will appear to different clients in different ways but there will be no ambiguity.

EXAMPLE:

```
CREATE TABLE new_order
(orderno NUMBER(4),
booking_date TIMESTAMP WITH LOCAL TIME ZONE);
```

The database is located in San Francisco where the time zone is -8:00. The user is located in New York where the time zone is -5:00. A New York user inserts the following record:

```
INSERT INTO new_order VALUES(1, TIMESTAMP “2007-05-10 6:00:00 -5:00”);
```

What will the result be when a San Francisco user queries: SELECTs the row, booking_date ?
Because it is ‘TS WITH LOCAL TIME ZONE’ Oracle will store that timestamp normalized to UTC time, which is 2007-05-10 11:00:00, so the query will return '2007-05-10 3.00.00.000000' because 2007-05-10 11:00:00 UTC is '2007-05-10 3.00.00.000000' in San Francisco

Date, **timestamp with local time zone** or **timestamp with time zone**? Depends, a medical application records a person's sleep patterns. You're probably going to be more interested in what time they went to bed, *in their timezone*, rather than what time they went to bed as of your own local timezone, so you’ll go for WITH LOCAL TIME ZONE

- **INTERVAL YEAR(n) TO MONTH**[2]**:** to store a difference in years and months (n=number of digits used to define the year value).Basically it is an interval literal that stores a period of time using the YEAR and MONTH datetime fields.
  - The TO_YMINTERVAL function converts a string representing a number of years and a number of months into an INTERVAL YEAR TO MONTH datatype.

    Example to_yminterval('03-11') => 3 years 11 months as an INTERVAL YEAR TO MONTH type

The following inserts are valid:

```
INSERT INTO coupons (coupon_id, duration) VALUES (1, INTERVAL '1' YEAR);
INSERT INTO coupons (coupon_id, duration) VALUES (3, INTERVAL '14' MONTH);
INSERT INTO coupons (coupon_id, duration) VALUES (4, INTERVAL '1-3' YEAR TO MONTH);
```

- **INTERVAL DAY(n1) TO SECOND(n2):** store a difference in days, hours, minutes and seconds (n1=number of digits to define the day, n2=number of decimals for the fractional part of the seconds)
  - The TO_DSINTERVAL function converts a string representing a number of days, hours, minutes and seconds into an INTERVAL DAY TO SECOND datatype.

    to_dsinterval('95 18:30:01) => 95 days, 18 hours, 30 minutes and 1 second

- Large Objects[3] (LOBs, 3 TYPES): LOBs can generally be used like other datatypes. Tables may have multiple columns with LOB datatypes.
  - **BLOB:** is an abbreviation for Binary Large OBject. BLOB accepts large binary objects, such as image or video files. The maximum size is calculated by way of a formula that includes several items, including a starting size of 4GB, something called the CHUNK parameter, and the setting for the database block size, which is a setting that affects all storage in the database.
  - **CLOB:** for Character Large OBject
  - **NCLOB:** Accepts CLOB data in Unicode

  IMPORTANT:

  1. LOBs cannot be primary keys, nor used with DISTINCT, GROUP BY, ORDER BY or joins.
  2. Size of the LOB is not defined at creation time, so it is wrong define a column such: `resume CLOB(200);`

[2] The big advantage of INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND is the size of each field, which is much lower than the size of DATE, TIMESTAMP…, they can be combined as follows:

```
SELECT sysdate, sysdate+to_dsinterval('003 17:00:00') as "3days 17hours later" from dual;
SYSDATE   3days 17h
--------- ---------
04-AUG-04 08-AUG-04
```

[3] There is the **LONG** type for storing character data of variable length up to 2 Gigabytes in length (bigger version of the VARCHAR2 datatype)., but Since Oracle 8i, Oracle advised against using the LONG datatype. Users should convert to CLOB or NCLOB types. NOTE: *Only one LONG column* is allowed *per table.*

- **USER DEFINED** (CREATE TYPE statement): this is PL/SQL and not included in the exam

Why doesn't Oracle RDBMS have a boolean datatype?

Since

```
flag char(1) check (flag in ( 'Y', 'N' )),
```

serves the same purpose, requires the same amount of space and does the same thing - I guess we feel this is a feature we can let them have that we really don't need.

I mean - what do you get back from a column in "access" that is a boolean? TRUE / FALSE. We'll give you Y/N -- if you would like TRUE/FALSE, we can accomplish that easily with DECODE(flag,'Y','TRUE','N','FALSE')

Explain How Constraints Are Created at the Time of Table Creation (5 types of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY and CHECK)

- The types of constraints are 5: NOT NULL, UNIQUE, PRIMARY KEY,FOREIGN KEY, and CHECK

1. **NOT NULL** constraint must be assigned a value for each row that is added to the table
2. **UNIQUE** constraint requires that if data is added to a column for a given row, that data must be unique for any existing value already in the column
3. A **PRIMARY KEY** constraint:
   a. is the combination of NOT NULL and UNIQUE
   b. may be assigned to one or more columns
   c. assigned to multiple columns is called a composite key
   d. A single table may only have one PRIMARY KEY

IMPORTANT

It is not possible to define over the same column(s) a PRIMARY KEY and UNIQUE constraints because a PRIMARY KEY is considered as a UNIQUE constraint + NOT NULL constraint, consequently the following is wrong:

```
CREATE TABLE BASKET_FIXING_REPROCESS (
  ID NUMBER,
  CONSTRAINT ord_pk PRIMARY KEY(ID),
  CONSTRAINT ord_uq UNIQUE(ID)
  );
```

4. A **FOREIGN KEY**:
   a. correlates one or more columns in one table with a set of similar columns in a second table
   b. requires that the second table already have a PRIMARYKEY assigned to the correlated columns before the FOREIGN KEY can be created
   c. Once created, the FOREIGN KEY ensures that any values added to the table will match existing values in the PRIMARY KEY columns of the second table
5. **CHECK**: it applies a small bit of code to define a particular business rule on incoming rows of data.

```
CREATE TABLE VENDORS
(VENDOR_ID NUMBER,VENDOR_NAME VARCHAR2(20),
STATUS NUMBER(1) CHECK (STATUS IN (4,5)),
CATEGORY VARCHAR2(5));
```

Restrictions on CHECK constrainsts:
A CHECK integrity constraint requires that a condition be true or unknown for every row of the table. If a statement causes the condition to evaluate to false, then the statement is rolled back. The condition of a CHECK constraint has the following limitations: - The condition must be a boolean expression that can be evaluated using the values in the row being inserted or updated. - The condition cannot contain subqueries or sequences. - The condition cannot include the SYSDATE, UID, USER, or USERENV SQL functions. - The condition cannot contain the pseudocolumns LEVEL, PRIOR, or ROWNUM. - The condition cannot contain a user-defined SQL function. Basically you can refer in you check conditions only fields from the current row. If you need something more complex, and a single-column or multi-column foreign key is not enough for this, you have to use a trigger.

- Constraints can be created with the:
  - CREATE TABLE statement
    - **"In line": definition of constraint when the column is defined**

```
CREATE TABLE PORTS (PORT_ID NUMBER PRIMARY KEY,PORT_NAME VARCHAR2(20));
```

    - **"Out of line": definition of the constraint after the definition of the column**

```
CREATE TABLE PORTS (PORT_ID NUMBER, PORT_NAME VARCHAR2(20), PRIMARY KEY
(PORT_ID) );
```

IMPORTANT: you cannot declare or add a NOT NULL constraint with the out-ofline syntax (See chapter 11. ALTER TABLE xxxx ADD CONSTRAINT)

  - the ALTER TABLE statement:

```
ALTER TABLE PORTS MODIFY PORT_ID CONSTRAINT PORT_ID_PK PRIMARY KEY;
```

- Multiple Constraints:

```
CREATE TABLE VENDORS VENDOR_ID NUMBER CONSTRAINT VENDOR_ID_PK PRIMARY KEY,
VENDOR_NAME VARCHAR2(20) NOT NULL,
STATUS NUMBER(1) CONSTRAINT STATUS_NN NOT NULL,
CATEGORY VARCHAR2(20),
CONSTRAINT STATUS_CK CHECK (STATUS IN (4, 5)),
CONSTRAINT CATEGORY_CK CHECK (CATEGORY IN
('Active','Suspended','Inactive')));
```

Datatype restrictions (BLOB, COB AND TS WITH TIME ZONE not allowed UNIQUE, PK and FK)

Datatype	NOT NULL	UNIQUE	PRIMARY KEY	FOREIGN KEY	CHECK
TIMESTAMP WITH TIME ZONE	Allowed	Not Allowed	Not Allowed	Not Allowed	Allowed
BLOB	Allowed	Not Allowed	Not Allowed	Not Allowed	Allowed
CLOB	Allowed	Not Allowed	Not Allowed	Not Allowed	Allowed

Note, however, those constraints may be applied to columns that have the datatype of TIMESTAMP WITH LOCAL TIME ZONE.

Manipulating Data (C3)

Describe Each Data Manipulation Language (DML) Statement

- There are six types of SQL statements; three types are subjects on the exam

Abbr	Type of SQL statement	SQL statements and Reserved words	
		Covered by the exam	Ignored by the exam
DDL	Data Definition Language	**CREATE** **ALTER (1)** **DROP** **RENAME** **TRUNCATE** **GRANT** **REVOKE** **FLASHBACK** **PURGE** **COMMENT**	**ANALYZE** **AUDIT** **ASSOCIATE** **STATISTICS** **DISASSOCIATE** **NOAUDIT**

DML	Data Manipulation Language	**SELECT**
		INSERT
		UPDATE
		DELETE
		MERGE (discussed CH 15)
			CALL
			LOCK TABLE
			EXPLAIN PLAN

TCL	Transaction Control Language	**COMMIT**
		ROLLBACK
		SAVEPOINT
			SET TRANSACTION
			SET CONSTRAINT

	Session Control Statements		**ALTER SESSION** **SET ROLE**
	System Control Statements		**ALTER SYSTEM**
	Embedded SQL Statements		**Any DML, DDL, or TCL that is integrated into a 3GL.**
(1) Except ALTER SYSTEM and ALTER SESSION, which are categorized under "System Control Statements" and "Session Control Statements", respectively.			

DML statements: SELEC, INSERT, UPDATE, DELETE, MERGE

Insert Rows into a Table

- The INSERT statement adds one or more rows to a table

- The INSERT syntax we reviewed in this chapter consists of the reserved words INSERT INTO, the name of the table, the optional column list, the reserved word VALUES, and the list of values to be entered

```
INSERT INTO CRUISES CRUISE_ID, CRUISE_TYPE_ID, CRUISE_NAME,
  CAPTAIN_ID, START_DATE, END_DATE, STATUS)
VALUES (1, 1, 'Day At Sea',101, '02-JAN-10', '09-JAN-10', 08 'Sched');
```

- If the INSERT statement is written so that the list of columns in the table is omitted, then the list of values must include one value for each column in the table, and the order of the columns in the table's structure will be how the list of values is expected to be sequenced within the INSERT statement

```
INSERT INTO CRUISES VALUES (1, 1, 'Day At Sea',101, '02-JAN-10', '09-JAN-10', 08
'Sched');
```

- The list of values in the INSERT statement may include expressions

- If any values violate any constraints applied to the target table, then an execution error will result—for example, all NOT NULL columns must be provided with some sort of value appropriate to the datatype of the column within the INSERT statement's list of values

- In the list of VALUES we can set DEFAULT as value (if the column was defined with a DEFAULT value), or we can use a function

```
INSERT INTO BASKET_FIXING_REPROCESS_TYPE (ID,display_order,start_range)
values ((15-14)*LENGTH('A'),DEFAULT,DEFAULT);
```

- VALUES Cannot be used in an INSERT with subquery

WRONG QUERY: `INSERT INTO new_orders (ord_id, ord_date, cust_id, ord_total)`

```
VALUES(SELECT order_id,order_date,customer_id,order_total FROM orders WHERE order_date
> '31-dec-1999');
```

We can achieve this without using VALUES, the following is right:

```
INSERT INTO FX_RATE_TYPE (ID, NAME, CREATED_BY, CREATED_ON, LAST_MODIFIED_BY,
LAST_MODIFIED_ON, ACTIVE) SELECT 1, 'Official Monthly Finance', 'admin.gfdtracker',
SYSDATE, 'admin.gfdtracker', SYSDATE, 1 FROM DUAL;
```

Update Rows in a Table

- The UPDATE statement modifies existing data in one or more rows within a database table

```
UPDATE CRUISES SET CRUISE_NAME = 'Bahamas', START_DATE = '01-DEC-11' WHERE
CRUISE_ID = 1;
```

- The UPDATE statement syntax starts with the reserved word UPDATE and the name of the target table, the reserved word SET, and then a series of assignment expressions in which the left side element is a table column, then the assignment operator (an equal sign), and then an expression that evaluates to a datatype appropriate for the target table's column identified on the left side of the equal sign, and finally an optional WHERE clause

- If additional assignment expressions are required, each additional assignment expression is preceded by a comma

- A If the WHERE clause is omitted, then all the rows in the table are changed according to the series of SET values listed in the UPDATE statement

- NOTE: the following syntax is OK:

```
update currency  set code='MXN', (name,active)=(SELECT 'Nuevo Peso', '1' from dual)
where id=95
```

- Note the following syntax is wrong:

```
update currency  set code='MXN', (name,active)=('Nuevo Peso',1) where id=95  -- WRONG:
oracle complains that it must be subquery, not a list of values
```

Delete Rows from a Table

- The DELETE statement is used to remove rows of data from a table

```
    DELETE FROM PROJECT_LISTING WHERE CONSTRUCTION_ID = 12;
```

- The syntax starts with the reserved words DELETE and the optional FROM, then the name of the target table, then an optional WHERE clause

- If the WHERE clause is omitted, all of the rows in the table are deleted

TCL statements: COMMIT, ROLLBACK SAVEPOINT

- TCL statements include:

- **COMMIT** : makes changes to the database permanent, and once committed, those changes can no longer be undone with a ROLLBACK ,there are two types of commit events: explicit commit and implicit commit
  - An explicit commit occurs with the COMMIT statement

    ```
    COMMIT;   (optionally could be COMMIT WORK; for compliance ANSI SQL)
    ```

  - An implicit commit occurs immediately before and after certain events that take place in the database, such as the execution of any valid DDL statement, such as CREATE, ALTER, DROP, GRANT, and REVOKE. Each is preceded and followed by an implicit commit

- **ROLLBACK**: it is used to undo changes to the database

- **SAVEPOINT**: it can be used to name a point within a series of SQL statements to which you may optionally roll back changes after additional DML statements are executed
  - Once a COMMIT is issued, all existing SAVEPOINTs are erased

Typical Question:

```
COMMIT;
SAVEPOINT A;
CREATE CLASS T …
INSERT INTO T …
ROLLBACK WORK TO A; -- THIS WILL FAIL BECAUSE SAVEPOINT A DOES NOT LONGER EXIST
```

- Any ROLLBACK that names non-existing SAVEPOINTs will not execute
- If ROLLBACK is issued without naming a SAVEPOINT, changes made by the user during the current session are rolled back to the most recent commit event

```
COMMIT;
UPDATE SHIPS SET HOME_PORT_ID = 21 WHERE SHIP_ID = 12;
SAVEPOINT SP_1;
UPDATE SHIPS SET HOME_PORT_ID = 22 WHERE SHIP_ID = 12;
ROLLBACK WORK TO SP_1;
COMMIT;
```

Note: All SAVEPOINT statements must include a name.

Typical Question

You executed the following SQL statements in the given order:

```
CREATE TABLE orders (order_id NUMBER(3) PRIMARY KEY, order_date DATE, customer_id
number(3));
INSERT INTO orders VALUES (100,'10-mar-2007',222);
ALTER TABLE orders MODIFY order_date NOT NULL;
UPDATE orders SET customer_id=333;
DELETE FROM order;
The DELETE statement results in the following error:
ERROR at line 1:
ORA-00942: table or view does not exist
```

What would be the outcome? All the statements up to the ALTER TABLE statement would be committed and the outcome of the UPDATE statement is retained uncommitted within the session, because the ALTER TABLE is a DDL and there is an implicit COMMIT

Retrieving Data Using the SQL SELECT Statement (C4)

SELECT statement: ROWNUM, ROWID, DISTINCT

Execute a Basic SELECT Statement

- A SELECT statement must include a SELECT list and a FROM clause

```
SELECT SHIP_ID, SHIP_NAME, CAPACITY FROM SHIPS BY SHIP_NAME;
```

- Any columns identified in the SELECT list must be in a table that is identified in the FROM clause

List the Capabilities of SQL SELECT Statements

- Pseudocolumns are defined by the system and are not stored with a table may be included in the SELECT list of a SELECT statement)
  - **ROWNUM**: This is the system-assigned number for a row. If you're looking for some way to number each row of output from a SELECT statement, beware: ROWNUM is assigned before the ORDER BY clause is processed, not after.

WRONG	RIGHT

`SELECT ... FROM ...`
`WHERE ...`
`and ROWNUM < 1000`
`ORDER BY ....`
	`SELECT ... FROM`
	`( SELECT ... FROM ...`
	`  WHERE ...`
	`  ORDER BY ....`
	`)  WHERE ROWNUM < 1000;`

  - **ROWID**: This is the system-assigned physical address for a given row. This can change from time to time by the database—

- DISTINCT = UNIQUE, can be used in a SELECT statement to list unique data sets: `SELECT DISTINCT LAST_NAME, FIRST_NAME FROM EMPLOYEES;` => returns different "last Name" and "First Name" existing in the table EMPLOYEES. **IT DOES COUNT NULL**

- The asterisk is a shorthand way of referring to all of a table's columns
- Expressions can transform data after it is retrieved from the database and before the data is produced as the SELECT statement's output
- Expressions may include arithmetic operators, SQL functions, and literal values
- Literal values include numbers, characters, dates, and intervals

- Arithmetic operators obey the rules of operator precedence:
    - Multiplication and division operators are evaluated before addition and subtraction, regardless of the order in which they appear in an expression
    - Parentheses have the highest authority in the order of operator precedence, which means that you can place parentheses to override any behaviour in the rest of the operations
- Functions can be used in expressions along with all of the other elements of expressions
- The WHERE clause identifies conditions that individual rows must meet in order to be displayed—in other words, it can be used to "restrict" rows from being displayed,

Typical Question: it is the same query 1 and query2 ? YES

QUERY 1: `SELECT * from currency where (display_order*minor_factor)=active`

QUERY 2: `SELECT * from currency where active=(display_order*minor_factor)`

- The ORDER BY clause sorts the data set output of a SELECT statement.
- GROUP BY / HAVING: The GROUP BY clause aggregates sets of records within a SELECT statement The HAVING clause can be used with GROUP BY to restrict sets of rows in the same fashion that ORDER BY can be used to restrict individual records

SELECT statement: PROJECTION, SELECTION and JOINING

- At the highest level, the SELECT statement can be characterized as having three fundamental capabilities:
    - **PROJECTION**: When a SELECT statement chooses fewer than all of the available column sin a table, it is exhibiting the concept of projection. It is accomplished through the SELECT list, also called the expression list, of the SELECT statement
    - **SELECTION**: When a SELECT statement chooses fewer than all of the available rows in a table, it is exhibiting the concept of SELECTion. It is accomplished with the WHERE clause of the SELECT statement
    - **JOINING**: When a SELECT statement chooses a combination of rows from more than one table by identifying common data that uniquely identifies rows, it is exhibiting the concept of joining. It can be accomplished in the SELECT statement with the WHERE clause or JOIN clause

The concept of NULL

The definition of NULL is the "absence of information". Sometimes it's mischaracterized as "zero" or "blank", but that is incorrect—a "zero", after all, is a known quantity.

For example:

```
SELECT 300*NULL from dual // this is NULL because "unknown"*300 = unknown
```

In The same way, the evaluation of the predicate `300*NULL<100` is NULL, consequently

`SELECT 1 from dual where 300*NULL<100` returns no row

It is important to understand how Oracle deals with NULL, because otherwise we can slip subtle but misleading SQL statements, for example:

```
SELECT COUNT(*) FROM DEPARTMENTS WHERE ID  NOT IN (SELECT DEPARMENT_ID FROM EMPLOYEES)
```

This query will work as expected as long as all employees are “assigned” to a department, if one of the employees has a NULL value for DEPARTMENT_ID , then this query will RETURN ALWAYS 0, which is incorrect. Why? Because in this case ORACLE cannot resolve the predicate ID NOT IN(…)

Summarizing, watch out when using NOT IN if one of the values of the subquery may be NULL, as it could behaves unexpectedly

In any case, to workaround this you can eliminate the NULL ambiguity using for example NVL

```
SELECT COUNT(*) FROM DEPARTMENTS WHERE ID  NOT IN (SELECT NVL(DEPARMENT_ID,999999)
FROM EMPLOYEES)
```

Or even better to rewrite the statement with NOT EXISTS

NULL when used with logical operators

NULL	AND	TRUE	=	NULL
NULL	AND	FALSE	=	FALSE
NULL	OR	TRUE	=	TRUE
NULL	OR	FALSE	=	NULL

For example:

`SELECT 'HERE' from dual WHERE (300*NULL<100 or 1=1)` =>SHOWS ‘HERE’ because NULL OR FALSE = true

`SELECT 'HERE' from dual WHERE NOT(300*NULL<100 AND 1<>1)` =>shows ‘HERE’ because the expression in the NOT parenthesis is FALSE (remember NOT(FALSE) =true), the expression is false because NULL and FALSE =FALSE

COLUMN ALIAS

- Each expression in the SELECT list may optionally be followed by a column alias.
- A column alias is placed after the expression in the SELECT list, separated by the optional keyword AS and a required space.
- If the column alias is enclosed in double quotes, it can include spaces and other special characters.

```
SELECT first_name, salary*12+bonus ‘ANNUAL SALARY + BONUS’ …. // wrong
SELECT first_name, salary*12+bonus “ANNUAL SALARY + BONUS” …. // right
```

- If the column alias is not enclosed in double quotes, it is named according to the standard rules for naming database objects.
- The column alias exists within the SQL statement and does not exist outside of the SQL statement.
- The column alias will become the new header in the output of the SQL statement.
- The column alias can be referenced within the ORDER BY clause, but nowhere else—such as WHERE, GROUP BY, or HAVING.

```
SELECT SHIP_ID, PROJECT_COST, PROJECT_NAME "The Project", DAYS FROM PROJECTS ORDER BY SHIP_ID
DESC, "The Project", 2;
```

Restricting and Sorting Data (C5)

Limit the Rows That Are Retrieved by a Query: WHERE, Boolean operators, LIKE, IN, BETWEEN, IS (NOT) NULL

- The WHERE clause comes after the FROM clause
- WHERE identifies which rows are to be included in the SQL statement
- WHERE is used by SELECT, UPDATE, and DELETE, WHERE is an optional clause
- Expressions form the building blocks of the WHERE clause
- Expressions may include column names, literal values, and as you'll see in Chapter 6. The WHERE clause compares expressions to each other using comparison operators and determines if the comparisons are true or false
- Boolean operators may separate each comparison to create a complex series of evaluations. Collectively, the final result for each row in the table will either be true or false; if true, the row is returned; if false, it is ignored
- The Boolean operators are AND, OR, and NOT

IMPORTANT: The rules of Boolean operator precedence require that NOT be evaluated first, then AND, and then OR

```
SELECT * FROM EMPLOYEE WHERE NAME=’BOB’ OR AGE=23 AND ORIGIN=’SPAIN’
```

This is similar to

```
SELECT * FROM EMPLOYEE WHERE NAME=’BOB’ OR (AGE=23 AND ORIGIN=’SPAIN’)
```

The rules of Boolean operator precedence require that NOT be evaluated first, then AND, and then OR

- Parentheses can override any Boolean operator precedence
- When comparing date datatypes: earlier date values are considered "less" than later dates, so anything in January will be "less than" anything in December of the same year
- When comparing character datatypes, the letter 'a' is less than the letter 'z', upper case letters are "lower than" lower case letters, and the character representation of '3' is greater than the character representation of '22', even though the results would be different if they were numeric datatypes
- LIKE can be used to activate wildcard searches. *The two wildcard symbols are : the underscore (_), representing a single character and the percent sign(%) representing zero or more characters*

- IN can be used to compare a single expression to a set of one or more expressions

```
SELECT PORT_NAME FROM PORTS WHERE COUNTRY IN ('UK', 'USA', 'Bahamas');
```

- BETWEEN can be used to see if a particular expression's value is within a range of values. BETWEEN is inclusive, not exclusive, so that BETWEEN 2and 3 includes the numbers 2 and 3 as part of the range

```
SELECT PORT_NAME FROM PORTS WHERE CAPACITY BETWEEN 3 AND 4;
```

- Operators ALL, ANY= SOME:
    - The `ALL` comparison condition is used to compare a value to a list or subquery. It must be preceded by =, !=, >, <, <=, >= and followed by a list or subquery.

```
SELECT empno, sal FROM   emp WHERE  sal > ALL (2000, 3000, 4000);
SELECT e1.empno, e1.sal FROM   emp e1 WHERE  e1.sal > ALL (SELECT e2.sal FROM   emp e2
WHERE  e2.deptno = 20);
```

    - ANY = SOME , they are synonyms, as example same queries that for ALL but using ANY
    - Note: this operators can be simulated easily with EXISTS / NOT EXISTS and IN/NOT IN operators

- Use IS NULL / IS NOT NULL when testing a column to see if its value is NULL, It is not used <column>=NULL

`SELECT PORT_NAME FROM PORTS`
`WHERE CAPACITY = NULL;`
	This SQL statement will never retrieve any rows, never ever, never ever ever

Sort the Rows That Are Retrieved by a Query: ORDER BY XXX ASC/DESC

- ORDER BY is an optional clause used to sort the rows retrieved in a SELECT statement
- If used, ORDER BY is always the last clause in the SELECT statement
- ORDER BY uses expressions to direct the sorting order of the result set of the SELECT statement
- Each expression is evaluated in order, so that the first item in the ORDER BY will do the initial sort of output rows, the second item listed will sort any rows that share identical data for the first ORDER BY element, and so on
- ORDER BY can sort by columns in the table, regardless of whether the columns appear in the SELECT statement's SELECT list or not
- ORDER BY can also sort by expressions of any kind, following the same rules of expressions that you've seen with the WHERE clause and the SELECT list
- All sorts default to ascending order(low->up), which can be specified with the optional keyword ASC
    - Numeric data is sorted by default in ascending order, from lower numbers to higher
    - Character data is sorted by default in ascending order, from 'A' to 'Z'
    - Date data is sorted by default in ascending order, from prior dates to later dates

- Sort order can be changed to descending order with the keyword DESC

- NULLS FIRST /LAST: In ascending order, NULL values will always be sorted last and thus appear after the other data. In descending order NULL values will appear first. The sort order of NULL values can be overridden using the *NULLS FIRST/LAST* clause.

```
SELECT stuff from mytab order by dept_id asc nulls first; -- nulls will appear first
```

- ORDER BY can identify columns by column alias, or by position within the SELECT list

```
SELECT SHIP_ID, PROJECT_COST, PROJECT_NAME "The Project", DAYS FROM PROJECTS ORDER BY SHIP_ID
DESC, "The Project", 2;
```

NOTE: 2 is the second column on the SELECT statement, in this case PROJECT_COST. This is called Reference by position

Using Single-Row Functions to Customize Output (C6). SEE APENDIX

Describe Various Types of Functions That Are Available in SQL

- SQL functions accept one or more input parameters. A few take no parameters
- Each function returns one value; no more, no less
- SQL functions perform tasks of various kinds
- Functions can be included anywhere a SQL expression can be included, provided that the rules of datatypes are respected
- Functions can be included in the WHERE clause of the SELECT, UPDATE, and DELETE statements
- Functions can be included in the SELECT expression list, INSERT value list, and UPDATE SET clause

The quote operator

There are several methods to put the quote into the string.

```
   1) SELECT 'This '' is quote' FROM dual; -- escaping the quote

   2) SELECT 'This ' || CHR(39) || ' is quote' FROM dual;

   3) DECLARE
       s_quote VARCHAR2(1) := '''' ;
      BEGIN
       dbms_output.put_line('This ' || s_quote || ' is quote' ) ;
      END;
```

And, finally, Oracle 10g has added new feature: quote operator.

```
      SELECT q'[This ' IS quote]' from dual ;
```

The general form is **q'X string X'**. Here X is just some character. If the brackets are used, Oracle expects the closing bracket for the end of the string.

Here are the additional examples that has the same result:

```
      SELECT q'(This ' IS quote)' from dual ;
      SELECT q'|This ' is quote|' FROM dual ;
      SELECT q'#This ' IS quote#' from dual ;
      SELECT q'#This ' is quote#' FROM dual ;
      SELECT q'?This ' IS quote?' from dual ;
      SELECT q'TThis ' is quoteT' FROM dual ;
```

Character functions: UPPER, LOWER, INITCAP, PAD, LPAD, RPAD, TRIM, LTRIM, RTRIM,INSTR, SUBSTR, LENGTH, CONCAT,SOUNDEX

Upper/lower(string)	Returns **string** with all letters in uppercase.

	SELECT EMPLOYEE_ID FROM EMPLOYEES WHERE UPPER(LAST_NAME) = 'MCGILLICUTTY';
initcap(string)	Returns **string**, with the first letter of each word in uppercase and all other letters in lowercase. initcap('tech on thenet'); => Tech On Thenet initcap('GEORGE BURNS'); => George Burns
Lpad/rpad(string1, number [,string2])	Returns **string1**, left/right-padded to length **number** using characters in **string2**; **string2** defaults to a single blank. lpad('tech', 7); => ' tech' lpad('tech', 2); => 'te' lpad('tech', 8, '0'); => '0000tech' lpad('tech on the net', 15, 'z');=>'tech on the net' lpad('tech on the net',16, 'z');=>'ztech on the net'
trim({[LEADING | TRAILING | BOTH] trim_char | trim_char } FROM trim_source})	removes all specified characters either from the beginning or the ending of a string. TRIM(' tech ') => 'tech' TRIM(' ' from ' tech ') => 'tech' TRIM(leading '0' from '000123') => '123' TRIM(trailing '1' from 'Tech1') => 'Tech' TRIM(both '1' from '123Tech111')=> '23Tech'
Ltrim/rtrim(string[, set])	Removes all characters in **set** from the left / right of **string**. **Set** defaults to a single blank. LTRIM(' tech'); => 'tech' LTRIM(' tech', ' '); => 'tech' LTRIM('000123', '0'); =>'123' LTRIM('123123Tech', '123'); =>'Tech' LTRIM('123123Tech123', '123'); =>'Tech123' LTRIM('xyxzyyyTech', 'xyz'); =>'Tech' LTRIM('6372Tech', '0123456789'); =>'Tech'
instr(string1, string2, start_at, occurrence)	Returns the location (a number) of a substring in a string. INSTR('Welcome to PSOUG.org', 'o') => 5 INSTR('Welcome to PSOUG.org', 'o', 1, 1) => 5 INSTR('Welcome to PSOUG.org', 'o', 1, 2) => 10 INSTR('Welcome to PSOUG.org', 'o', 1, 3) =>18 INSTR('Welcome to PSOUG.org', 'o', -2, 2) => 10 (Count back 2, then find the 2nd 'o' backwards) INSTR('MATARATAS LARATA','TA', -3, 2)) => 3
substr(extraction_string [FROM starting_position] [FOR length])	SUBSTR('PEPE PEREZ',6) => PEREZ SUBSTR('PEPE PEREZ',6,3) => PER SUBSTR('abcdefg',-5,4) => cdef
length(string)	Returns the integer length of **string**, or null if **string** is null. LENGTH('Supercalifragilisticexpialidocious') => 34
concat(string1,string2)	Returns **string1** concatenated with **string2**. It is equivalent to the concatenation operator (||). SELECT CONCAT('Hello, ', 'world!') FROM DUAL;

	SELECT 'Hello, ' || 'world!' FROM DUAL;

soundex(string)	Returns a character string containing the phonetic representation of **string**. This function allows words that are spelled differently but sound alike in English to be compared for equality. A single SOUNDEX value is relatively worthless. But two combined together can be surprisingly helpful. The reason is that similar sounding words tend to generate the same SOUNDEX pattern.
	SELECT SOUNDEX('Worthington'), SOUNDEX('Worthen') FROM DUAL; => W635 W635
	Notice how the two different words produce the same SOUNDEX pattern. That means we can do queries like this:
	SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME
	FROM EMPLOYEES
	WHERE SOUNDEX(LAST_NAME) = SOUNDEX('Worthen');

Numeric functions : ROUND, REMAINDER, MOD, TRUNC,

round(number,decimal)	Returns **number** rounded to **decimal** places right of the decimal point. When **decimal** is omitted, **number** is rounded to 0 places. Note that **decimal**, an integer, can be negative to round off digits left of the decimal point. round(123.456,-1) => 120 round(123.456,-2) => 100 ROUND(12.355143, 2) => 12.36 ROUND(259.99,-1) => 260 round(123.456) => 123
remainder(n1,n2)	REMAINDER(9,3),REMAINDER(10,3),REMAINDER(11,3) => 0 1 -1
mod(dividend,divider)	Returns remainder of **dividend** divided by **divider** ; returns the **dividend** if **divider** is 0. MOD(9,3), MOD(10,3), MOD(11,3) => 0 1 2
trunc (base [,number])	Returns **base** truncated to **number** decimal places. When **number** is omitted, **base** is truncated to 0 places. **Number** can be negative to truncate (make zero) **number** digits left of the decimal point. trunc(1234.5678,2) => 1234.56 trunc(1234.5678,-2) => 1200 trunc(1234.5678,-1) => 1230 trunc(1234.5678) => 1234

Other functions: CASE, NVL, NVL2, DECODE, LEAST/GREATEST, LEAST, GREATEST, COALLESCE, NULLIF

CASE expression1 WHEN condition1 THEN result1 WHEN condidition2 THEN result2 ELSE resultfinal END	SELECT SHIP_NAME, CAPACITY,
	CASE CAPACITY
	   WHEN 2052 THEN 'MEDIUM'
	   WHEN 2974 THEN 'LARGE'
	END AS "SIZE"
	FROM SHIPS WHERE SHIP_ID <= 4;
	Codd Crystal 2052 MEDIUM
	Codd Elegance 2974 LARGE
	Codd Champion 2974 LARGE
	Codd Victorious 2974 LARGE

nvl(expression1, expression2)	If **expression1** is null, **expression2** is returned in the place of a null value. Otherwise, **expression1** is returned. The expressions may be any datatype. `NVL(NULL,'HERE') => HERE`
nvl2(expression1, expression2, expression3)	Similar to **NLV**, except that if **expression1** is not null, **expression2** is returned. If **expression1** is null, **expression3** is returned. The expressions may be any datatype, except **LONG**. `NVL2(NULL,'FIRST NULL', 'SECOND NULL')=> SECOND NULL`
nullif(e1,e2)	If e1 and e2 are the same then return NULL, otherwise returns e1 `NULLIF('A','A') => null`

decode(expr search , result [,. n] [,default])	Compares **expr** to the search value; if **expr** is equal to a search, returns the result. Without a match, **DECODE** returns default, or **NULL** if default is omitted.
	`SELECT decode(code,`
	`'GGP','United Kingdom',`
	`'USD','United States',`
	`'DEFAULT') from currency`

least / greatest(expression [,...n])	Returns the least of the list of **expressions.** `least(4,5) => 4`

coalesce	returns the first non-null expression in the list. If all expressions evaluate to null, then the **COALESCE function** will return null.
	is equivalent to: CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END
	`coalesce(null,null,4,null) => 4`
	NOTE: COALESCE expects consistent datatypes.
	`coalesce('a',sysdate) => ERROR`

Conversion functions: TO_CHAR, TO_ NUMBER, TO_DATE,TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_DSINTERVAL, TO_YMINTERVAL
This functions takes a parameter to define the format

Date Format Elements	
Parameter	***Explanation***
YEAR	Year, spelled out
YYYY	4-digit year
YYY YY Y	Last 3, 2, or 1 digit(s) of year.
IYY IY I	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	4-digit year based on the ISO standard
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).
MM	Month (01-12; JAN = 01).
MON	Abbreviated name of month.

MONTH	Name of month, padded with blanks to length of 9 characters.
RM	Roman numeral month (I-XII; JAN = I).
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
D	Day of week (1-7).
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DY	Abbreviated name of day.
J	Julian day; the number of days since January 1, 4712 BC.
HH	Hour of day (1-12).
HH12	Hour of day (1-12).
HH24	Hour of day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
FF	Fractional seconds.

Number Format Elements

Element	***Example***	***Description***
, .	1,234.56	Commas and decimal points
$	$123.45	Leading dollar sign.
0	0012.34	Leading or trailing 0.
9	123	Any digit.
B	B123	Leading blank for integers.
C	C123	The ISO currency symbol defined in the NLS_ISO_CURRENCY parameter.
D	123D99	The current decimal character defined in the NLS_NUMERIC_CHARACTERS parameter. The default value is a period.
EEEE	1.2EEE	Returns a value in scientific notation.
G	9G123	Returns the group separator (e.g., a comma).
L	L123	Returns the local currency symbol.
MI	123MI	negative value with trailing minus sign; returns positive value with a trailing blank.
PR	123PR	The negative values in angle brackets.
RN	I	values in Roman numerals, uppercase.
rn	i	values in Roman numerals, lowercase.
S (prefix)	S1234	negative values with a leading minus sign, positive values with a leading positive sign.
S (suffix)	1234S	negative values with a trailing minus sign, positive values with a trailing positive sign.

Number Format Elements		
Element	***Example***	***Description***
TM	TM	The text minimum number format model returns the smallest number of characters possible.
U	U123	the Euro currency symbol or the NLS_DUAL_CURRENCY parameter.
V	123V99	a value multiplied by 10n, where n is the number of 9s after the V.
X	XXXX	the hexadecimal value.

TO_CHAR(value, [format_mask], [nls_language])	**NUMBERS:** TO_CHAR(1210.73, '9999.9') => '1210.7' TO_CHAR(1210.73, '9,999.99') => '1,210.73' TO_CHAR(1210.73, '$9,999.00') => '$1,210.73' TO_CHAR(1123.90, '$9,999') => '$1.124' TO_CHAR(11235.90, '$9,999') => '#######' TO_CHAR(21, '000099') => '000021' **DATES**: TO_CHAR(sysdate, 'yyyy/mm/dd');=> '2003/07/09' TO_CHAR(sysdate, 'Month DD, YYYY');=> 'July 09, 2003' TO_CHAR(sysdate, 'FMMonth DD, YYYY'); => 'July 9, 2003' TO_CHAR(sysdate, 'MON DDth, YYYY'); => 'JUL 09TH, 2003' TO_CHAR(sysdate, 'FMMON DDth, YYYY'); => 'JUL 9TH, 2003' TO_CHAR(sysdate, 'FMMon ddth, YYYY'); => 'Jul 9th, 2003'
TO_NUMBER(string1, [format_mask], [nls_language])	TO_NUMBER('1210.73', '9999.99') => the number 1210.73 TO_NUMBER('546', '999') => the number 546 TO_NUMBER('23', '99') => the number 23
TO_DATE(string1, [format_mask], [nls_language])	TO_DATE('2003/07/09', 'yyyy/mm/dd')=>date of July 9, 2003 TO_DATE('070903', 'MMDDYY') => date of July 9, 2003 TO_DATE('20020315', 'yyyymmdd')=> date value of Mar 15, 2002
TO_TIMESTAMP(string1, [format_mask] ['nlsparam'])	TO_TIMESTAMP('2003/12/13 10:13:18', 'YYYY/MM/DD HH:MI:SS') => '13-DEC-03 10.13.18.000000000 AM' as a timestamp value. TO_TIMESTAMP('2003/DEC/13 10:13:18', 'YYYY/MON/DD HH:MI:SS') =>'13-DEC-03 10.13.18.000000000 AM' as a timestamp value.
TO_TIMESTAMP_TZ(string1 , [format_mask] ['nlsparam'])	TO_TIMESTAMP_TZ('2003/12/13 10:13:18 -8:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM') => '13-DEC-03 10.13.18.000000000 AM -08:00' as a timestamp with time zone value. TO_TIMESTAMP_TZ('2003/DEC/13 10:13:18 -8:00', 'YYYY/MON/DD HH:MI:SS TZH:TZM') => '13-DEC-03 10.13.18.000000000 AM -08:00' as a timestamp with time zone value
TO_DSINTERVAL(character [nls_parameter])	TO_DSINTERVAL('150 08:30:00') => '+000000150' TO_DSINTERVAL('80 12:30:00') => '+000000080' TO_DSINTERVAL('95 18:30:00') => '+000000095'

TO_YMINTERVAL(character)	TO_YMINTERVAL('03-11') => 3 years 11 months as an INTERVAL YEAR TO MONTH type TO_YMINTERVAL('01-05') => 1 year 5 months as an INTERVAL YEAR TO MONTH type TO_YMINTERVAL('00-01') => 0 years 1 month as an INTERVAL YEAR TO MONTH type

Date functions: ADD_MONTHS, AT TIME ZONE, AT LOCAL, CAST (E AS D), current_date/ current_timestamp/localtimestamp/sessiontimezone, systimestamp/dbtimezone/sysdate, sys_extract_utc, extract(fm FROM e),from_tz,last_day, months_between, next_day, new_time, numtodsinterval, numtoyminterval, round, trunc

add_months(date, int). Note:there is not substract_months	Returns the date ***date*** plus ***int*** months. ADD_MONTHS('01-Aug-03', -3) -> '01-May-03'
at time zone / at local(expression)	TO_TIMESTAMP('2012-MAY-24 02:00:00','RRRR-MON-DD HH24:MI:SS') AT TIME ZONE DBTIMEZONE "DB Time" => 24-MAY-12 02.00.00.000000000 +01:00 TO_TIMESTAMP_TZ('2003/12/13 10:13:18 -8:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM') at local =>13-DEC-03 19.13.18.000000000 EUROPE/BERLIN NOTE: without AT LOCAL, the result would be: 13-DEC-03 10.13.18.000000000 -08:00
Cast(e AS d)	Parameters: e is an expression; d is a datatype. Process: Converts e to d. Particularly useful for converting text representations of datetime information into datetime formats, particularly TIMESTAMP WITH LOCAL TIME ZONE. Output: A value in the d datatype. CAST('19-JAN-10 11:35:30' AS TIMESTAMP WITH LOCAL TIME ZONE) =>19-JAN-10 11.35.30.000000000 AM CAST('19-JAN-10 11:35:30' AS TIMESTAMP) => 19-JAN-10 11.35.30.000000000 NOTE: Cast is a general conversion function, for example, the following converts from date to varchar CAST('22-Aug-2003' AS varchar2(30)) => convert the date (ie: 22-Aug-2003) into a varchar2(30) value.
1. *current_date,* 2. *current_timesta*	First group returns date information reflecting session timing, while the second group reflects timing of DB server

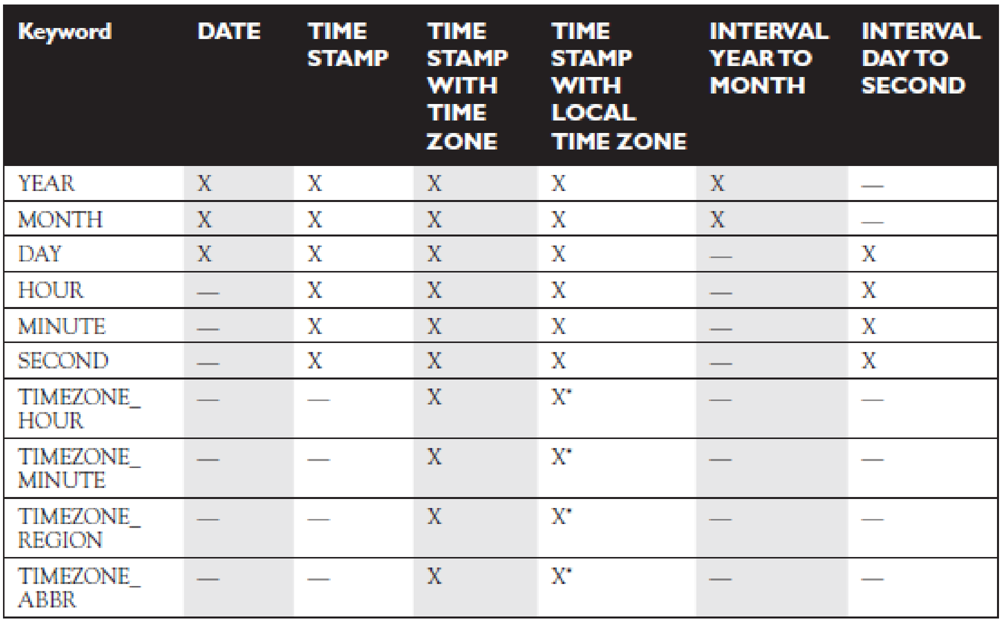
mp *3. localtimestamp,* *4. sessiontimezone* *1. systimestamp* *2. dbtimezone* *3. sysdate*	Note: the difference between current_timestamp and localtimestamp is the current_timestamp returns a TIMESTAMP WITH TIME ZONE and the localtimestamp return TIMESTAMP

sys_extract_utc	extracts the UTC (Coordinated Universal Time—formerly Greenwich Mean Time) from a datetime value with time zone offset or time zone region name.
	`SYS_EXTRACT_UTC(TIMESTAMP '2000-03-28 11:30:00.00 -08:00')`
	`=> 28-MAR-00 07.30.00 PM`

extract (fm FROM e)

extracts and returns the value of a specified datetime field from a datetime or interval value expression

```
EXTRACT(MINUTE FROM TO_TIMESTAMP('2009-10-11 12:13:14', 'RRRR-MM-DD HH24:MI:SS'))
=>13
```



Keyword	DATE	TIME STAMP	TIME STAMP WITH TIME ZONE	TIME STAMP WITH LOCAL TIME ZONE	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
YEAR	X	X	X	X	X	—
MONTH	X	X	X	X	X	—
DAY	X	X	X	X	—	X
HOUR	—	X	X	X	—	X
MINUTE	—	X	X	X	—	X
SECOND	—	X	X	X	—	X
TIMEZONE_HOUR	—	—	X	X*	—	—
TIMEZONE_MINUTE	—	—	X	X*	—	—
TIMEZONE_REGION	—	—	X	X*	—	—
TIMEZONE_ABBR	—	—	X	X*	—	—

from_tz(ts,tz)	Transforms ts, a TIMESTAMP value, and tz, a character value representing the time zone, into a value of the datatype TIMESTAMP WITH TIME ZONE.
	The second parameter, tz, can be in one of two formats: either the format of 'TZH:TZM', where TZH and TZM are time zone hours and time zone minutes; or the format of a character expression that results in a string in the TZR with optional TZD format
	Output: A value of the TIMESTAMP WITH TIME ZONE datatype.
	
	`FROM_TZ( TIMESTAMP '2012-10-12 07:45:30', '+07:30')`
	`=>12-OCT-12 07.45.30.000000000 AM +07:30`

last_day(date)	Returns the date of the last day of the month that contains **date.**
	`LAST_DAY('14-FEB-11'), LAST_DAY('20-FEB-12')`
	`=>28-FEB-11, 29-FEB-12`

months_between (date1, date2)	Returns number of months between dates **date1** and **date2**. When **date1** is later than **date2**, the result is positive. If it is earlier, the result is negative.
	`MONTHS_BETWEEN('01-JAN-12', '01-FEB-12') => -1`
	`MONTHS_BETWEEN('01-JAN-12', '01-MAR-12') => -2`
	`MONTHS_BETWEEN('10-AUG-14', '10-JUL-14') => 1`

next_day(date, string)	Returns the date of the first weekday named by **string** that is later than **date**. `NEXT_DAY('31-MAY-11','Saturday') => 04-JUN-11`

new_time(d,t1,t2)	Parameters: d is a DATE datatype and is required. t1 and t2 are time zone Indications, for a given value of d, NEW_TIME translates the time d according to the offset specified between t1 and t2. In other words, t1 is assumed to be the time zone in which d is recorded
	`TO_CHAR( NEW_TIME(`
	`TO_DATE('1983-JAN-03 14:30:56',`
	`'RRRR-MON-DD HH24:MI:SS'),'AST','HST') ,'DD-MON-RR HH:MI:SS')`
	`=>03-JAN-83 08:30:56`

numtodsinterval (number, `string')	Converts **number** to an **INTERVAL DAY TO SECOND** literal, where **number** is a number or an expression resolving to a number, such as a numeric datatype column. `NUMTODSINTERVAL(36,'HOUR') => 1 12:0:0.0`
numtoyminterval (number, `string')	Converts **number** to an **INTERVAL DAY TO MONTH** literal, where **number** is a number or an expression resolving to a number, such as a numeric datatype column. `NUMTOYMINTERVAL(27,'MONTH') => 2-3`

round (date[, format])	Returns the **date** rounded to the unit specified by the format model **format**. When **format** is omitted, **date** is rounded to the nearest day.
	`SYSDATE TODAY,`
	`ROUND(SYSDATE,'MM') ROUNDED_MONTH,`
	`ROUND(SYSDATE,'RR') ROUNDED_YEAR`
	`=> 16-AUG-13, 01-SEP-13, 01-JAN-14`

trunc (date [, format])	Returns **date** with any time data truncated to the unit specified by **format**. When **format** is omitted, **date** is truncated to the nearest whole day.
	`SYSDATE TODAY,`
	`TRUNC(SYSDATE,'MM') TRUNCATED_MONTH,`
	`TRUNC(SYSDATE,'RR') TRUNCATED_YEAR FROM DUAL;`
	`16-AUG-13, 01-AUG-13, 01-JAN-13`

NOTE:

- TO_CHAR(SYSDATE, 'DD-MON-RRRR HH:MM:SS') => MM it is not minute, it is MONTH !!!

- These functions take into account the session settings, for example, if the NLS_TERRITORY is set to USA, then NEXT_DAY(any_date, 1) will be the date of the next Sunday, while in other NLS_TERRITORY could be MONDAY

Reporting Aggregated Data Using the Group Functions (C7)

Group Functions = Aggregate Functions = Multirow Functions

- Multirow functions return one value for every set of zero or more rows considered within a SELECT statement

- There are group functions to determine minimum and maximum values, calculate averages, and more

- Group functions can be used to determine rank within a group of rows

- Aggregate and scalar data cannot be included in the same SELECT statement's SELECT list

- WHERE clause can not have an aggregate functions,

THE FOLLOWING SELECT IS WRONG!!!!: `SELECT customer_id,COUNT(order_id) FROM orders WHERE COUNT(order_id)>3 AND order_date BETWEEN ADD_MONTHS(SYSDATE,-6) AND SYSDATE GROUP BY customer_id;`

- You can nest aggregate functions two levels deep, meaning you can use an aggregate as a parameter into another aggregate, provided you do not go any further-you cannot, for example, pass a first aggregate into a second aggregate that is in turn passed to a third aggregate.

- The COUNT function counts occurrences of data, as opposed to the SUM function, which adds up numeric values. NOTE:

GROUP FUNCTIONS COUNT AND DISTINCT

- COUNT function returns the number of rows for which the expression evaluates to a non-null value. For example, suppose in table vfn_tranche there 120 rows having volatility_required_id to some **non NULL value**, and 789 with the null value.

- NOTE that COUNT will never return NULL, if no values, then it will return 0, this is not the same with other aggregates functions , such as AVG, MIN, MAN, or SUM.

```
SELECT count(nvl2(volatility_required_id,1,null)) from vfn_tranche => 120
SELECT count(nvl2(volatility_required_id,null,1)) from vfn_tranche => 789
```

- DISTINCT = UNIQUE, can be used in a SELECT statement to list unique data sets: SELECT DISTINCT LAST_NAME, FIRST_NAME FROM EMPLOYEES; => returns different "last Name" and "First Name" combinations existing in the table EMPLOYEES. **IT DOES COUNT NULL**

- COUNT(DISTINCT**(..)) DOES NOT INCLUDE NULL VALUES**, in this case, DISTINCT can return NULL but count won't include it

- COUNT(*) : it returns the number of rows of the query, regardless the NULL values, so, it is contradictory with the count(column) , but it is the way oracle works!!!

In summary:

```
SELECT count(distinct ...)"  IS NOT  a "SELECT count(*) from (SELECT distinct ...)"
```

- The MIN and MAX functions can operate on date, character, or numeric data
- A The AVG and MEDIAN functions can perform average and median calculations, and they can ignore NULL values in their computations

RANK as AGGREGATE OR ANALYTIC FUNCTIONS

As aggregate

Look at the following table

(ID, NAME, SALARY, DEPARTAMENT)
(1,’Bill’, 50000,’Marketing’)
(2,’John’, 70000,’Production’)
(3,’Steve’, 60000,’Production’)
(4,’Paul’, 85000,’Management’)

It is clear that in terms of salary, the first position (rank 1) is for Paul, the second (rank 2) for John and so on…

Now, a new employee (‘Larry’) wants to join the company and his salary is determined to be 65000, what would be his rank in terms of salary? That is precisely what function RANK as aggregate determines, the answer is 3 (above Steve, below John)

```
SELECT RANK(65000) WITHIN GROUP (ORDER BY salary) from employees; // RESULT IS 3
```

Imagine now the table EMPLOYEE as follows:

(1,’Bill’, 50000,’Marketing’)
(2,’John’, 70000,’Production’)
(3,’Steve’, 60000,’Production’)
(4,’Paul’, 85000,’Management’)
(5,’Charles’, 70000,’Assembly’)

In this scenario, What would the rank of Larry? 4

```
SELECT RANK(65000) WITHIN GROUP (ORDER BY salary) from employees; // RESULT IS 4
```

This is because if you have 2 items at rank 2, the next rank listed would be ranked 4

DENSE_RANK allows no to skip ranks in case of having ranks with multiple items, for example

```
SELECT DENSE_RANK(65000) WITHIN GROUP (ORDER BY salary) from employees; // RESULT IS 3
```

As analytic

The rank is calculated respective to the other rows, for example the following query returns the range respective to other rows having the same department (note the syntax of RANK as analytic is quite different to the one for aggregate)

```
SELECT NAME, SALARY, RANK() OVER (PARTITION BY DEPARTAMENT ORDER BY SALARY) FROM
EMPLOYEES WHERE DEPARTMENT = 'PRODUCTION';
```

```
(‘Charles’, 70000, 1)
(‘Steve’,60000,2)
```

Now, imagine we need to know what is the second top salary of each department. The following query would give us this information

```
SELECT * FROM (SELECT NAME, SALARY, RANK() OVER (PARTITION BY DEPARTAMENT ORDER BY
SALARY) RANK FROM EMPLOYEES) WHERE RANK=2;
```

This example shows how useful these functions are, this query without using RANK functions would be much more difficult

DENSE_RANK as analytic works in the same manner as the aggregate version

FUNCTIONS FIRST, LAST

Related to RANGE, For a given range of sorted values, they return either the first value (FIRST) or the last value (LAST) of the population of rows defining e1, in the sorted order. For example:

```
SELECT MAX(SQ_FT) KEEP (DENSE_RANK FIRST ORDER BY GUESTS)
"Largest"
FROM SHIP_CABINS;
Largest
----------------------
225
```

Group Data by Using the GROUP BY / HAVING Clause

- The GROUP BY clause is an optional clause in the SELECT statement in which you can specify how to group rows together in order to process them as a group
- The row groups identified by GROUP BY can have aggregate functions applied to them, so that the final result of the SELECT is not a single aggregate value, but a series of aggregate function results, one per group
- GROUP BY can specify columns in a table, which will have the effect of grouping rows in the SELECT that share the same values in those columns Whatever you specify in the GROUP BY may also be included in the SELECT statement’s SELECT list—but this is not required
- The effect of GROUP BY on a column is to change that column into an “aggregate” value; in other words, by grouping rows that have common data for a given column, and by specifying the column in the GROUP BY, you elevate the information in the column from “scalar” to “aggregate” for the purposes of that SELECT statement
- GROUP BY can specify one or more expressions
- The GROUP BY clause is processed by SQL before the SELECT list. Therefore it doesn’t recognize column aliases created in the SELECT list—this applies to ROLLUP and CUBE as well.

```
SELECT share_class_id  from nav group by share_class_id -- wrong
SELECT share_class_id sc from nav group by sc -- wrong
```

Include or Exclude Grouped Rows by Using the HAVING Clause

- The HAVING clause is an optional clause for the SELECT statement that works in coordination with GROUP BY
- You cannot use HAVING unless GROUP BY is present
- HAVING specifies groups of rows that will be included in the output of the SELECT statement
- HAVING performs the same function for GROUP BY that WHERE performs for the rest of the SELECT statement
- HAVING specifies groups using the same expression logic and syntax that WHERE would use, is OK the following:

  ```
  SELECT order_id FROM order_items GROUP BY order_id HAVING SUM(unit_price*quantity) =
  (SELECT MAX(SUM(unit_price*quantity)) FROM order_items GROUP BY order_id;
  ```

- The GROUP BY and HAVING clauses are not required, but if used, they must follow the WHERE clause and precede the ORDER BY clause
- While GROUP BY typically precedes HAVING in common practice, this is not required, and they can appear in either order

  ```
  SELECT ROOM_STYLE, ROOM_TYPE, TO_CHAR(MIN(SQ_FT),'9,999') "Min"
  FROM SHIP_CABINS WHERE SHIP_ID = 1 GROUP BY ROOM_STYLE, ROOM_TYPE HAVING
  ROOM_TYPE IN ('Standard', 'Large') OR MIN(SQ_FT) > 1200 ORDER BY 3;
  ```

Displaying Data from Multiple Tables (C8)

Write SELECT Statements to Access Data from More Than One Table Using Equijoins and Non-Equijoins/View Data That Generally Does Not Meet a Join Condition by Using Outer Joins

- A join is any SELECT statement that SELECTs from two or more tables

- A join will connect rows in one table with rows in another table

Joins are characterized in many ways. One way a join is defined is in terms of whether it is an inner join or an outer join. Another issue is that of equijoins and non-equijoins. These descriptions are not mutually exclusive descriptions.

Equijoins versus Non-Equijoins:

- The equijoin identifies a particular column in one table's rows, and relates that column to another table's rows, and looks for equal values in order to join pairs of rows together, in other words, if it uses the equal operator

  ```
  SELECT e.first_name, d.department_name FROM employees e INNER JOIN
  departments d ON e.department_id = d.department_id
  ```

- The non-equijoin differs from the equijoin in that it doesn't look for exact matches but instead looks for relative matches, such as one table's value that is between two values in the second table, in other words, it if does not use the equal operator

  ```
  SELECT zip_codes.zip_code, zones.ID AS zip_zone, zones.low_zip,
  zones.high_zip FROM zones INNER JOIN zip_codes ON zip_codes.zip_code
  BETWEEN zones.low_zip AND zones.high_zip
  ```

Inner versus Outer joins

- **The inner join** compares a row in one table to rows in another table and only produces output from the first row if a matching row in the second table is found

  ```
  SELECT SHIP_ID, SHIP_NAME, PORT_NAME
  FROM SHIPS INNER JOIN PORTS // INNER is OPTIONAL
  ON HOME_PORT_ID = PORT_ID
  ORDER BY SHIP_ID;
  ```

  Before we move on to outer joins, let's review an old variation to the syntax we just reviewed for an inner join. Here it is:

  ```
  SELECT S.SHIP_ID, S.SHIP_NAME, P.PORT_NAME
  FROM SHIPS S, PORTS P
  WHERE S.HOME_PORT_ID = P.PORT_ID
  ORDER BY S.SHIP_ID;
  ```

- **The outer join** compares rows in two tables and produces output whether there is a matching row or not

  - **the left outer** join shows all the rows in one table and only the matching rows in the second;

```
SELECT SHIP_ID, SHIP_NAME, PORT_NAME
FROM SHIPS LEFT OUTER JOIN PORTS
ON HOME_PORT_ID = PORT_ID
ORDER BY SHIP_ID;
```

- **the right outer** join does the same thing in reverse;

```
SELECT SHIP_ID, SHIP_NAME, PORT_NAME
FROM SHIPS RIGHT OUTER JOIN PORTS
ON HOME_PORT_ID = PORT_ID
ORDER BY SHIP_ID;
```

- **the full outer** join shows all rows in both tables one way or the other—either as a matched rowset or as a standalone row

```
SELECT SHIP_ID, SHIP_NAME, PORT_NAME
FROM SHIPS FULL OUTER JOIN PORTS
ON HOME_PORT_ID = PORT_ID
ORDER BY SHIP_ID;
```

COMPLEX QUERIES are allowed with these kinds of joins

```
SELECT p.product_name, i.item_cnt
FROM (SELECT product_id, COUNT (*) item_cnt
FROM order_items
GROUP BY product_id) i RIGHT OUTER JOIN products p ON i.product_id = p.product_id;
```

The natural join

does not name the connecting column but assumes that two or more tables have columns with identical names, and that these are intended to be the connecting, or joining, columns.NOTE: a natural JOIN is a inner JOIN

```
SELECT EMPLOYEE_ID, LAST_NAME, STREET_ADDRESS
FROM EMPLOYEES NATURAL JOIN ADDRESSES;
```

TYPICAL QUESTION: natural join forbids such table prefixes on join column names. Their use would result in a syntax error. However, table prefixes are allowed on other columns—but not the join columns in a natural join.

If the tables COUNTRIES and CITIES have two common columns named COUNTRY and COUNTRY_ISO_CODE, the following two SELECT statements are equivalent:

```
SELECT * FROM COUNTRIES NATURAL JOIN CITIES
SELECT * FROM COUNTRIES JOIN CITIES USING (COUNTRY, COUNTRY_ISO_CODE)
```

The following example is similar to the one above, but it also preserves unmatched rows from the first (left) table:

```
SELECT * FROM COUNTRIES NATURAL LEFT JOIN CITIES
```

NOTES ON JOINS

- The USING keyword can empower an inner, outer, or other join to connect based on a set of commonly named columns, in much the same fashion as a natural join. NOTE THAT THE USING COLUMNS CAN NOT USE PREFIX (ALIAS), JUST LIKE THE NATURAL JOIN

```
SELECT EMPLOYEE_ID, E.LAST_NAME, A.STREET_ADDRESS
FROM EMPLOYEES E LEFT JOIN ADDRESSES A
USING (EMPLOYEE_ID);
```

- Joins can connect two, three, or more tables

```
SELECT P.PORT_NAME, S.SHIP_NAME, SC.ROOM_NUMBER
FROM PORTS P JOIN SHIPS S ON P.PORT_ID = S.HOME_PORT_ID
JOIN SHIP_CABINS SC ON S.SHIP_ID = SC.SHIP_ID;
```

- The table alias only exists for the duration of the SQL statement in which It is declared, they are necessary to eliminate ambiguity in referring to columns of the same name in a join

```
SELECT EM.EMPLOYEE_ID, LAST_NAME, STREET_ADDRESS
FROM EMPLOYEES EM INNER JOIN ADDRESSES AD
ON EM.EMPLOYEE_ID = AD.EMPLOYEE_ID;
```

Join a Table to Itself by Using a Self-Join (recursive joins)

- The self-join connects a table to itself, typically connect a column in a table with another column in the same table

- Self-joins can otherwise behave as equijoins, non-equijoins, inner joins, and outer joins

```
SELECT A.POSITION_ID, A.POSITION, B.POSITION BOSS
FROM POSITIONS A LEFT OUTER JOIN POSITIONS B
ON A.REPORTS_TO = B.POSITION_ID
ORDER BY A.POSITION_ID;
```

Cartesian Product a.k.a CROSS-JOIN. NOTE: If any of the tables of the Cartesian product is empty (no rows), the result of the query will be EMPTY !!!!

- The Cartesian product is also known as a cross-join

- The cross-join connects every row in one table with every row in the other table

- It is created by SELECTing from two or more tables without a join condition of any kind

- The Cartesian product is rarely useful

```
SELECT * FROM VENDORS CROSS JOIN ONLINE_SUBSCRIBERS;
```

ANOTHER WAY OF ACHIEVING THIS:

```
SELECT * FROM VENDORS, ONLINE_SUBSCRIBERS;
```

IMPORTANT: Please Note the following:

- If any of the tables of the Cartesian product is empty (no rows), the result of the query will be EMPTY, in other words. If table vendors had no row, then the previous query won't return anything, if it is required to manage scenarios in which if one of the table is empty then result of the other table still appears, then use OUTER JOIN

- The syntax:

  ```
  SELECT * FROM VENDORS, ONLINE_SUBSCRIBERS;
  ```

  Is similar to the "old variation" of the inner join (see before) but without the WHERE CLAUSULE,

Using Subqueries to Solve Queries (C9)

Define Subqueries

- A subquery is a SELECT statement contained within a SQL statement
- The outer SQL statement is called the parent. The outermost level is the top level
- A top-level SQL statement containing a subquery may be a SELECT, INSERT,UPDATE, or DELETE, or else a CREATE TABLE or CREATE VIEW
- Subqueries may be nested within other subqueries
- Many subqueries could function as standalone queries. Some are correlated, meaning that they contain references that tie them into their parent queries

Describe the Types of Problems That Subqueries Can Solve

- A subquery can provide lookup data to assist a parent query in completing a WHERE clause or something comparable
- Subqueries can help combine multiple steps into a single query, reducing what otherwise might be several consecutive SQL statements into a single statement
- Subqueries in a CREATE TABLE or INSERT or UPDATE statement can draw from data from the database to populate database objects quickly
- Subqueries can name queries for subsequent reference
- Note that the different types of subqueries aren't mutually exclusive. A single type of subquery may fall into multiple categories of subqueries described in this chapter.

List the Types of Subqueries (5 main types)

VERY IMPORTANT: For all types of subqueries, if the subquery returns 0 rows, then the value returned by the subquery expression is NULL

Single row subquery

returns one row of data to the parent query, It uses operator s like =, <, >

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEES
WHERE SHIP_ID =  (SELECT SHIP_ID FROM EMPLOYEES WHERE LAST_NAME = 'Smith' AND
FIRST_NAME = 'Al')
```

IMPORTANT: single row subqueries must return at most ONE value, if they return more, then ERROR, for example, if the above subquery was `SELECT SHIP_ID FROM`, then chances are that the subquery will return more than one row and that would cause an error

Multiple row subquery

may return more than one row of data to the parent query, it uses operators like IN, and ANY, ALL (these with and operator before, for example ANY, >ANY, !=ALL…)

```
SELECT SHIP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEES
WHERE SHIP_ID IN  (SELECT SHIP_ID FROM EMPLOYEES WHERE LAST_NAME = 'Smith')
```

```
SELECT empno, sal FROM   emp WHERE  sal > ALL (2000, 3000, 4000);
```

IMPORTANT: The use of greater-than or lesser-than comparison conditions does not support multirow subqueries unless combined with ALL, ANY, or SOME. By themselves, they are used with single-row subqueries

Multiple-column (single row, multirow)

return two or more columns worth of data at once to the parent query, which must test for all of the columns at once, two types:

- Single row:

```
SELECT INVOICE_ID
FROM INVOICES
WHERE (INVOICE_DATE, TOTAL_PRICE) =
(SELECT START_DATE, SALARY
FROM PAY_HISTORY HERE PAY_HISTORY_ID = 4);
```

- Multirow:

```
SELECT EMPLOYEE_ID
FROM EMPLOYEES
WHERE (FIRST_NAME, LAST_NAME) IN
(SELECT FIRST_NAME, LAST_NAME FROM CRUISE_CUSTOMERS)
AND SHIP_ID = 1;
```

Scalar subqueries

always return one value, represented in one column of one row, every time

```
SELECT VENDOR_NAME,
(SELECT TERMS_OF_DISCOUNT FROM INVOICES WHERE
INVOICE_ID = 1) AS DISCOUNT
FROM VENDORS ORDER BY VENDOR_NAME;
```

IMPORTANT

Scalar subquery expressions cannot be used in the following locations:
n In CHECK constraints
n In GROUP BY clauses
n In HAVING clauses
n In a function-based index (which is coming up in Chapter 11)
n As a DEFAULT value for a column
n In the RETURNING clause of any DML statement
n In the WHEN conditions of CASE
n In the START WITH and CONNECT BY clauses, which we discuss in
Chapter 16.
Other than that, they can be used anywhere you would use an expression.

Correlates subqueries

Use data from a parent query to determine their own result. A correlated subquery might be a single-row, multiple-row, or multiple column subquery

```
SELECT A.SHIP_CABIN_ID, A.ROOM_STYLE, A.ROOM_NUMBER, A.SQ_FT
FROM SHIP_CABINS A
WHERE A.SQ_FT > (SELECT AVG(SQ_FT) FROM SHIP_CABINS
WHERE ROOM_STYLE = A.ROOM_STYLE)
ORDER BY A.ROOM_NUMBER;
```

Steps for correlated subqueries

1. The candidate row is fetched from the table specified in the outer query.
2. Rows are returned by the inner query, after being evaluated with the value from the candidate row in the outer query.
3. The WHERE clause of the outer query is evaluated.
4. The procedure is repeated for the subsequent rows of the table, till all the rows are processed.

Write Single-Row and Multiple-Row Subqueries

- **Single row:** The results of a single-row subquery can be compared from within the parent using a scalar comparison operator, such as the equal sign, or the greater-than or less-than sign. The column names are not required to match in such a comparison, but the datatypes must match, so that the parent query may compare columns of any name to subquery columns of any name, provided the datatypes match

- **Multiple-row**: They are compared differently to the parent query than single-row, using the multiple-row comparison conditions, such as IN, ANY ,or ALL, in combination with single-row comparison operators such as >, to avoid getting an execution error message

Write a Multiple-Column Subquery

- Multiple-column subqueries return several columns' worth of data to the parent query all at once

- The parent query must compare all of the columns together; the data types of each expression comparison much match between the parent and the subquery
- Multiple-column subqueries may return single-row or multiple-row answers

Use Scalar Subqueries in SQL

- Scalar subqueries return data in the form of one value, in one column’s worth of one row
- Scalar subqueries may be used almost anywhere that any expression could be used

Solve Problems with Correlated Subqueries

- Correlated subqueries use data from the parent in subquery predicates to determine what data to return to the parent query
- Correlated subqueries may present some performance degradation; however, they can perform tasks that could not otherwise be accomplished in a single query

Update and Delete Rows Using Correlated Subqueries

- The UPDATE and DELETE statements can use correlated subqueries
- The UPDATE can use correlated subqueries in the SET or the WHERE clause

```
-- Single column example

UPDATE MASTER_ORDERS X
   SET QTY=(SELECT COALESCE (Y.QTY, X.QTY)
              FROM ORDERS Y
              WHERE X.ORDER_NUM = Y.ORDER_NUM)
WHERE X.ORDER_NUM IN (SELECT ORDER_NUM  FROM ORDERS)


-- Multiple column example

UPDATE <table_name> <alias>
SET (<column_name_list>) = (
  SELECT <column_name_list>
  FROM <table_name> <alias> WHERE <alias.table_name> <condition> <alias.table_name>);


UPDATE INVOICES INV
SET TERMS_OF_DISCOUNT = '10 PCT'
WHERE TOTAL_PRICE = (SELECT MAX(TOTAL_PRICE)
FROM INVOICES
WHERE TO_CHAR(INVOICE_DATE, 'RRRR-Q') =
TO_CHAR(INV.INVOICE_DATE, 'RRRR-Q'));
```

- The DELETE statement can use correlated subqueries in the WHERE clause

Use the EXISTS and NOT EXISTS Operators

- The EXISTS operator can be used by a parent query to test a subquery and determine if it returns any rows at all, it is used to test whether the values retrieved by the inner query exist in the result of the outer query

```
SELECT PORT_ID, PORT_NAME
```

```
FROM PORTS P1
WHERE EXISTS (SELECT *FROM SHIPS S1 WHERE P1.PORT_ID = S1.HOME_PORT_ID);
```

IMPORTANT: EXISTS counts the number of rows, regardless its value, so if the subquery returns one row with value NULL, the EXISTS will consider it

- NOT EXISTS reverses the findings of EXISTS

Use the WITH Clause (subquery factoring clause)

- It enables users to reuse the same query block in a SELECT statement, if it occurs more than once in a complex query.
- It can improve the performance of a large query by storing the result of a query block having the WITH clause in the user's temporary tablespace.
- The query name in the WITH clause is visible to other query blocks in the WITH clause as well as to the main query block.

- The WITH clause can dynamically name a subquery so that the SELECT statement following the WITH clause can reference that subquery by name ,treating it as a dynamic table in real time
- Any subquery names assigned within the WITH clause are only good for that statement; they are not stored in the database

```
WITH
PORT_BOOKINGS AS (
      SELECT P.PORT_ID, P.PORT_NAME, COUNT(S.SHIP_ID) CT
      FROM PORTS P, SHIPS S
      WHERE P.PORT_ID = S.HOME_PORT_ID
      GROUP BY P.PORT_ID, P.PORT_NAME
),
DENSEST_PORT AS (
      SELECT MAX MAX_CT
      FROM PORT_BOOKINGS
)
SELECT PORT_NAME FROM PORT_BOOKINGS WHERE CT = (SELECT MAX_CT FROM DENSEST_PORT);
```

Creating Other Schema Objects – Views, Indexes, Synonyms- (C10)

Create and Use Simple and Complex Views

- A VIEW is a SELECT statement that is stored in the database and assigned a name

- The columns and expressions of the SELECT statement used to create a view become the columns of the VIEW

- You can use SELECT statements on views just as you would a table

- You can use INSERT, UPDATE, and/or DELETE statements on some views, depending on the constraints of the view's underlying table or tables, as well as other issues such as whether the view's SELECT statement includes aggregate functions or not, in other words, under certain conditions IT IS POSSIBLE TO INSERT, UPDATE and DELETE rows on a view, this will actually perform the action on the underlyings tables, conditions:

  - Each column in the view must map to a column of a single table. For example, if a view column maps to the output of a TABLE clause (an unnested collection), then the view is not inherently updatable.
  - The view must not contain any of the following constructs:
    - A set operator
    - a DISTINCT operator
    - An aggregate or analytic function
    - A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
    - A collection expression in a SELECT list
    - A subquery in a SELECT list
    - A subquery designated WITH READ ONLY
    - Joins, with some exceptions, as documented in Oracle Database Administrator's Guide

  - We can force a view to NOT TO BE updatable using WITH READ ONLY

```
create view vista_empleados
 as SELECT apellido, nombre, sexo, seccion from empleados with read only;
```

- An "inline view" is a subquery that replaces a table reference in a FROM clause of a DML statement

```
SELECT A.SHIP_ID, A.COUNT_CABINS, B.COUNT_CRUISES
FROM
      (SELECT SHIP_ID, COUNT(SHIP_CABIN_ID) COUNT_CABINS
      FROM SHIP_CABINS GROUP BY SHIP_ID) A
JOIN
      (SELECT SHIP_ID, COUNT(CRUISE_ORDER_ID) COUNT_CRUISES
      FROM CRUISE_ORDERS GROUP BY SHIP_ID) B
ON A.SHIP_ID = B.SHIP_ID;
```

- The VIEW can be treated like a table, with some limitations

- A VIEW based on a table that is subsequently altered may require recompilation with the ALTER VIEW . . . COMPILE statement: `ALTER VIEW VW_EMPLOYEES COMPILE;`

IMPORTANT: if you have a view on a table and the table is dropped, then the view IS NOT dropped automatically, it is marked as INVALID, in order to see what views are invalid, you can use the dictionary view USER_OBJECTS, checking the column STATUS

- WITH CHECK OPTION: it is an option that allows prevents you from updating a row to become a value that is not in view anymore, in other words: it makes sure you only create data that matches the definition of the view, its another type of integrity constraint.

```
create table t ( x int );
create or replace view v1  as SELECT * from t where x > 0
create or replace view v2 as SELECT * from t where x > 0 with check option
insert into v1 values ( -1 ); -- OK
insert into v2 values ( -1 ); -- ORA-01402: view WITH CHECK OPTION where-clause
violation
```

IMPORTANT TIP:

WITH CHECK OPTION can be used to define complex constraints, in the previous example, it is not allowed to insert a negative or zero number to table T, and this is really a constraint on t, but this constraint is "applied" only through the view v2. In some scenarios this can be really useful!!!

Another example:

```
INSERT INTO (SELECT order_id, order_date, customer_id FROM ORDERS WHERE
order_total=1000 WITCH CHECK OPTION) VALUES (13,SYSDATE,101)
```

This inline view will fail with this update, because it would insert a row having order_id=13, order_date=SYSDATE and customer_id=101 but the order_total would be null and this won't match the WHERE, and this is not allowed because the WITH CHECK OPTION

TYPICAL QUESTION: Imagine a table ORDER_ITEMS that among other columns has the following columns with constraint NOT NULL
ORDER_ID, LINE_ITEM_ID and PRODUCT_ID
Does the following view allow DML operations on it?

```
CREATE VIEW V1 AS SELECT order_id, product_id FROM order_items;
```

The answer is NO because when an insert statement were executed, then LINE_ITEM_ID would be NULL and that is not allowed, therefore ERROR

Create, Maintain, and Use Sequences

- A SEQUENCE is an object that dispenses numbers according to the rules established by the sequence

```
CREATE SEQUENCE sequence_name sequence_options;
```

- SEQUENCE OPTIONS:
    - **INCREMENT BY *integer*** Each new sequence number requested will increment by this number. A negative number indicates the sequence will descend. If omitted, the increment defaults to 1.

- **START WITH *integer*** Specifies the first number that will start the sequence. If omitted, START WITH defaults to MINVALUE (which we discuss in a bit) for ascending sequences, or MAXVALUE for descending sequences, unless NOMINVALUE or NOMAXVALUE are specified either explicitly or implicitly (by default), in which case START WITH defaults to 1.
- **MAXVALUE *integer*** Specifies the maximum number for the sequence. If omitted, then NOMAXVALUE is assumed.
- **NOMAXVALUE** Specifies that there is no MAXVALUE specified.
- **MINVALUE *integer*** Specifies the minimum number for the sequence. If omitted, NOMINVALUE is assumed, unless a MINVALUE is required by the presence of CYCLE, in which case the default is 1.
- **NOMINVALUE** Specifies that there is no MINVALUE specified.
- **CYCLE** When the sequence generator reaches one end of its range, restart at the other end. In other words, in an ascending sequence, once the generated value reaches the MAXVALUE, the next number generated will be the MINVALUE. In a descending sequence, once the generated value reaches the MINVALUE, the number generated will be the MAXVALUE.
- **NOCYCLE** When the sequence generator reaches the end of its range, stop generating numbers. NOCYCLE is the default. If no range is specified, NOCYCLE has no effect.

```
CREATE SEQUENCE SEQ_ORDER_ID START WITH 10 INCREMENT BY 5;
```

- SEQUENCES are ideal for populating primary key values
- The NEXTVAL pseudo column of a sequence returns the next available number in the sequence and must be the first reference to the sequence in any given login session
- The CURRVAL pseudo column can return the existing value as already defined by NEXTVAL;

SEQUENCES NEXTVAL and CURRVAL with oracle session problem

1) It can only be referenced in a session after the NEXTVAL reference has occurred, in other words,any attempt to retrieve the CURRVAL before NEXTVAL will lead to error, for example, suppose MERLIN_SEQ is a sequence

```
SELECT MERLIN_SEQ.CURRVAL FROM DUAL // WILL FAIL BECAUSE NEXTVAL MUST BE EXECUTED FIRST

SELECT MERLIN_SEQ.NEXTVAL FROM DUAL // OK

SELECT MERLIN_SEQ.CURRVAL FROM DUAL // OK
```

2) CURRVAL will keep the same value in the lifecycle of the session, regardless another session increases the sequence, in the previous example, if another session makes a NEXTVAL , then we'll see the value we had in our session, no the new in the other session

- A pseudo column reference to a sequence is a valid expression and can be referenced anywhere that a valid expression is allowed

`SELECT PROJECT_COST / (3 * SEQ_PROJ_COST.NEXTVAL) FROM PROJECTS; //`That is valid syntax. Whether it's useful or not is up to your business rules. But SQL recognizes this syntax and will execute it successfully

- If a valid sequence reference to NEXTVAL occurs within a DML statement that fails, the sequence still advances

Create and Maintain Indexes

- An INDEX object is based on one or more columns in a table

```
CREATE INDEX IX_INV_INVOICE_DATE ON INVOICES(INVOICE_DATE);
```

- The INDEX copies data from its table's columns on which it is built, and pre sorts that data in order to speed future queries

IMPLICIT INDEX (indexes created automatically by ORACLE)

If you create a constraint on a table that is of type PRIMARY KEY or UNIQUE, then as part of the creation of the constraint, SQL will automatically create an index to support that constraint on the column or columns, if such an index does not already exist.

```
CREATE TABLE SEMINARS(SEMINAR_ID NUMBER(11) PRIMARY KEY,SEMINAR_NAME VARCHAR2(30)
UNIQUE); -- will create two indexes, one for SEMINAR_ID and one for SEMINAR_NAME
```

NOTE: THESE ARE THE ONLY IMPLICIT INDEXES

- When the DML statements INSERT, UPDATE, or DELETE are executed on an indexed table so that the indexed data is changed, the index is automatically updated by SQL, thus adding to the workload of the DML statements
- INDEX objects can be built on one or more columns. Multiple-column indexes are "composite" indexes

```
CREATE INDEX IX_INV_INVOICE_VENDOR_ID ON INVOICES(VENDOR_ID, INVOICE_DATE);
```

Create Function-Based Indexes

- A function-based index can be based on an expression; it does not necessarily need to include a SQL function, and can simply be based on an expression of any kind

```
CREATE TABLE GAS_TANKS (GAS_TANK_ID NUMBER(7), TANK_GALLONS NUMBER(9), MILEAGE
NUMBER(9));

CREATE INDEX IX_GAS_TANKS_001 ON GAS_TANKS (TANK_GALLONS * MILEAGE);
```

- Queries that use the same function or expression may benefit from the index

Create Indexes Using the CREATE TABLE Statement (see implicit index above)

- When a PRIMARY KEY or UNIQUE constraint is created as part of a CREATE TABLE statement, and if no existing index supports the constraint, then an index is automatically created as part of the constraint

```
CREATE TABLE INVOICES (INVOICE_ID NUMBER(11) PRIMARY KEY, INVOICE_DATE DATE);
```

- **USING INDEX**: You can specify an index's creation as part of the CREATE TABLE statement. The USING INDEX clause only works for PRIMARY KEY and UNIQUE constraints.

```
ALTER TABLE emp ADD CONSTRAINT emp_id_pk PRIMARY KEY (emp_id) USING INDEX emp_id_idx;
```

- The USING INDEX clause can be invoked to explicitly define the index

```
CREATE TABLE INVOICES (INVOICE_ID NUMBER(11) PRIMARY KEY
USING INDEX (CREATE INDEX IX_INVOICES ON INVOICES(INVOICE_ID)),INVOICE_DATE DATE);
```

```
--NOTE THAT INDEX IS REFERRING TO TABLE INVOICES WHICH IS BEING CREATED AT THE VERY
MOMENT THE INDEX IS CREATED
```

IMPORTANT
If there is an index on a column or a set of columns, it is not possible to create another index on that column(s)

Create Private and Public Synonyms (CREATE OR REPLACE (PUBLIC) SYNONYM sss FOR zzz)

- SYNONYM objects are aliases for other objects in the database, it can be created for objects that do not yet exist in the database
- A private SYNONYM is created with the CREATE SYNONYM statement and is owned by a user account

```
CREATE OR REPLACE SYNONYM CO FOR CRUISE_ORDERS;
```

- A PUBLIC SYNONYM is created with the CREATE PUBLIC SYNONYM statement and is owned by the PUBLIC user account `CREATE PUBLIC SYNONYM WH FOR WORK_HISTORY;`
- PUBLIC SYNONYMs are automatically accessible to all users; however, before user accounts can access the aliased object, the user must have the appropriate privileges for the object for which the PUBLIC SYNONYM is an alias.
- If you have a synonym for a database object, and the database object is dropped, the synonym is not dropped—it exists independently of the object it renames.
- while you can create private SYNONYM for your objects, you need a system privilege CREATE PUBLIC SYNONYM to create a public synonym
- A public synonym and a private synonym can exist with the same name for the same table

Managing Schema Objects (C11)

IMPORTANT: Regarding to the ALTER TABLE statement: it is a DDL statement, and that when any DDL statement executes, it causes an implied commit event to occur.

ALTER SEQUENCE xxxx:

Use the ALTER SEQUENCE statement to change the increment, minimum and maximum values, cached numbers, and behavior of an existing sequence. This statement affects only future sequence numbers.

```
ALTER SEQUENCE customers_seq  CYCLE CACHE 5;

ALTER SEQUENCE customers_seq MAXVALUE 1500;
```

ALTER TABLE xxxx ADD

Can be used to add a column to a table: A column is added by specifying the column name and datatype; optionally, you can add a constraint and a default value

```
ALTER TABLE CRUISE_ORDERS ADD FIRST_TIME_CUSTOMER VARCHAR2(5) DEFAULT 'YES' NOT NULL;
```

ALTER TABLE xxxx MODIFY

Can be used to modify existing columns in a table

```
ALTER TABLE CRUISE_ORDERS MODIFY (ORDER_DATE DATE NULL);
```

- A column's datatype and other characteristics can be modified, but only insofar as the change does not conflict with any existing data in the table
- You cannot change a column's datatype if the column contains data already

```
ALTER TABLE CRUISE_ORDERS
MODIFY ORDER_DATE DATE
DEFAULT SYSDATE CONSTRAINT NN_ORDER_DATE NOT NULL;
```

ALTER TABLE xxxx DROP COLUMN

Used to remove a column from a table

```
ALTER TABLE ORDER_RETURNS DROP COLUMN CRUISE_ORDER_DATE;
ALTER TABLE ORDER_RETURNS DROP (CRUISE_ORDER_DATE); // BOTH WAYS ARE OK
```

- Once a column is removed with DROP, the data is lost
- A If you drop a column with a constraint, the constraint is also dropped. The same is true for any index objects on the column; they are also dropped

TYPICAL PROBLEM (similar to problem dropping a PK / Unique constraint)

- YOU CAN NOT DROP A PRIMARY KEY/UNIQUE COLUMN IF IT IS is referenced by some foreign keys, UNLESS IT IS USED THE "CASCADE CONSTRAINTS" OPTION

```
ALTER TABLE BASKET_FIXING_REPROCESS_TYPE DROP COLUMN ID CASCADE CONSTRAINTS;
```

ALTER TABLE xxxx SET UNUSED

Dropping a column can consume significant processing time if the table involved contains a lot of data, to mitigate this, you can use the SET UNUSED

```
ALTER TABLE ORDER_RETURNS SET UNUSED COLUMN CRUISE_ORDER_DATE;
ALTER TABLE ORDER_RETURNS SET UNUSED (CRUISE_ORDER_DATE, FORM_TYPE, NAME_SUFFIX);
```

- SET UNUSED renders a column permanently unavailable; it cannot be recovered !!. After a table has columns that are set to UNUSED, they can be dropped with the

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

- The SET UNUSED clause can benefit a large table in heavy production that cannot afford the overhead processing power of a DROP
- After executing the SET UNUSED on a column, you can add a new column called with the same name
- After executing the SET UNUSED on a column, all constraints are dropped

ALTER TABLE xxxx ADD CONSTRAINT

The following way works for constrainst UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK (not the NOT NULL constraint, however):

```
ALTER TABLE CRUISE_ORDERS PRIMARY KEY (CRUISE_ORDER_ID);
```

NOTES:

- CONSTRAINT: is optional, but if it is included, then it is compulsory the name of the constraint you make up, according to the rules for naming database objects

  ```
  ALTER TABLE CRUISE_ORDERS ADD CONSTRAINT PK_NEW_CONSTRAINT
  PRIMARY KEY (CRUISE_ORDER_ID);
  ```

- Sometimes it is useful to defer a constraint until next commit, so it can be violated meanwhile

ADDING A NOT NULL CONSTRAINT

As it is said, the ALTER TABLE xxx ADD CONSTRAINT does not work for NOT NULL constraints:
The syntax for working with the NOT NULL constraint has the same limitation with the ALTER TABLE statement that we already observed it has with the CREATE TABLE statement. That is to say: you cannot declare a NOT NULL constraint with the out-ofline syntax; it can only be created with the in-line syntax. In other words, this works:

```
ALTER TABLE CRUISE_ORDERS MODIFY CRUISE_ORDER_ID NOT NULL;
```

So does this:

```
ALTER TABLE CRUISE_ORDERS MODIFY CRUISE_ORDER_ID CONSTRAINT NN_CRUISE_ORDER_ID NOT
NULL;
```

But this does not:

```
ALTER TABLE CRUISE_ORDERS ADD CONSTRAINT NN_THIS_IS_WRONG NOT NULL (CRUISE_ORDER_ID);
```

This last sample code will fail. You cannot use the out-of-line syntax to create a NOT NULL constraint—you must use the in-line form.

SET CONSTRAINT xxx DEFERRED / INMEDIATE. INITIALLY DEFERRED DEFERRABLE

```
SET CONSTRAINT FK_SHIPS_PORTS DEFERRED;
```

You can set the normal behavior of the constraint at any time:

```
SET CONSTRAINT FK_SHIPS_PORTS IMMEDIATE;
```

VERY IMPORTANT: one constraint can be deferred it it was declared as DEFERRABLE in its definition:

ALTER TABLE SHIPS ADD CONSTRAINT FK_SHIPS_PORTS FOREIGN KEY (HOME_PORT_ID) REFERENCES PORTS (PORT_ID) DEFERRABLE;

Note: defining a DEFERRABLE constraint does not mean that the constraint is deferred at the time of creation, it means it *could be* deferred . Oracle allows to deferred a deferrable constraint at creation time by using INITIALLY DEFERRED DEFERRABLE

```
ALTER TABLE egg ADD CONSTRAINT eggREFchicken FOREIGN KEY (cID) REFERENCES chicken(cID)
    INITIALLY DEFERRED DEFERRABLE;
```

Note the following statement:

```
SET CONSTRAINT ALL DEFERRED;
```

ALTER TABLE xxxx DROP (CONSTRAINT / PRIMARY KEY / UNIQUE)

- <u>DROP PRIMARY KEY</u>

  ```
  ALTER TABLE table_name DROP PRIMARY KEY options;
  ```

  Options:

  - **CASCADE:** This drops any dependent constraints as well. In other words, a FOREIGN KEY may refer to this PRIMARY KEY constraint. Using CASCADE here will drop any and all FOREIGN KEY constraints that reference this PRIMARY KEY constraint. CASCADE is optional; the default is to not cascade.

  - **KEEP INDEX or DROP INDEX:** The default is DROP INDEX

TYPICAL PROBLEM

- YOU CAN NOT DROP A PRIMARY KEY / UNIQUE IF IT IS is referenced by some foreign keys, these Foreign Keys can be removed by using CASCADE:

```
ALTER TABLE  BASKET_FIXING_REPROCESS_TYPE DROP PRIMARY KEY CASCADE;
```

- In oracle 9, dropping a PRIMARY KEY removes the index created for the primary key. But In Oracle 10 this is not true, you need to specify DROP INDEX if you want to remove the index as well

```
ALTER TABLE  BASKET_FIXING_REPROCESS_TYPE DROP PRIMARY KEY CASCADE DROP INDEX;
```

- DROP UNIQUE: you don't need the name of the constraint, just the list of columns that are included in the constraint: `ALTER TABLE table_name DROP UNIQUE (column1, column2, . . . ) options;`
- DROP CONSTRAINT: to drop other constraints, you need the constraint name:

```
 ALTER TABLE table_name DROP CONSTRAINT constraint_name options;
```

ALTER TABLE xxxx (DISABLE / ENABLE VALIDATE / NO VALIDATE) CONSTRAINT

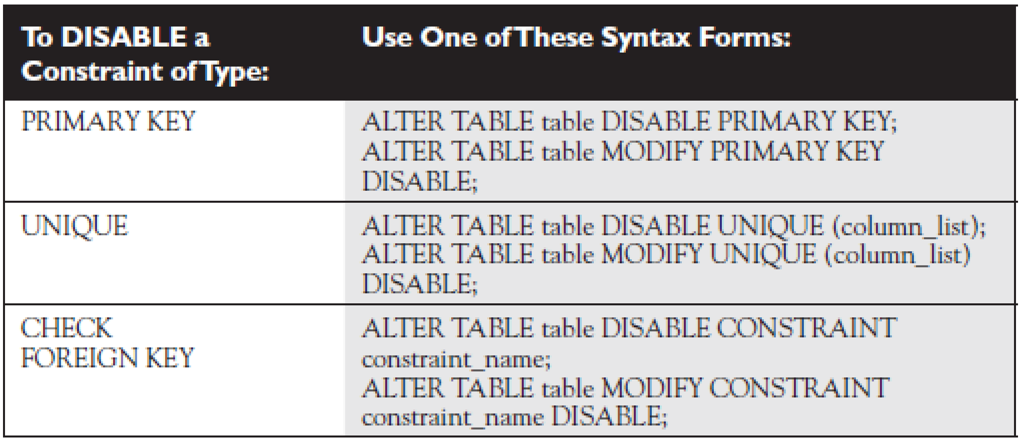
```
   ALTER TABLE table_name
   (DISABLE / ENABLE)
   (VALIDATE / NO VALIDATE)
   (PRIMARY KEY / UNIQUE (column1, column2, . . . ) / CONSTRAINT)
   CASCADE
```
(optionally for referential integrity constraints)

You can enable or disable integrity constraints at the table level using the CREATE TABLE or ALTER TABLE statement. You can also set constraints to VALIDATE or NOVALIDATE, in any combination with ENABLE or DISABLE, where:

ENABLE ensures that all incoming data conforms to the constraint
DISABLE allows incoming data, regardless of whether it conforms to the constraint
VALIDATE ensures that existing data conforms to the constraint
NOVALIDATE means that some existing data may not conform to the constraint

In addition:

- **ENABLE VALIDATE** is the same as ENABLE. The constraint is checked and is guaranteed to hold for all rows.
- **ENABLE NOVALIDATE** means that the constraint is checked, but it does not have to be true for all rows. This allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid.

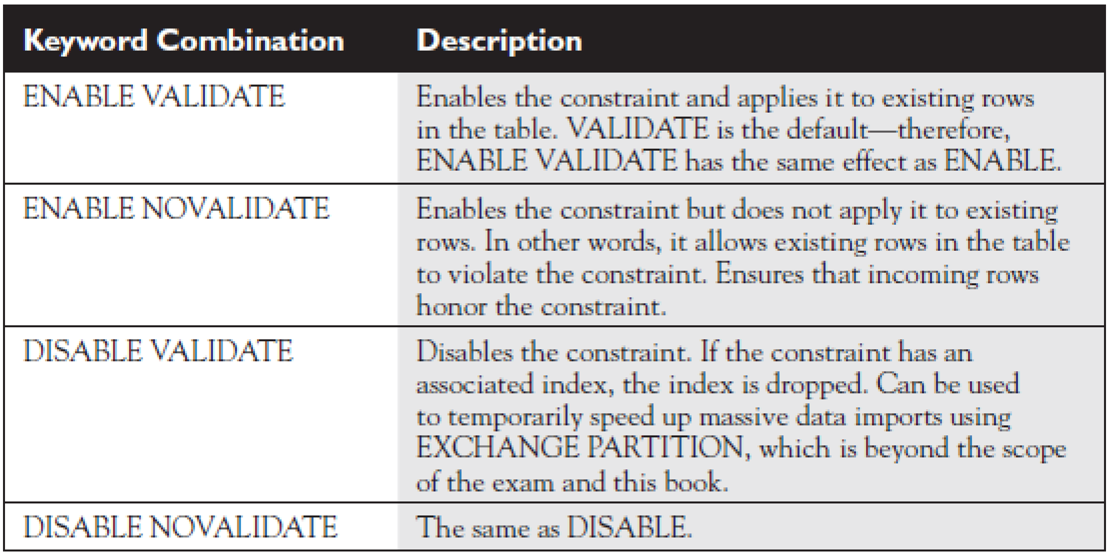


To DISABLE a Constraint of Type:	Use One of These Syntax Forms:

PRIMARY KEY	ALTER TABLE table DISABLE PRIMARY KEY;
	ALTER TABLE table MODIFY PRIMARY KEY DISABLE;

UNIQUE	ALTER TABLE table DISABLE UNIQUE (column_list);
	ALTER TABLE table MODIFY UNIQUE (column_list) DISABLE;

CHECK
FOREIGN KEY
	ALTER TABLE table DISABLE CONSTRAINT constraint_name;
	ALTER TABLE table MODIFY CONSTRAINT constraint_name DISABLE;



Keyword Combination	Description
ENABLE VALIDATE	Enables the constraint and applies it to existing rows in the table. VALIDATE is the default—therefore, ENABLE VALIDATE has the same effect as ENABLE.
ENABLE NOVALIDATE	Enables the constraint but does not apply it to existing rows. In other words, it allows existing rows in the table to violate the constraint. Ensures that incoming rows honor the constraint.
DISABLE VALIDATE	Disables the constraint. If the constraint has an associated index, the index is dropped. Can be used to temporarily speed up massive data imports using EXCHANGE PARTITION, which is beyond the scope of the exam and this book.
DISABLE NOVALIDATE	The same as DISABLE.

```
   Examples:

ALTER TABLE SHIPS DISABLE CONSTRAINT FK_SH_PO;
ALTER TABLE SHIPS ENABLE CONSTRAINT FK_SH_PO;
ALTER TABLE SHIPS ENABLE VALIDATE CONSTRAINT FK_SH_PO;
```

DROP TABLE xxxx CASCADE

```
DROP TABLE SHIP_HISTORY;
```

The previous statement drops the SHIP_HISTORY table,

TYPICAL PROBLEM (similar to problem dropping a PK / UNIQUE column)

- YOU CAN NOT DROP A PRIMARY KEY / UNIQUE COLUMN IF IT IS is referenced by some foreign keys, UNLESS IT IS USED THE "CASCADE CONSTRAINTS" OPTION

```
DROP TABLE SHIP_HISTORY CASCADE CONSTRAINTS;
```

DELETE and ON DELETE on CONSTRAINTS DEFINITIONS

Remember that you cannot DELETE a row in a table if dependent child rows exist. The reason is the FOREIGN KEY constraint on the child table, which is dependent on the PRIMARY KEY in the parent table. As we saw earlier, one alternative approach is to DISABLE the FOREIGN KEY constraint in SHIPS. Once disabled, you may delete the parent rows in PORTS. Another alternative is create the FOREIGN KEY with the ON DELETE CASCADE clause.

```
ALTER TABLE SHIPS DROP CONSTRAINT FK_SHIPS_PORTS;
ALTER TABLE SHIPS ADD CONSTRAINT FK_SHIPS_PORTS FOREIGN KEY (HOME_PORT_ID)
   REFERENCES PORTS (PORT_ID) ON DELETE CASCADE;
```

Perform FLASHBACK Operations

See chapter 6. =>Track the Changes to Data over a Period of Time:

Create and Use External Tables

- An external table is a read-only table that is defined within the database but exists outside of the database. In more technical terms, the external table's metadata is stored inside the database, and the data it contains is outside of the database.

- External tables have a number of restrictions on them. You can query them with the SELECT statement, but you cannot use any other DML statements on them.You can't create an INDEX on them, and they won't accept constraints.

- The communication between the external table's data storage file and database objects is based on the logic of SQL*Loader (ORACLE_LOADER) or Oracle Data Pump (ORACLE_DATAPUMP)

- You use a database object known as the **DIRECTORY** object as part of the definition of an external table

```
CREATE OR REPLACE DIRECTORY INVOICE_FILES AS '\LOAD_INVOICES';

-- object INVOICE_FILES is the directory

GRANT READ ON DIRECTORY INVOICE_FILES TO USER;

-- grant read to user 'USER'
```

Next, we execute a CREATE TABLE statement that references the directory, along with the necessary clauses to tell Oracle SQL to load the external file, and how to load it (line numbers added):

```
01 CREATE TABLE INVOICES_EXTERNAL
02 ( INVOICE_ID CHAR(3),
03 INVOICE_DATE CHAR(9),
04 ACCOUNT_NUMBER CHAR(13)
05 )
06 ORGANIZATION EXTERNAL
07 (TYPE ORACLE_LOADER – WE ARE USING THE ORACLE_LOADER to load data
08 DEFAULT DIRECTORY INVOICE_FILES -- MANDATORY CLAUSE !!!!
09 ACCESS PARAMETERS
10 (RECORDS DELIMITED BY NEWLINE
11 SKIP 2
12 FIELDS (INVOICE_ID CHAR(3),
13 INVOICE_DATE CHAR(9),
14 ACCOUNT_NUMBER CHAR(13))
15 )
```

Notes:

- lines 2 through 4 where we declared our table using the datatypes CHAR. You'll recall these are fixed-length datatypes. We did this to accommodate the transfer of rows in from the text file in lines 12 through 14. Each column's datatype is set to CHAR, the fixed-length alphanumeric datatype, and the counts for each datatype correspond to the counts of the columns in the text file 'INVOICE_DATA.TXT', which is identified in line 16 and is in the directory stored in the directory object INVOICE_FILES, named in line 8.

- Lines 1 through 5 form a complete CREATE TABLE statement by themselves, without the external table clause. But starting on line 6 are the keywords and clauses used to declare the external table, and together, lines 1 through 17 form the complete CREATE TABLE statement for our example.

- Line 6 includes the keywords ORGANIZATION EXTERNAL, which are required.

Oracle allows to transfer data from an ordinary oracle table to an external table, this process is called "unloading" and it is achived using the ORACLE_DATAPUMP

```
create table export_empl_info
 organization external
 ( type oracle_datapump
   default directory xtern_data_dir
   location ('empl_info_rpt.dmp')
 ) as SELECT * from empl_info
```

NOTE that a CREATE TABLE AS SELECT (CTAS) is used!!!

NOTE: A external table can be used to build a view

```
CREATE VIEW empvu AS SELECT * FROM empdept;  -- empdet is an external table
```

Using the Set Operators (C12)

Describe Set Operators : UNION, INTERSECT, MINUS, UNION ALL

- UNION combines the output of two SELECT statements, eliminating any duplicate rows that might exist

```
SELECT CONTACT_EMAIL_ID, EMAIL_ADDRESS FROM CONTACT_EMAILS WHERE STATUS = 'Valid'
UNION
SELECT ONLINE_SUBSCRIBER_ID, EMAIL FROM ONLINE_SUBSCRIBERS;
```

- INTERSECT combines the output of two SELECT statements, showing only the unique occurrences of data present in both rowsets, and ignoring anything that doesn’t appear in both sets

```
SELECT EMAIL_ADDRESS FROM CONTACT_EMAILS WHERE STATUS = 'Valid'
INTERSECT
SELECT EMAIL FROM ONLINE_SUBSCRIBERS;
```

- MINUS takes the first SELECT statement’s output and subtracts any occurrences of identical rows that might exist within the second SELECT statement’ soutput

```
SELECT EMAIL_ADDRESS FROM CONTACT_EMAILS WHERE STATUS = 'Valid'
MINUS
SELECT EMAIL FROM ONLINE_SUBSCRIBERS;
```

- UNION ALL does the same thing as UNION but does not eliminate duplicate rows

Use a Set Operator to Combine Multiple Queries into a Single Query

- The set operators are placed between two SELECT statements
- The two SELECT statements can be simple or complex and can include their own GROUP BY clauses, WHERE clauses, subqueries, and more
- The ORDER BY clause, if used, must be the final clause of the combined SELECT statements
- You can connect multiple SELECT statements with multiple set operators
- The set operators have equal precedence
- You can use parentheses to override set operator precedence

Control the Order of Rows Returned

- If an ORDER BY clause is used, it must be placed at the very end of the SQL statements, if it is in the middle then error, for example the following statemet is wrong:

```
SELECT employee_id, department_id FROM employees WHERE department_id= 50 ORDER BY
department_id
```

```
UNION
SELECT employee_id, department_id FROM employees WHERE department_id= 90
UNION
SELECT employee_id, department_id FROM employees WHERE department_id= 10;
```

- Multiple SELECTs that are connected with set operators may be sorted by position or reference

- When using ORDER BY reference, the column name in force is whatever column name exists in the first SELECT statement

```
SELECT 'Individual' CONTACT_CATEGORY, LAST_NAME || ', ' || FIRST_NAME
POINT_OF_CONTACT FROM CRUISE_CUSTOMERS
      UNION
SELECT CATEGORY,VENDOR_NAME FROM VENDORS ORDER BY POINT_OF_CONTACT;
```

Result:

```
CONTACT_CATEGORY          POINT_OF_CONTACT
----------------          -------------------------------------------
Supplier Acme Poker Chips
Partner Acme Steaks
Individual Bryant,        William
Individual Gilbert,       Nada
Individual MacCaulay,     Nora
```

IMPORTANT

- The name of the columns of the query is the names in the first SELECT, hence the ORDER BY is applied to first SELECT statement

```
SELECT 'a' col1, 'b' col2 from DUAL
UNION
SELECT 'c' col1, 'd' col4 from DUAL
--result have the columns "col1" and "col2"
```

- The number and the types of the columns must be consistent among the SELECTs statements:

```
SELECT 'a' col1, 'b' col2 from DUAL
UNION
SELECT 'c' col1 from DUAL -- ERROR: ORA-01789: query block has incorrect number of
result columns

SELECT 'a' col1, 'b' col2 from DUAL
UNION
SELECT 'c' col1, SYSDATE from DUAL -- ERROR: ORA-01790: expression must have same
datatype as corresponding expression
```

Generating Reports by Grouping Related Data (C13)

OLAP Reporting: GROUPING SETS, GROUP BY ROLLUP, GROUP BY CUBE

Imagine the scenario in wich you have to generate a report in which you need to show data of table grouped by two different columns, for example

```
SELECT manager_id, null hire_date, count(*) FROM Employees GROUP BY manager_id, 2
UNION ALL
SELECT null, hire_date, count(*) FROM  Employees GROUP BY 1, hire_date
```

The above rewritten as a Grouping Set…

```
SELECT manager_id, hire_date, count(*)
FROM employees
GROUP BY GROUPING SETS (manager_id, hire_date);
```

This is much more efficient than the version with UNION ALL

Where a large number of groupings are needed then the CUBE and ROLLUP statements extend this idea by calculating multiple groupings in a single statement.

- **GROUP BY CUBE** (hire_date, manager_id, product) will produce 2^3 =8 groupings
  1) hire_date, manager_id, product
  2) hire_date, manager_id
  3) hire_date, product
  4) manager_id, product
  5) hire_date
  6) manager_id
  7) product
  8) Grand Total

GROUP BY CUBE always calculates ALL the combinations - which may be far more than needed

- **GROUP BY ROLLUP** (hire_date, manager_id, product) will produce 4 groupings
  1) hire_date, manager_id, product
  2) hire_date, manager_id
  3) hire_date,
  4) Grand Total

GROUP BY ROLLUP calculates all combinations for the first column listed in the ROLLUP clause.

This can be further tuned by using parenthesis to remove some of the combinations
GROUP BY ROLLUP (hire_date, (manager_id, product)) will produce
1) hire_date, manager_id, product
2) hire_date
3) Grand Total

Grouping function

CUBE and ROLLUP will generate NULLs for each dimension at the subtotal levels.
The Grouping() function can be used to identify these rows, which can be very useful when performing additional calculations such as Ranking within a group.

The values returned by grouping() are:
0 for NULL data values
1 for NULL indicating a dimension subtotal

The results of Grouping() can be passed into a decode() e.g.
SELECT … PARTITION BY GROUPING(column1) ..
SELECT … PARTITION BY DECODE(GROUPING(column1), 1, 'My SubTotal', column1)) …

<u>As a summary, GROUPING is used to identify if the NULL value in an expression is a stored NULL value or created by ROLLUP or CUBE.</u>

<u>Combining (concatenating) Groupings:</u>

The CUBE and ROLLUP clauses can be combined as part of a standard GROUP BY clause

e.g. GROUP BY manager_id, ROLLUP (hire_date, product)

Notes
Grouping sets are typically 80 - 90% more efficient at producing sub-totals than equivalent SQL code.

ROLLUP/CUBE can be used with all aggregate functions (MAX, MIN, AVG, etc.)

A HAVING clause will apply to all the data returned.

Article: http://www.orafaq.com/node/56 (by Shouvik Basu)

Much of the OLAP reporting feature embedded in Oracle SQL is ignored. People turn to expensive OLAP reporting tools in the market - even for simple reporting needs. This article outlines some of the common OLAP reporting needs and shows how to meet them by using the enhanced aggregation features of Oracle SQL.

The article is divided in two sections. The first introduces the GROUP BY extensions of SQL, and the second uses them to generate some typical reports. A section at the end introduces the common OLAP terminologies.

The enhanced SQL aggregation features are available across all flavors of Oracle including Oracle Standard Edition One. It might be worth mentioning here, that Oracle OLAP, the special OLAP package Edition and Standard Edition One. Enhanced aggregation features discussed here have been tested on Oracle 9i and Oracle 10g.

Advanced Aggregation Extensions of GROUP BY

GROUPING SETS clause, GROUPING function and GROUPING_ID function

The fundamental concept of enhanced aggregation features of Oracle is that of GROUPING SETS. All other aggregation features can be expressed in terms of it. With GROUPING SETS clause comes the functions GROUPING, GROUPING_ID and GROUP_ID.

The GROUPING SETS clause in GROUP BY allows us to specify more than one GROUP BY options in the same record set. All GROUPING clause query can be logically expressed in terms of several GROUP BY queries connected by UNION. Table-1 shows several such equivalent statements. This is helpful in forming the idea of the GROUPING SETS clause. A blank set ( ) in the GROUPING SETS clause calculates the overall aggregate.

Table 1 - GROUPING SET queries and the equivalent GROUP BY queries

```
Set A - Aggregate Query with GROUPING SETS
Set B - Equivalent Aggregate Query with GROUP BY

A1. SELECT a, b, SUM(c) FROM tab1 GROUP BY GROUPING SETS ( (a,b) )

B1. SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b

A2. SELECT a, b, SUM( c ) FROM tab1 GROUP BY GROUPING SETS ( (a,b), a)

B2. SELECT a, b, SUM( c ) FROM tab1 GROUP BY a, b UNION
    SELECT a, null, SUM( c ) FROM tab1 GROUP BY a

A3. SELECT a,b, SUM( c ) FROM tab1 GROUP BY GROUPING SETS (a,b)

B3. SELECT a, null, SUM( c ) FROM tab1 GROUP BY a
    UNION
    SELECT null, b, SUM( c ) FROM tab1 GROUP BY b

A4. SELECT a, b, SUM( c ) FROM tab1 GROUP BY GROUPING SETS ( (a, b), a, b, ( ) )

B4. SELECT a, b, SUM( c ) FROM tab1 GROUP BY a, b UNION
    SELECT a, null, SUM( c ) FROM tab1 GROUP BY a, null UNION
    SELECT null, b, SUM( c ) FROM tab1 GROUP BY null, b UNION
    SELECT null, null, SUM( c ) FROM tab1
```

Example (**Table-1** Set 4) is like a superset of all the above cases and also includes an overall aggregate by the use of ( ). We will see latter that this result is similar to that of CUBE (a, b). The first 3 columns of **Table-2** show the result of a query of this type.

GROUPING clause uses a single scan to compute all the required aggregates. So the performance is better than its logical equivalent of *several GROUP BY and UNION*.

The general syntax of a SQL with GROUPING SETS is -

```
SELECT <grouping_columns>, <aggregate_functions>
FROM <table_list>
WHERE <where_condition>
GROUP BY GROUPING SETS (<column_set_1>, ... , <column_set_N>
```

The "column sets" can have none, one or more "grouping column" from SELECT. However, all columns from the SELECT should be present in at least one of the column sets. In mathematical terms -

UNION UNION should be
equal to

So the following two queries below will return error -

```
(1) SELECT a, b, c, SUM(d ) FROM tab1  GROUP BY GROUPING SETS ( (a,b), b)
```

```
--- Reason (a,b) U ( b ) is not equal to (a,b,c)

(2) SELECT a, b, SUM( c ) FROM tab1 GROUP BY GROUPING SETS (a, ( ) )
--- Reason (a) U ( ) is not equal to ( a, b )
```

Table 2 - A GROUPING SET query with GROUPING and GROUPING_ID Function on EMP

```
SELECT deptno, job, SUM(sal),
GROUPING(deptno) GDNO, GROUPING (job) GJNO,
GROUPING_ID(deptno, job) GID_DJ, GROUPING_ID(job, deptno) GID_JD
FROM EMP
GROUP BY GROUPING SETS ( (deptno, job), deptno, job, ( ))

    DEPTNO JOB         SUM(SAL)       GDNO       GJNO     GID_DJ     GID_JD
---------- --------- ---------- ---------- ---------- ---------- ----------

        10 CLERK           1300          0          0          0          0
        10 MANAGER         2450          0          0          0          0
        10 PRESIDENT       5000          0          0          0          0
        20 CLERK           1900          0          0          0          0
        20 ANALYST         6000          0          0          0          0
        20 MANAGER         2975          0          0          0          0
        30 CLERK            950          0          0          0          0
        30 MANAGER         2850          0          0          0          0
        30 SALESMAN        5600          0          0          0          0
        10                 8750          0          1          1          2
        20                10875          0          1          1          2
        30                 9400          0          1          1          2
           ANALYST         6000          1          0          2          1
           CLERK           4150          1          0          2          1
           MANAGER         8275          1          0          2          1
           PRESIDENT       5000          1          0          2          1
           SALESMAN        5600          1          0          2          1
                          29025          1          1          3          3

18 rows SELECTed.
```

<u>GROUPING Function and GROUPING_ID Function</u>

From **Table-2** we see that when aggregates are displayed for a column its value is null. This may conflict in case the column itself has some null values. There needs to be some way to identify NULL in column, which means aggregate and NULL in column, which means value. GROUPING function is the solution to that.

This function returns a flag "1" for a row in the result set if that column has been aggregated in that row. Otherwise the value is "0". There can be only one column expression as the argument of the GROUPING function and that column should also be in the SELECT. GROUPING function can be used to substitute the NULL value, which usually appears in columns at the aggregation level by something meaningful like Total.

GROUPING function has the general syntax of **GROUPING ( )**. It is used only in SELECT clause. It takes only a single column expression as argument.

GROUPING_ID takes a set of columns. It applies the GROUPING function on each column in its argument and composes a bit vector with the "0" and "1" values. It returns the decimal equivalent of the bit vector. The columns GID_DJ and GID_JD show the use of GROUPING_ID function and also show how interchanging the order of the columns inside the GROUPING_ID function might impact the result.

CUBE

This is the most generalized aggregation clause. The general syntax is **CUBE ( )**. It is used with the GROUP BY only. CUBE creates a subtotal of all possible combinations of the set of column in its argument. Once we compute a CUBE on a set of dimension, we can get answer to all possible aggregation questions on those dimensions. **Table-3** shows a cube building.

It might be also worth mentioning here that
GROUP BY CUBE(a, b, c) is equivalent to
GROUP BY GROUPING SETS ((a, b, c), (a, b), (b, c), (a, c), (a), (b), (c), ( )).

ROLLUP

ROLLUP clause is used with GROUP BY to compute the aggregate at the hierarchy levels of a dimension. ROLLUP(a, b, c) assumes that the hierarchy is "a" drilling down to "b" drilling down to "c".

ROLLUP (a, b, c) is equivalent to GROUPING SETS ((a, b, c), (a, b), (a), ( )).
The general syntax of ROLLUP is **ROLLUP( )**

Composite Columns

A composite column is a collection of columns that can be used in CUBE or ROLLUP. They are treated as unit before computing the aggregate. Composite columns usage in CUBE and ROLLUP and the equivalent GROUPING SETS -

```
. CUBE( (a, b), c) is equivalent to GROUPING SETS ( (a, b, c), (a, b) , c, ( ))

. ROLLUP ( a, (b, c) ) is equivalent to GROUPING SETS ( (a, b, c), ( a ), ( ) )
```

Partial GROUPING SETS, CUBE or ROLLUP

If any column appears in GROUP BY but outside the aggregation clauses discussed above. It can be thought of as being first column of the resulting GROUPING SET equivalent. The following examples make this clear.

```
 GROUP BY a, CUBE( b, c) is equivalent to
  GROUP BY GROUPING SETS ( (a, b, c), (a, b), (a, c), (a) )

GROUP BY a, ROLLUP( b, c) is equivalent to
  GROUP BY GROUPING SETS ( (a, b, c), (a, b), (a) )
```

From the book

Use the ROLLUP Operation to Produce Subtotal Values

- The ROLLUP operation is only allowed with the GROUP BY clause
- ROLLUP calculates subtotals and total values for the grouped sets of records
- The keyword ROLLUP follows GROUP BY
- Following the keyword ROLLUP is a set of parentheses that identifies the GROUP BY items that are to be aggregated with ROLLUP

- ROLLUP may be included with other GROUP BY expressions; each must be separated by commas

Use the CUBE Operation to Produce Cross tabulation Values

- The CUBE operation is only allowed with the GROUP BY clause
- CUBE tallies subtotals and totals for all combinations of the grouped expressions
- The keyword CUBE appears after GROUP BY and is followed by the CUBE list, enclosed in parentheses, citing the GROUP BY expressions to be CUBEd

Use the GROUPING Function to Identify the Row Values Created by ROLLUP or CUBE

- The GROUPING function identifies a grouped row set as either a regular row or a super aggregate row
- A regular row is a non-ROLLUP or non-CUBE row of typical GROUP B Youtput
- A super aggregate row is a GROUP BY row that represents a subtotal or total as directed by ROLLUP or CUBE
- The GROUPING function returns a value of 1 for super aggregate rows or 0for regular rows
- You can combine GROUPING with other functions to customize output format and behaviour for super aggregate rows versus regular rows

Use GROUPING SETS to Produce a Single Result Set

- The GROUPING SETS operator is ideal for GROUP BY queries that work with multiple groups and relatively large amounts of dat
- A The GROUPING SETS operator allows you to specify one or more GROUPBY combinations in a single query
- The use of GROUPING SETS offers advantages over ROLLUP or CUBE when only some of the subtotaled rows are desired

Managing Objects with Data Dictionary Views (C14)

Use the Data Dictionary Views to Research Data on Your Objects

- The data dictionary is made of tables that store data about the database
- The data dictionary contains the metadata (data of data) for your database
- It contains information about tables, views, constraints, indexes, sequences, synonyms, roles, privileges, and any and all other objects you might create in the database
- It keeps track of all the users in the database, and which user account owns which objects, who has privileges on which object, the status of each object, and more
- Oracle automatically updates and maintains the data dictionary views with each DDL statement executed throughout the database

<table>
<tr><th colspan="3">Prefixes of some of the Data Dictionary Views</th></tr>
<tr><th>Prefix</th><th>views</th><th>Description</th></tr>
<tr><td colspan="3">NOTE: Views with the same name but different prefixes, such as DBA, ALL and USER, use the same base tables from the data dictionary</td></tr>
<tr><td>USER_</td><td>359</td><td>Objects owned by the current user accessing the view:
<table>
<tr><td>USER_CATALOG</td><td>All tables, views, synonyms, and sequences owned by USER</td></tr>
<tr><td>USER _OBJECTS</td><td>Objects owned by USER</td></tr>
<tr><td>USER _COL_PRIVS</td><td>Grants on columns of tables owned by USER</td></tr>
<tr><td><b>USER _CONSTRAINTS</b></td><td>Constraints on tables owned by USER</td></tr>
<tr><td>USER _DEPENDENCIES</td><td>Dependencies to and from a user’s objects</td></tr>
<tr><td>USER _CONS_COLUMNS</td><td>Accessible columns in constraint definitions for tables owned by USER</td></tr>
<tr><td>USER _ERRORS</td><td>Current errors on stored objects owned by USER</td></tr>
<tr><td>USER _INDEXES</td><td>Indexes owned by USER</td></tr>
<tr><td>USER _IND_COLUMNS</td><td>Columns in user tables used in indexes owned by USER</td></tr>
<tr><td>USER _SEQUENCES</td><td>Sequences owned by USER</td></tr>
<tr><td><b>USER _SYNONYMS</b></td><td>Private synonyms owned by USER (Public synonyms are displayed in ALL_SYNONYMS and DBA_SYNONYMS.)</td></tr>
<tr><td>USER _TABLES (synonym TABS)</td><td>Tables owned by USER</td></tr>
<tr><td>USER _TAB_COLUMNS (synonym COLS)</td><td>Columns in USER’s own tables and views</td></tr>
<tr><td>USER_TAB_COMMENTS</td><td>displays comments on the tables and views owned by the current user. See <b>COMMENT</b> statement BELOW</td></tr>
<tr><td>USER _TAB_PRIVS</td><td>Grants on objects owned by USER</td></tr>
<tr><td>USER _VIEWS</td><td>Views owned by USER</td></tr>
<tr><td>USER_CONS_COLUMNS</td><td>describes columns that are owned by the current user and that are specified in constraint definitions.</td></tr>
</table></td></tr>
<tr><td>ALL_</td><td>334</td><td>Objects owned by any user in the database to which the current user has privileges. <b>ONLY ACCESIBLE TO ADMINISTRATORS</b></td></tr>
<tr><td>DBA_</td><td>670</td><td>All objects in the database.</td></tr>
</table>

V_$ (for views) V$ (for public synonyms)	488	Dynamic performance views, each of which has a public synonym counterpart. Stores information about the local database instance. NOTE: Oracle formally recommends that the dynamic nature of these views (V$) does not guarantee read consistency for anything other than the simplest of single-view queries
GV_$ (for views) GV$ (for public synonyms)	450	Global dynamic performance views.
Other SM$, AUDIT_, CHANGE_, TABLE_ CLIENT_, COLUMN_, DICT_, DATABASE_, DBMS_, GLOBAL_, INDEX_, LOGSTDBY_, NLS_, RESOURCE_, ROLE_, SESSION_, CLIENT_RESULT_CACHE_STATS$, or no prefix, etc.	40	The remaining views of the data dictionary have a variety of prefixes and unique individual names.

- Most (but not all) of the data dictionary views are stored with comments that provide brief descriptions about each view and what it contains; many of the columns of the views also have comments

```
SELECT '*TABLE: ' || TABLE_NAME, COMMENTS
FROM ALL_TAB_COMMENTS
WHERE OWNER = 'SYS'
AND TABLE_NAME = 'USER_SYNONYMS'
UNION
SELECT 'COL: ' || COLUMN_NAME, COMMENTS
FROM ALL_COL_COMMENTS
WHERE OWNER = 'SYS'
AND TABLE_NAME = 'USER_SYNONYMS' ;
```

That's the query; here are the results:

```
'*TABLE:'||TABLE_NAME COMMENTS
----------------------- ----------------------------------------------
*TABLE: USER_SYNONYMS The user's private synonyms
COL: DB_LINK Database link referenced in a remote synonym
COL: SYNONYM_NAME Name of the synonym
COL: TABLE_NAME Name of the object referenced by the synonym
COL: TABLE_OWNER Owner of the object referenced by the synonym
```

- You can add comments of your own alongside the data dictionary record for your own objects that you've created

COMMENT STATEMENT

- The COMMENT statement is how you store a comment in the data dictionary for any table you own, and also for its associated columns

```
COMMENT ON TABLE PORTS
IS 'Listing of all ports of departure and arrival.';
```

To see the results, you could use this query:

```
SELECT COMMENTS
```

```
FROM USER_TAB_COMMENTS
WHERE TABLE_NAME = 'PORTS';
COMMENTS
---------------------------
Listing of all ports of departure and arrival.
```

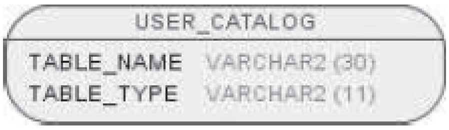
Query Various Data Dictionary Views : DICTIONARY, USER_CATALOG, USER_OBJECTS, USER_TABLES (synonym TABS), USER_TAB_COLUMNS (synonym COLS), USER_CONSTRAINTS

- The **DICTIONARY** view is a great starting point for finding what you might be looking for in the data dictionary

  ```
  DESC DICTIONARY;
  ```

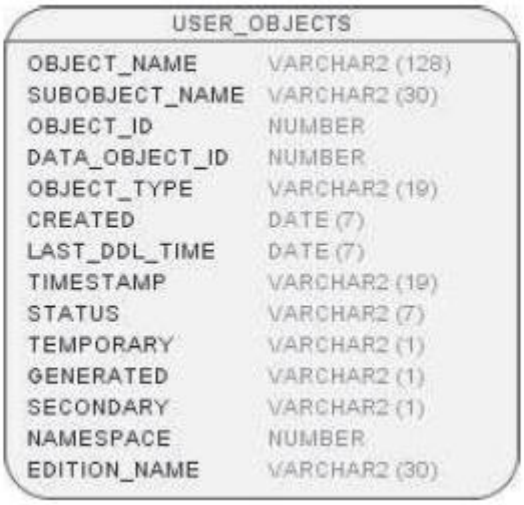
- The **USER_CATALOG** view contains a summary of information about some of the major objects owned by your user account

```
SELECT TABLE_TYPE, COUNT(*)
FROM USER_CATALOG
GROUP BY TABLE_TYPE;
TABLE_TYPE COUNT(*)
----------- ----------------------
SEQUENCE 21
TABLE 35
VIEW 2
SYNONYM 1
```



USER_CATALOG	
TABLE_NAME	VARCHAR2 (30)
TABLE_TYPE	VARCHAR2 (11)

- The **USER_OBJECTS** view is similar but with much more information



USER_OBJECTS	
OBJECT_NAME	VARCHAR2 (128)
SUBOBJECT_NAME	VARCHAR2 (30)
OBJECT_ID	NUMBER
DATA_OBJECT_ID	NUMBER
OBJECT_TYPE	VARCHAR2 (19)
CREATED	DATE (7)
LAST_DDL_TIME	DATE (7)
TIMESTAMP	VARCHAR2 (19)
STATUS	VARCHAR2 (7)
TEMPORARY	VARCHAR2 (1)
GENERATED	VARCHAR2 (1)
SECONDARY	VARCHAR2 (1)
NAMESPACE	NUMBER
EDITION_NAME	VARCHAR2 (30)

- **USER_TABLES** table (synonym **TABS**) is helpful for inspecting table metadata, as is its companion **USER_TAB_COLUMNS** (synonym **COLS**). This section will look at USER_TAB_COLUMNS in particular.

// will display some of the most basic information about a table (*TABLE_NAME*) and its columns.	// If you have the name of a column and aren't sure which table it might be part of, the data dictionary can assist

```
SELECT COLUMN_NAME, DECODE(
 DATA_TYPE,
'DATE' , DATA_TYPE ,
'NUMBER' , DATA_TYPE || DECODE(DATA_SCALE,
NULL,
NULL,
'(' || DATA_PRECISION || ',' || DATA_SCALE ||
')'),
'VARCHAR2', DATA_TYPE || '(' || DATA_LENGTH ||
')', NULL)
DATA_TYPE
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'TABLE_NAME';

Here's the output:

COLUMN_NAME                     DATA_TYPE
------------------------------  ---------------
INVOICE_ID                      NUMBER
INVOICE_DATE                    DATE
ACCOUNT_NUMBER                  VARCHAR2(80)
TERMS_OF_DISCOUNT               VARCHAR2(20)
VENDOR_ID                       NUMBER
TOTAL_PRICE                     NUMBER(8,2)
SHIPPING_DATE                   DATE
```

```
SELECT TABLE_NAME
FROM USER_TAB_COLUMNS
WHERE COLUMN_NAME = 'EMPLOYEE_ID';
```

- The status of objects is also stored—for example, the data dictionary flags views that are invalid and might need recompilation

One of the many useful tasks you can accomplish with the data dictionary is to check for the status of a view that you've created. Remember from Chapter 10 that a view is a named query based on a table, and that after the view has been created, if the table is altered for any reason, you may have to recompile the view.

```
SELECT STATUS, OBJECT_TYPE, OBJECT_NAME
FROM USER_OBJECTS
WHERE STATUS = 'INVALID'
ORDER BY OBJECT_NAME;
```

- All roles and privileges of all users on all objects are stored somewhere in the data dictionary

Privileges are discussed at length in Chapter 18, when we discuss user access. For now, note that privileges can be inspected using the following views:

  - **USER_SYS_PRIVS** System privileges granted to the current user
  - **USER_TAB_PRIVS** Granted privileges on objects for which the user is the owner, grantor, or grantee
  - **USER_ROLE_PRIVS** Roles granted to the current user
  - **DBA_SYS_PRIVS** System privileges granted to users and roles
  - **DBA_TAB_PRIVS** All grants on objects in the database
  - **DBA_ROLE_PRIVS** Roles granted to users and roles
  - **ROLE_SYS_PRIVS** System privileges granted to roles
  - **ROLE_TAB_PRIVS** Table privileges granted to roles
  - **SESSION_PRIVS** Session privileges which the user currently has set

- Inspecting CONSTRAINTS: The USER_CONSTRAINTS view is one of the more useful views.

The USER_CONSTRAINTS view is one of the more useful views. Here's a query you might run to check the current state of constraints on a table CRUISES:

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, R_CONSTRAINT_NAME, STATUS
FROM USER_CONSTRAINTS
WHERE TABLE_NAME = 'CRUISES';
```

Here is an example of the output:

```
CONSTRAINT_NAME      CONSTRAINT_TYPE R_CONSTRAINT_NAME STATUS
------------------   -----------------  ---------------- --------
PK_CRUISE                P                             ENABLED
FK_CRUISES_CRUISE_TYPES  R           PK_CRUISE_TYPE_ID ENABLED
FK_CRUISES_SHIPS         R           PK_SHIP           ENABLED
FK_CRUISES_EMPLOYEES     R           PK_EMPLOYEES      ENABLED
```

The output lists all of the constraints on the CRUISES table. We're seeing four: one primary key and three foreign keys. How do you know which constraint is a PRIMARY KEY, and which is a FOREIGN KEY? The answer is by the CONSTRAINT_TYPE column:

- P = PRIMARY KEY
- R = FOREIGN KEY. The R is for “referential integrity”.
- U = UNIQUE
- C = CHECK or NOT NULL constraint
-

Other columns of USER_CONSTRAINTS:

- **R_CONSTRAINT_NAME** Name of the unique constraint definition for referenced table

- **DELETE_RULE** column shows if a foreign key constraint was created with ON DELETE CASCADE or ON DELETE SET NULL.

- **SEARCH_CONDITION** column is particularly useful for inspecting CHECK constraint criteria. For example:

```
SELECT SEARCH_CONDITION
FROM USER_CONSTRAINTS
WHERE CONSTRAINT_NAME = 'CK_PROJECT_COST'
AND CONSTRAINT_TYPE = 'C';
SEARCH_CONDITION
------------------------
PROJECT_COST < 1000000
```

Manipulating Large Data Sets (C15)

Manipulate Data Using Subqueries: CREATE TABLE AS SELECT (CTAS)

The CREATE TABLE AS SELECT statement, also known as CTAS, uses a subquery to populate the new table's rows

```
CREATE TABLE INVOICES_ARCHIVED AS
SELECT *
FROM INVOICES
WHERE SHIPPING_DATE < (ADD_MONTHS(SYSDATE,-12));
```

- CTAS can also be used to name each column in the new table

```
CREATE TABLE ROOM_SUMMARY (SHIP_ID, SHIP_NAME, ROOM_NUMBER, TOT_SQ_FT)
AS
SELECT A.SHIP_ID,
A.SHIP_NAME,
B.ROOM_NUMBER,
B.SQ_FT + NVL(B.BALCONY_SQ_FT,0)
FROM SHIPS A JOIN SHIP_CABINS B
ON A.SHIP_ID = B.SHIP_ID;
```

- CTAS can also define the data type of each new column

- Subqueries in CTAS must provide a name for each column; complex expressions should be named with a column alias

VERY IMPORTANT: the created table will inherit the column data type definitions and the NOT NULL constraint that were created explicitly created (as part of PK are not included because it is implicitly created), this means that other constrainsts are not passed, even the PRIMARY KEY

However, these constraints can be defined as follows:

```
CREATE TABLE emp5 (empid PRIMARY KEY) AS SELECT empid FROM emp;
```

- The UPDATE statement can use a correlated subquery to set values to one or more columns from one or more rows within a data source at one time

```
UPDATE PORTS PT
SET (TOT_SHIPS_ASSIGNED, TOT_SHIPS_ASGN_CAP) =
(SELECT COUNT(S.SHIP_ID) TOTAL_SHIPS,
SUM(S.CAPACITY) TOTAL_SHIP_CAPACITY
FROM SHIPS S
WHERE S.HOME_PORT_ID = PT.PORT_ID
GROUP BY S.HOME_PORT_ID);
```

- In the UPDATE statement with correlated subquery, the table alias for the UPDATE table can be referenced within the subquery

- The INSERT statement can be used with a subquery to insert more than one row at a time

Describe the Features of Multitable INSERTs

- Multitable inserts are useful for applying conditional logic to the data being considered for insertion
- Conditional logic can evaluate incoming rows of data in a series of steps, using several evaluation conditions, and offer alternative strategies for adding data to the database, all in a single SQL statement
- Multitable INSERT statements offer flexibility and performance efficiency over the alternative approaches of using multiple SQL statements
- EXAM WATCH: Multitable INSERT statements require a subquery.
- EXAM WATCH: Remember: a table alias defined in a subquery of a multitable INSERT is not recognized throughout the rest of the INSERT statement. Also, if a multitable INSERT statement fails for any reason, the entire statement is rolled back and no rows are inserted in any of the tables.

Use the Following Types of Multitable INSERTs: Unconditional, Conditional, and Pivot

Multitable Insert Unconditional: INSERT ALL INTO XXX (…) values (…) INTO XXX (…) VALUES (…) … SELECT statement

```
INSERT ALL
INTO CO_2008 (CRUISE_ORDER_ID, ORDER_DATE, CRUISE_CUSTOMER_ID, SHIP_ID)
VALUES (CRUISE_ORDER_ID, ORDER_DATE,CRUISE_CUSTOMER_ID, SHIP_ID)

INTO CO_ELCARO (CRUISE_ORDER_ID, ORDER_DATE, CRUISE_CUSTOMER_ID, SHIP_ID)
VALUES  (CRUISE_ORDER_ID, ORDER_DATE,CRUISE_CUSTOMER_ID, SHIP_ID)

INTO CO_ARCHIVED (CRUISE_ORDER_ID, ORDER_DATE,CRUISE_CUSTOMER_ID, SHIP_ID)
VALUES  (CRUISE_ORDER_ID, ORDER_DATE,CRUISE_CUSTOMER_ID, SHIP_ID)

SELECT CRUISE_ORDER_ID, ORDER_DATE, CRUISE_CUSTOMER_ID, SHIP_ID FROM CRUISE_ORDERS;
```

Note the following is OK:

```
INSERT ALL
  INTO t1
  INTO t2
  INTO t3
  INTO t4
SELECT owner,object_type,object_name,object_id,created FROM all_objects;
```

Multitable Insert Conditional: INSERT WHEN …THEN INTO XXX (…) VALUES (…) WHEN …THEN INTO XXX (…) VALUES (…) … ELSE… SELECT statement

```
INSERT [FIRST/ALL]
- IF first, then only insert of the first matched condition is execute
- IF ALL, then all insert of the all matched condition is execute, by default is ALL
WHEN (TO_CHAR(DATE_SHIPPED,'RRRR') <= '2009') THEN
INTO INVOICES_THRU_2009 (INVOICE_ID, INVOICE_DATE,SHIPPING_DATE, ACCOUNT_NUMBER)
VALUES  (INV_NO, DATE_ENTERED, DATE_SHIPPED, CUST_ACCT)
```

```
WHEN (TO_CHAR(DATE_SHIPPED,'RRRR') <= '2008') THEN
INTO INVOICES_THRU_2008 (INVOICE_ID, INVOICE_DATE, SHIPPING_DATE, ACCOUNT_NUMBER)
VALUES (INV_NO, DATE_ENTERED, DATE_SHIPPED, CUST_ACCT)

WHEN  (TO_CHAR(DATE_SHIPPED,'RRRR') <= '2007') THEN
INTO  INVOICES_THRU_2007 (INVOICE_ID, INVOICE_DATE,SHIPPING_DATE, ACCOUNT_NUMBER)
VALUES (INV_NO, DATE_ENTERED, DATE_SHIPPED, CUST_ACCT)

--OPTIONALLY, ELSE
SELECT INV_NO, DATE_ENTERED, DATE_SHIPPED, CUST_ACCT FROM WO_INV;
```

- Multitable INSERT statements may use conditional operations such as the WHEN condition and the ELSE clause
- A WHEN condition can be used to evaluate incoming data and determine if it should be inserted into the database, and if yes, which table and which columns are to be inserted
- The ELSE clause is a last alternative choice that will execute if no WHEN condition evaluated to true
- Both WHEN and ELSE are associated with their own unique INSERT statement directives; depending on which conditions apply, the appropriate INSERT statement directives will execute
- Each condition can INSERT data in different ways into different tables
- The INSERT FIRST statement tests each WHEN condition and executes the associated INSERT statement directives with the first WHEN condition that evaluates to true.The INSERT ALL statement executes all of the WHEN conditions that evaluate to true
- The ELSE clause executes for either the INSERT FIRST or INSERT ALL statement when none of the WHEN conditions have executed
- The subquery of a multitable INSERT determines the data that will be considered in the insert logic; it can be a complex query, and can include joins, GROUP BY clauses, set operators, and other complex logic

PIVOT

You can use a conditional multitable INSERT statement to transform data from a spreadsheet structure to a rows-and-columns structure. This section describes the technique. NOTE:PIVOT AND UNPIVOT operators are not included in the exam!!!!, so here is the PIVOT technique included in the exam:

Suposse the following is the structure of a spreadsheet, this information is in table SHIP_CABIN_GRID

ROOM_TYPE	OCEAN	BALCONY	NO_WINDOW	The idea is to move the information of some columns to rows, in this case, the columns “OCEAN”, “BALCONY” and “NO_WINDOW” are reduced to a single column
ROYAL	1745	1635		
SKYLOFT	722	722		

PRESIDENTIAL	1142	1142	1142	“WINDOW_TYPE”
LARGE	225		211	
STANDARD	217	554	586	

NOTE: the number are the prices for one type of room (OCEAN, BALCONY and NO_WINDOW) for a particular window_type (royal, skyloft, presidential, large and standard)

Our goal now is to to build a table with the following structure from_SHIP_CABIN_GRID

(ROOM_TYPE, WINDOW_TYPE, PRICE)

This can be achieved by the following pivot multitable insert

```
INSERT ALL
WHEN OCEAN IS NOT NULL THEN
INTO SHIP_CABIN_STATISTICS (ROOM_TYPE, WINDOW_TYPE, SQ_FT)
VALUES (ROOM_TYPE, 'OCEAN', OCEAN)
WHEN BALCONY IS NOT NULL THEN
INTO SHIP_CABIN_STATISTICS (ROOM_TYPE, WINDOW_TYPE, SQ_FT)
VALUES (ROOM_TYPE, 'BALCONY', BALCONY)
WHEN NO_WINDOW IS NOT NULL THEN
INTO SHIP_CABIN_STATISTICS (ROOM_TYPE, WINDOW_TYPE, SQ_FT)
VALUES (ROOM_TYPE, 'NO WINDOW', NO_WINDOW)
SELECT ROWNUM RN, ROOM_TYPE, OCEAN, BALCONY, NO_WINDOW
FROM SHIP_CABIN_GRID;
```

Merge Rows in a Table: MERGE combines INSERT, UPDATE, and DELETE into a single statement

- The MERGE statement is one of the SQL DML statements, alongside SELECT, INSERT, UPDATE, and DELETE

- MERGE replicates some of the functionality found in INSERT, UPDATE, and DELETE and combines it all into a single statement that executes with a single pass through the database

```
MERGE INTO table USING table | subquery
  ON condition
  WHEN MATCHED THEN UPDATE SET col = expression | DEFAULT where_clause
                    DELETE where_clause
  WHEN NOT MATCHED THEN INSERT (col, col2) VALUES (expr1, expr2 | DEFAULT)
    where_clause
WHERE condition;
```

- MERGE doesn’t do anything new that you cannot already do with existing DML statements, but it does them more efficiently in combination

```
MERGE INTO WWA_INVOICES WWA
    USING ONTARIO_ORDERS ONT
 ON (WWA.CUST_PO = ONT.PO_NUM)
   WHEN MATCHED THEN UPDATE SET  WWA.NOTES = ONT.SALES_REP
   WHEN NOT MATCHED THEN INSERT
     (WWA.INV_ID, WWA.CUST_PO, WWA.INV_DATE, WWA.NOTES)
       VALUES (SEQ_INV_ID.NEXTVAL, ONT.PO_NUM, SYSDATE, ONT.SALES_REP)
```

```
WHERE SUBSTR(ONT.PO_NUM,1,3) <> 'NBC';
```

- The MERGE statement includes an “update clause” and an “insert clause”
  - The WHEN MATCHED THEN UPDATE keywords form the “update clause”
  - The WHEN NOT MATCHED THEN INSERT keywords form the “insert clause”
- The DELETE clause of the MERGE statement only deletes rows that were first updated with the “update clause” and remain after a successful update; they must also meet the WHERE condition of the “delete clause”
- The DELETE WHERE condition evaluates the updated value, not the original value that was evaluated by the UPDATE SET ... WHERE condition.

```
 MERGE INTO WWA_INVOICES WWA
  USING ONTARIO_ORDERS ONT
  ON (WWA.CUST_PO = ONT.PO_NUM)
  WHEN MATCHED THEN UPDATE SET WWA.NOTES = ONT.SALES_REP
  DELETE WHERE WWA.INV_DATE < TO_DATE('01-SEP-09')
  WHEN NOT MATCHED THEN INSERT
  (WWA.INV_ID, WWA.CUST_PO, WWA.INV_DATE, WWA.NOTES)
 VALUES (SEQ_INV_ID.NEXTVAL, ONT.PO_NUM, SYSDATE, ONT.SALES_REP)
WHERE SUBSTR(ONT.PO_NUM,1,3) <> 'NBC';
```

IMPORTANT

- You cannot update a column that is referenced in the ON condition clause.
- You cannot specify DEFAULT when updating a view.

For example, the following won’t work because the column on the ON is being updated, that is an error

```
MERGE INTO customers c
USING customer_vu cv ON (c.customer_id = cv.customer_id)
WHEN MATCHED THEN
UPDATE SET
c.customer_id = cv.customer_id,
c.cust_name = cv.cust_name,
c.cust_email = cv.cust_email,
c.income_level = cv.income_level
WHEN NOT MATCHED THEN
INSERT VALUES(cv.customer_id,cv.cust_name,cv.cust_email,cv,income_level)
WHERE cv.income_level >100000;
```

Track the Changes to Data : FLASHBACK TABLE xxx TO (BEBORE DROP / SCN / RESTORE POINT / TIMESTAMP)

It is possible to track changes of a “table” in a point in the past, there are 4 ways, the syntax is

FLASHBACK TABLE xxx TO <any of the four ways>

BEFORE DROP: to restore a table that has been dropped

```
CREATE TABLE HOUDINI (VOILA VARCHAR2(30));
INSERT INTO HOUDINI (VOILA) VALUES ('Now you see it.');
COMMIT;
DROP TABLE HOUDINI;
FLASHBACK TABLE HOUDINI TO BEFORE DROP;
SELECT * FROM HOUDINI;
```

RECYCLE BIN (USER_RECYCLEBIN), PURGE TABLE zzzz

- If a table has been dropped, it goes into the recycle bin
- You can investigate the contents of the recycle bin to determine what objects are available for use by FLASHBACK operations

  ```
  SELECT * FROM USER_RECYCLEBIN;
  ```

- Once an object has been dropped, if it is also purged with the PURGE statement (permanently removes a given item from the recycle bin) , it is no longer able to be recovered with FLASHBACK TABLE

  ```
  PURGE TABLE HOUDINI;
  ```

- In addition to restoring a dropped table, the FLASHBACK TABLE statement can also be used to restore data within the table as of a particular time in the database
  - IMPORTANT: all the constraints except referential integrity constraints that reference other tables are retrieved automatically after the table is flashed back.

SCN(System Change Number): The system change number, or SCN, is a numeric stamp that the database automatically increments for every committed transaction that occurs in the database. This includes both explicit and implicit commits, for all external or internal transactions. The SCN is automatically managed by the database in real time. Every committed transaction is assigned an SCN. In order to determine the current SCN at any given moment in the database, use the function DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER

```
SELECT DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER FROM DUAL;
```

Imagine we perform four commits in a table called CHAT, then we can use the following query to check the SCN of these commits

```
SELECT CHAT_ID, ORA_ROWSCN, SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM CHAT;

CHAT_ID                ORA_ROWSCN             SCN_TO_TIMESTAMP(ORA_ROWSCN)
---------------------- ---------------------- -------------------------
1                      5576336                30-JUL-09 10.43.37.000000000 PM
2                      5576336                30-JUL-09 10.43.37.000000000 PM
3                      5576336                30-JUL-09 10.43.37.000000000 PM
4                      5576336                30-JUL-09 10.43.37.000000000 PM
```

```
FLASHBACK TABLE HOUDINI TO SCN 5576336;
```

RESTORE POINT:

```
CREATE RESTORE POINT balance_acct_01;
FLASHBACK TABLE HOUDINI TO RESTORE POINT balance_acct_01;
DROP RESTORE POINT balance_acct_01;
```

TIMESTAMP[4]:

```
 FLASHBACK TABLE uwclass.t TO TIMESTAMP TO_TIMESTAMP('02-MAY-07
12.51.52.050000');
```

FLASHBACK QUERY (FQ):): SELECT * FROM tablename AS OF TIMESTAMP/SCN …

For querying a table as it existed in the past. The FQ feature is a clause of the SELECT statement involving the keywords AS OF.

- The AS OF clause of SELECT can query data in the table as it existed ASOF a particular time in the past, as defined by a TIMESTAMP value or SCN (see chapter 11), and within the limitations of the undo retention period
- The AS OF clause comes after the FROM and before any WHERE clause that might be used within the SELECT statement

There are two ways for performing a FQ

`SELECT *`
`FROM tablename`
`AS OF TIMESTAMP timestamp_expression;`
	`SELECT *`
	`FROM tablename`
	`AS OF SCN scn_expression;`

IMPORTANT: The timestamp_expression or scn_expression cannot be a subquery.	

FLASHBACK VERSION QUERY (FVQ):SELECT * FROM tab VERSION BETWEEN TIMESTAMP/SCN … AND …

takes the FQ feature a step further. You can display rows from multiple committed versions of the database over a range of time

`SELECT * FROM tablename`
` VERSIONS BETWEEN TIMESTAMP`
`timestamp_expression1`
`AND timestamp_expression2;`
	`SELECT * FROM tablename`
	` VERSIONS BETWEEN SCN scn_expression1`
	`AND scn_expression2;`

[4] Conversion Functions: SCN numbers can be converted into their equivalent TIMESTAMP values, and viceversa. The conversions are not exact, however, because the SCN and TIMESTAMP do not represent moments in time that are precisely identical. The conversion functions are
n SCN_TO_TIMESTAMP(s1) : Takes an SCN expression as input, returns a TIMESTAMP value roughly corresponding to when the SCN was set.
n TIMESTAMP_TO_SCN(t):Takes a TIMESTAMP expression representing a valid past or present timestamp as input, returns an SCN value roughly corresponding to when the TIMESTAMP occurred.

- The VERSIONS BETWEEN clause can display rows as they existed in their various states of changes within a range of time
- The VERSIONS BETWEEN clause marks time ranges in terms of SCN or timestamp values
- The VERSIONS BETWEEN clause activates a number of pseudo columns to identify the time range and other data associated with the historic data returned by the VERSIONS BETWEEN clause

PSEUDOCOLUMN	EXPLANATION
VERSIONS_STARTTIME VERSIONS_STARTSCN	Starting time or SCN for when the version of the row was created. If NULL, then the row version was created before the lower time boundary returned by the BETWEEN clause.
VERSIONS_ENDTIME VERSIONS_ENDSCN	Expiration time or SCN for the version of the row. If NULL, then the row version is still current, or the row version resulted from a DELETE (see VERSIONS_OPERATION).
VERSIONS_XID	Identifies the transaction that created the row. (Useful for Flashback Transaction Query.)
VERSIONS_OPERATION	Identifies the operation that performed whatever change created the row version: either I for INSERT, U for UPDATE, or D for DELETE.

```
SELECT CHAT_ID, VERSIONS_STARTSCN, VERSIONS_ENDSCN, VERSIONS_OPERATION
FROM CHAT VERSIONS BETWEEN TIMESTAMP MINVALUE
AND MAXVALUE ORDER BY CHAT_ID, VERSIONS_OPERATION DESC;
```

- **IMPORTANT**: Note that you cannot use the VERSIONS clause when querying a view. But you can use SELECT with a VERSIONS clause to create a view, meaning that VERSIONS can be included in the subquery of a CREATE VIEW statement.

FLASHBACK TRANSACTION QUERY (FTQ): it is a query on the data dictionary view FLASHBACK_TRANSACTION_QUERY.

```
DESC FLASHBACK_TRANSACTION_QUERY;

XID RAW(8)
START_SCN NUMBER
START_TIMESTAMP DATE
COMMIT_SCN NUMBER
COMMIT_TIMESTAMP DATE
LOGON_USER VARCHAR2(30)
UNDO_CHANGE# NUMBER
OPERATION VARCHAR2(32)
TABLE_NAME VARCHAR2(256)
TABLE_OWNER VARCHAR2(32)
ROW_ID VARCHAR2(19)
UNDO_SQL VARCHAR2(4000)
```

- The latest column (UNDO_SQL) contains the SQL statements that can undo a particular transaction, which is defined by first column XID. This is the global transaction identifier.Each transaction that is executed within the database is tracked and assigned a global transaction identifier, which is essentially a transaction identification number. The XID value is of the RAW datatype, which is a binary value, and is not interpreted by the Oracle database. But it can be converted into readable form using the RAWTOHEX function, which converts RAW data into character data that represents the hexadecimal equivalent of the RAW data, which is binary. The function HEXTORAW converts character data containing hexadecimal notation back into the RAW datatype.

- How to get the XID? It is one of the pseudocolumns of the FLASHBACK VERSION QUERY (FVQ) (see above).
- In summary, the data dictionary view FLASHBACK_TRANSACTION_QUERY can be used with the VERSIONS BETWEEN clause to identify metadata associated with a particular transaction that caused the changes to a particular version of the row returned by the VERSIONS BETWEEN clause

```
SELECT UNDO_SQL
FROM FLASHBACK_TRANSACTION_QUERY
WHERE XID = (SELECT VERSIONS_XID
FROM CHAT
VERSIONS BETWEEN TIMESTAMP MINVALUE
AND MAXVALUE
WHERE CHAT_ID = 1
AND VERSIONS_OPERATION = 'D');
```

??????????? DIFFERENCE BETWEEN SCN AND XID ???

IMPORTANT: You cannot roll back a FLASHBACK TABLE statement.

Hierarchical Retrieval (C16)

Interpret the Concept of a Hierarchical Query

- A hierarchical query extends the parent-child relationship into a structure that can be multigenerational, in which multiple levels of relationships maybe added to a given table so that each row may form a relationship at a new generational level beyond the typical parent-child-grandchild-etc. relationship

- Hierarchical queries are based on a self-join table

- All rows in a hierarchical query represent a node

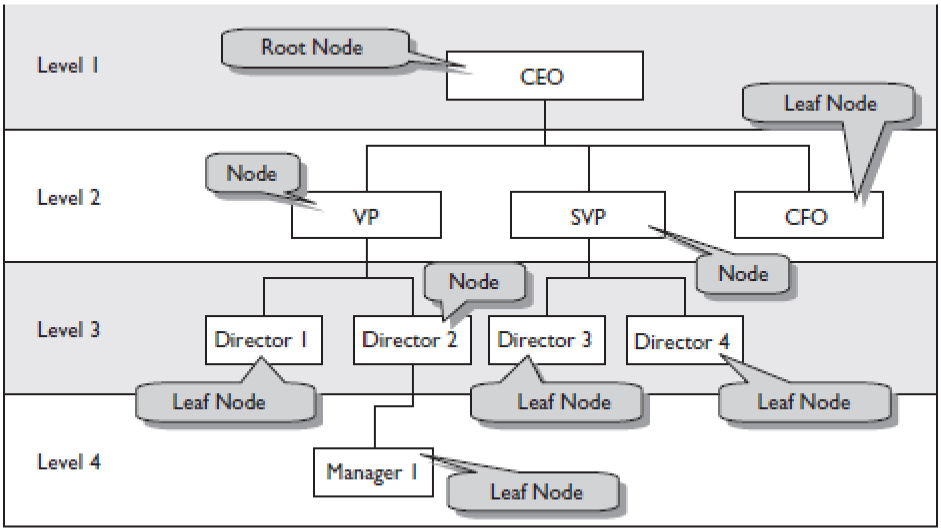
- The starting point of the hierarchical query is the root node

- Any node that branches into two or more children is a fork

- Any node that ends with no children is a leaf node

FIGURE 16-1

Architecture of a hierarchical join



Create and Format Hierarchical Data : SELECT LEVEL, title FROM employee START WITH employee_id=1 CONNECT BY reports_to = PRIOR employee_id (ORDER SIBLINGS BY TITLE)

- The SELECT statement clauses START WITH and CONNECT BY are used to form a hierarchical query:
  - The START WITH clause identifies the root node
  - The CONNECT BY clause defines the self-join relationships
  - <u>There must be at least one use of the PRIOR keyword in the CONNECT BY, according to Oracle's documentation</u>
  - The PRIOR row determines the direction of the hierarchical query
  - The pseudo column LEVEL identifies the generational level from the root node: that is key of the hierarchical query

<u>In summary: for a given row (node), CONNECT BY / PRIOR establishes the nodes that are above (prior) and below of the hierarchy. LEVEL gives the level on the hierarchy</u>

```
SELECT LEVEL, EMPLOYEE_ID, TITLE
FROM EMPLOYEE_CHART
START WITH EMPLOYEE_ID = 1
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID;
```

The output of this query looks like this:

```
LEVEL            EMPLOYEE_ID            TITLE
---------------- ---------------------- --------------------
1                1                       CEO
2                2                       VP
3                5                       Director 1
3                6                       Director 2
4                9                       Manager 1
2                3                       SVP
3                7                       Director 3
3                8                       Director 4
2                4                       CFO
```

- <u>Create a Tree-Structured Report</u>:

```
SELECT LEVEL, EMPLOYEE_ID, LPAD(' ', LEVEL*2) || TITLE TITLE
FROM EMPLOYEE_CHART
START WITH EMPLOYEE_ID = 1
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID;
```

- <u>Choosing Direction:</u> **PRIOR** determine position

```
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID
```

To move in the opposite direction, move the PRIOR keyword to the opposite side of the equal sign, like

```
CONNECT BY PRIOR REPORTS_TO = EMPLOYEE_ID
```

Tip: PRIOR is used to determine what rows appears before or after, as a tip, think as a row

```
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID
```

⟹

PRIOR is where the row is headed. The row means that the direction of the result of the query is from top hierarchy to down, because REPORTS_TO represents to the "boss" and employee_id to the subordinate

The other case works in the same manner

- Remember, to reverse direction in a hierarchical query, move PRIOR to the other side of the equal sign. Also, if you prefix a column name with PRIOR in the SELECT list (SELECT PRIOR EMPLOYEE_ID, ...), you specify the "prior" row's value.

- The ORDER SIBLINGS BY clause sorts rows within a generational level without compromising the default hierarchical ordering of output rows, in other words<u>, it orders by all rows belonging to the same LEVEL</u>

```
SELECT LEVEL, EMPLOYEE_ID, LPAD(' ', LEVEL*2) || TITLE TITLE_FORMATTED
FROM EMPLOYEE_CHART
START WITH EMPLOYEE_ID = 1
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID ORDER SIBLINGS BY TITLE;
```

- The SYS_CONNECT_BY_PATH function can show the complete path to any given node from the root node within a single data element

```
SELECT LEVEL, EMPLOYEE_ID, SYS_CONNECT_BY_PATH(TITLE,'/') TITLE
FROM EMPLOYEE_CHART
START WITH EMPLOYEE_ID = 1
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID;
```

```
LEVEL                  EMPLOYEE_ID            TITLE
---------------------- ---------------------- ------------------------------
1                      1                      /CEO
2                      2                      /CEO/VP
3                      5                      /CEO/VP/Director 1
3                      6                      /CEO/VP/Director 2
4                      9                      /CEO/VP/Director 2/Manager 1
2                      3                      /CEO/SVP
3                      7                      /CEO/SVP/Director 3
3                      8                      /CEO/SVP/Director 4
2                      4                      /CEO/CFO
```

- The CONNECT_BY_ROOT operator can reference a root node row from any row of a hierarchical query

```
SELECT LEVEL, EMPLOYEE_ID, TITLE, CONNECT_BY_ROOT TITLE AS ANCESTOR
FROM EMPLOYEE_CHART START WITH EMPLOYEE_ID = 3
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID;
```

- The order of clauses, if used, in a SELECT statement is SELECT, FROM,WHERE, START WITH, CONNECT BY, and ORDER BY, where the START WITH and CONNECT BY can be interchanged

- A single SELECT statement may define a join and also a CONNECT BY. The join may be defined with the keyword JOIN or with a WHERE clause. If that is the case, the order of processing is: the join (as defined by JOIN or WHERE), then the CONNECT BY, and finally the remaining WHERE clause predicates. When combined with a join, CONNECT BY treats the joined rows as a view, and performs the hierarchical query as if the combined rows were a single table. Note that the pseudocolumn LEVEL should not be given a table alias in a joined query with CONNECT BY.

- if you prefix a column name with PRIOR in the SELECT list (SELECT PRIOR EMPLOYEE_ID, ...), you specify the “prior” row’s value.

Exclude Branches from the Tree Structure

- The CONNECT BY clause can be used to exclude complete branches from the tree

```
SELECT LEVEL, EMPLOYEE_ID, LPAD(' ', LEVEL*2) || TITLE TITLE
FROM EMPLOYEE_CHART START WITH EMPLOYEE_ID = 1
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID AND EMPLOYEE_ID <> 3;
```

- The WHERE clause can exclude individual rows but will not exclude complete branches automatically

```
SELECT LEVEL, EMPLOYEE_ID, LPAD(' ', LEVEL*2) || TITLE TITLE
FROM EMPLOYEE_CHART
WHERE EMPLOYEE_ID NOT IN (SELECT ID FROM FORMER_EMPLOYEES)
START WITH EMPLOYEE_ID = 1
CONNECT BY REPORTS_TO = PRIOR EMPLOYEE_ID AND TITLE <> 'SVP';
```

Regular Expression Support (C17)

Using Metacharacters

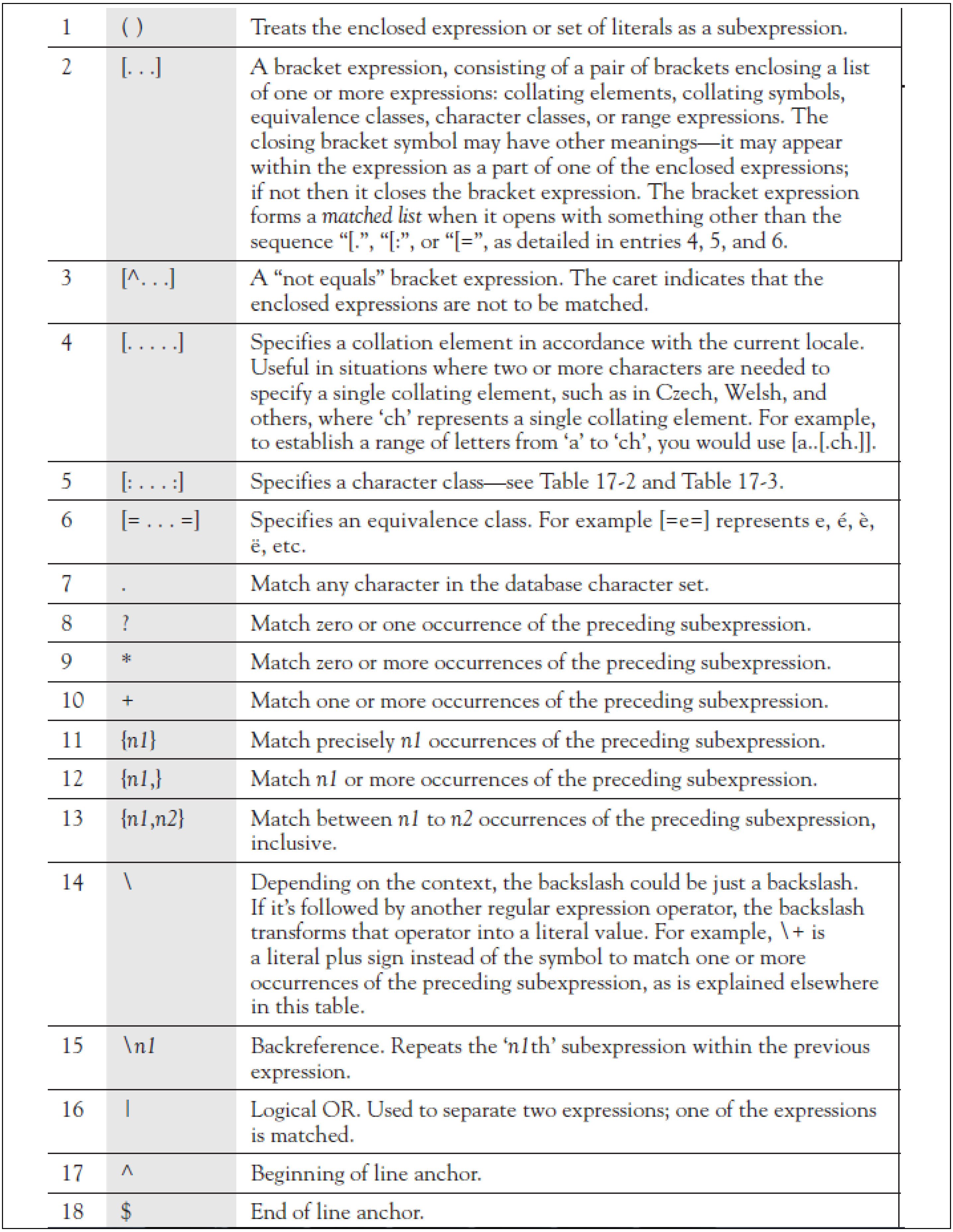
- Metacharacter operators form the foundation of regular expressions
- Regular expression patterns are built with meta character operators
- Depending on the context, certain character literals may be regular expression operators with special capabilities, or they may be character literals
- Regular expressions can include character literals
- Literals enclosed in square brackets represent a set of possible values, or matched list
- Parentheses enclose a grouped expression, or sub expression
- An expression followed by a plus sign, question mark, or asterisk will be interpreted as a pattern that can repeat based on each operator's rules
- You can specify character ranges
- Character classes can serve as an alternative to ranges and provide better support for multilingual applications

Metacharacters

POSIX Characters

Character Ranges

Pattern Matches



1	( )	Treats the enclosed expression or set of literals as a subexpression.
2	[. . .]	A bracket expression, consisting of a pair of brackets enclosing a list of one or more expressions: collating elements, collating symbols, equivalence classes, character classes, or range expressions. The closing bracket symbol may have other meanings—it may appear within the expression as a part of one of the enclosed expressions; if not then it closes the bracket expression. The bracket expression forms a *matched list* when it opens with something other than the sequence "[.", "[:", or "[=", as detailed in entries 4, 5, and 6.
3	[^. . .]	A "not equals" bracket expression. The caret indicates that the enclosed expressions are not to be matched.
4	[. . . . .]	Specifies a collation element in accordance with the current locale. Useful in situations where two or more characters are needed to specify a single collating element, such as in Czech, Welsh, and others, where 'ch' represents a single collating element. For example, to establish a range of letters from 'a' to 'ch', you would use [a..[.ch.]].
5	[: . . . :]	Specifies a character class—see Table 17-2 and Table 17-3.
6	[= . . . =]	Specifies an equivalence class. For example [=e=] represents e, é, è, ë, etc.
7	.	Match any character in the database character set.
8	?	Match zero or one occurrence of the preceding subexpression.
9	*	Match zero or more occurrences of the preceding subexpression.
10	+	Match one or more occurrences of the preceding subexpression.
11	{*n1*}	Match precisely *n1* occurrences of the preceding subexpression.
12	{*n1*,}	Match *n1* or more occurrences of the preceding subexpression.
13	{*n1*,*n2*}	Match between *n1* to *n2* occurrences of the preceding subexpression, inclusive.
14	\	Depending on the context, the backslash could be just a backslash. If it's followed by another regular expression operator, the backslash transforms that operator into a literal value. For example, \+ is a literal plus sign instead of the symbol to match one or more occurrences of the preceding subexpression, as is explained elsewhere in this table.
15	\n1	Backreference. Repeats the '*n1*th' subexpression within the previous expression.
16	|	Logical OR. Used to separate two expressions; one of the expressions is matched.
17	^	Beginning of line anchor.
18	$	End of line anchor.

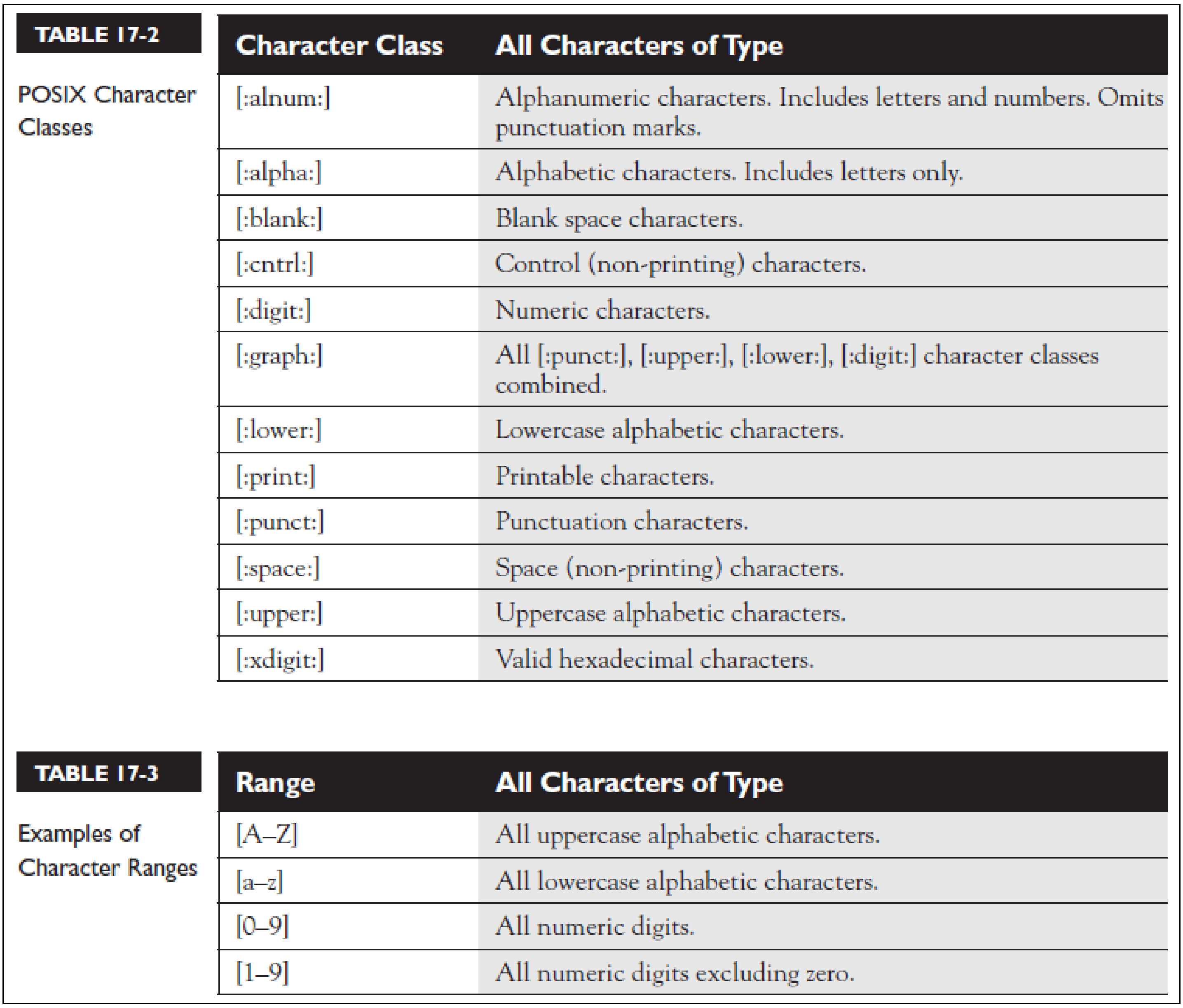


TABLE 17-2

POSIX Character Classes

Character Class	All Characters of Type
[:alnum:]	Alphanumeric characters. Includes letters and numbers. Omits punctuation marks.
[:alpha:]	Alphabetic characters. Includes letters only.
[:blank:]	Blank space characters.
[:cntrl:]	Control (non-printing) characters.
[:digit:]	Numeric characters.
[:graph:]	All [:punct:], [:upper:], [:lower:], [:digit:] character classes combined.
[:lower:]	Lowercase alphabetic characters.
[:print:]	Printable characters.
[:punct:]	Punctuation characters.
[:space:]	Space (non-printing) characters.
[:upper:]	Uppercase alphabetic characters.
[:xdigit:]	Valid hexadecimal characters.

TABLE 17-3

Examples of Character Ranges

Range	All Characters of Type
[A–Z]	All uppercase alphabetic characters.
[a–z]	All lowercase alphabetic characters.
[0–9]	All numeric digits.
[1–9]	All numeric digits excluding zero.

NOTE: Character classes such as [:alpha:] are preferable to letter ranges such as [a–z] in multilingual environments for consistency and flexibility in your applications.

```
Match pattern attribute, it is used in the regular expression functions
'c' = case sensitive
'i' = case insensitive search
'm' = treats the source string as multiple lines
'n' = allows the period (.) wild character to match newline
'x' = ignore whitespace characters
```

Notes:

- [abc] = a | b | c

- Character classes are supported only within bracket expressions. Thus you cannot write `[:digit:]{3}`, but instead must use `[[:digit:]]{3}`. Another example: [[:alnum:]]123 => matches any alphanumeric followed by “123”

- Regular expressions are greedy. By this we mean that the expression will match the largest string it can. Think of it as the expression parser takes the entire sting and compares it to the pattern. The parser then gives back characters until it finds that the string has no match or if finds the match. Lets use a string ‘0123423434’ If my pattern is ‘.*4’ (zero of more characters followed by the digit 4). The first match will be the entire sting.

- The pattern ‘^St[a-z]*’ matches a string that starts with ‘St’ followed by zero or more lower case letters. The pattern ‘stop$’ only matches “stop” if it is the last word on the line.

- You can even define a set of characters in terms of what it is not. The following example uses [^[:digit:]] to allow for any character other than a digit to separate the groups of a phone number: `SELECT park_name FROM park WHERE REGEXP_LIKE(description,'[[:digit:]]{3}[^[:digit:]][[:digit:]]{4}');` Any time you specify a caret (^) as the first character within a bracket expression, you are telling the regular expression engine to begin by including all possible characters in the set, and to then exclude only those that you list following the caret.

- Bracket expressions can be much more complex and varied than those we've shown so far. For example, the following bracket expression: [[:digit:]A-Zxyza-c] includes all digits, the uppercase letters A through Z, lowercase x, y, and z, and lowercase a through

Regular Expression Functions

<table>
<tr><th colspan="2">REGEXP_SUBSTR (up to 5 parameter, 2 mandatory)</th></tr>
<tr><td>Syntax</td><td>REGEXP_SUBSTR(source_string, pattern
[, position [, occurrence
[, match_parameter]]])</td></tr>
<tr><td>Searches for a comma followed by one or more occurrences of non-comma characters followed by a comma</td><td>REGEXP_SUBSTR('500 Oracle Parkway, Redwood Shores, CA', ',[^,]+,')
=></td></tr>
<tr><td>Look for http:// followed by a substring of one or more alphanumeric characters and optionally, a period (.)</td><td>REGEXP_SUBSTR('Go to http://www.oracle.com/products and click on database',
'http://([[:alnum:]]+\.?){3,4}/?') =></td></tr>
<tr><td>Extracts try, trying, tried or tries</td><td>REGEXP_SUBSTR('We are trying to make the subject easier.','tr(y(ing)?|(ied)|(ies))')</td></tr>
<tr><td>Extract the 3rd field treating ':' as a delimiter</td><td>REGEXP_SUBSTR('system/pwd@orabase:1521:sidval',
'[^:]+', 1, 3)</td></tr>
<tr><td>Extract from string with vertical bar delimiter</td><td>CREATE TABLE regexp (testcol VARCHAR2(50));

INSERT INTO regexp(testcol) VALUES ('One|Two|Three|Four|Five');

SELECT * FROM regexp;

SELECT REGEXP_SUBSTR(testcol,'[^|]+', 1, 3) FROM regexp;</td></tr>
<tr><td>Equivalence classes</td><td>REGEXP_SUBSTR('iSelfSchooling NOT ISelfSchooling',
'[[=i=]]SelfSchooling') =></td></tr>
<tr><td>Parsing Demo</td><td>set serveroutput on
DECLARE
 x VARCHAR2(2); y VARCHAR2(2);
 c VARCHAR2(40) := '1:3,4:6,8:10,3:4,7:6,11:12';
BEGIN
  x := REGEXP_SUBSTR(c,'[^:]+', 1, 1);
  y := REGEXP_SUBSTR(c,'[^,]+', 3, 1);

  dbms_output.put_line(x ||' '|| y);
END;
/</td></tr>
</table>

REGEXP_INSTR(up to 7 parameter, 2 mandatory)	
Syntax	REGEXP_INSTR(<source_string>, <pattern>[[, <start_position>][, <occurrence>][, <return_option>][, <match_parameter>][, <sub_expression>]])
Find character 'o' followed by any 3 alphabetic characters: case insensitive Our thanks to Cassio for spotting a typo here.	REGEXP_INSTR('500 Oracle Pkwy, Redwood Shores, CA', '[o][[:alpha:]]{3}', 1, 1, 0, 'i') => 5 REGEXP_INSTR('500 Oracle Pkwy, Redwood Shores, CA', '[o][[:alpha:]]{3}', 1, 1, 1, 'i') => 9 REGEXP_INSTR('500 Oracle Pkwy, Redwood Shores, CA', '[o][[:alpha:]]{3}', 1, 2, 0, 'i') => 28 REGEXP_INSTR('500 Oracle Pkwy, Redwood Shores, CA', '[o][[:alpha:]]{3}', 1, 2, 1, 'i') => 32
Find the position of try, trying, tried or tries	SELECT REGEXP_INSTR('We are trying to make the subject easier.', 'tr(y(ing)?|(ied)|(ies))')=>

Using Sub-Expression option	SELECT testcol, REGEXP_INSTR(testcol, 'ab', 1, 1, 0, 'i', 0) FROM test;
	SELECT testcol, REGEXP_INSTR(testcol, 'ab', 1, 1, 0, 'i', 1) FROM test;
	SELECT testcol, REGEXP_INSTR(testcol, 'a(b)', 1, 1, 0, 'i', 1) FROM test;

REGEXP_LIKE	
Syntax	REGEXP_LIKE(<source_string>, <pattern>, <match_parameter>)

AlphaNumeric Characters	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:alnum:]]');
	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:alnum:]]{3}');
	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:alnum:]]{5}');

Alphabetic Characters	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:alpha:]]');
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,

	'[[:alpha:]]{3}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:alpha:]]{5}');

Control Characters	INSERT INTO test VALUES ('zyx' || CHR(13) || 'wvu');
	COMMIT;
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:cntrl:]]{1}');

Digits	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:digit:]]');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:digit:]]{3}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:digit:]]{5}');

Lower Case	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:lower:]]');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:lower:]]{2}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:lower:]]{3}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:lower:]]{5}');

Printable Characters	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:print:]]{5}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:print:]]{6}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:print:]]{7}');

Punctuation	TRUNCATE TABLE test;
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:punct:]]');

Spaces	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:space:]]');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol,
	'[[:space:]]{2}');

	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:space:]]{3}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:space:]]{5}');

Upper Case	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:upper:]]');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:upper:]]{2}');
	
	SELECT * FROM test WHERE REGEXP_LIKE(testcol, '[[:upper:]]{3}');

Values Starting with 'a%b'	SELECT testcol FROM test WHERE REGEXP_LIKE(testcol, '^ab*');
'a' is the third value	SELECT testcol FROM test WHERE REGEXP_LIKE(testcol, '^..a.');
Contains two consecutive occurances of the letter 'a' or 'z'	SELECT testcol FROM test WHERE REGEXP_LIKE(testcol, '([az])\1', 'i');
Begins with 'Ste' ends with 'en' and contains either 'v' or 'ph' in the center	SELECT testcol FROM test WHERE REGEXP_LIKE(testcol, '^Ste(v|ph)en$');

Use a regular expression in a check constraint	CREATE TABLE mytest (c1 VARCHAR2(20),
	CHECK (REGEXP_LIKE(c1, '^[[:alpha:]]+$')));

Identify SSN
 	
Thanks: Byron Bush HIOUG
	CREATE TABLE ssn_test (ssn_col VARCHAR2(20));
	
	INSERT INTO ssn_test VALUES ('111-22-3333');
	INSERT INTO ssn_test VALUES ('111=22-3333');
	INSERT INTO ssn_test VALUES ('111-A2-3333');
	INSERT INTO ssn_test VALUES ('111-22-33339');
	INSERT INTO ssn_test VALUES ('111-2-23333');
	INSERT INTO ssn_test VALUES ('987-65-4321');
	COMMIT;
	
	SELECT ssn_col from ssn_test
	WHERE REGEXP_LIKE(ssn_col,'^[0-9]{3}-[0-9]{2}-[0-9]{4}$');

Replacing Patterns

- The REGEXP_REPLACE function can replace substrings within a target string using regular expressions

REGEXP_REPLACE	
Syntax	REGEXP_REPLACE(<source_string>, <pattern>, <replace_string>, <position>, <occurrence>, <match_parameter>)
Looks for the pattern xxx.xxx.xxxx and reformats pattern to (xxx) xxx-xxxx	col testcol format a15 col result format a15 SELECT testcol, REGEXP_REPLACE(testcol, '([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})', '(\1) \2-\3') RESULT FROM test WHERE LENGTH(testcol) = 12;

Put a space after every character	SELECT testcol, REGEXP_REPLACE(testcol, '(.)', '\1 ') RESULT
	FROM test WHERE testcol like 'S%';

Replace multiple spaces with a single space	SELECT REGEXP_REPLACE('500 Oracle Parkway, Redwood
	Shores, CA', '( ){2,}', ' ') RESULT FROM DUAL;

Insert a space between a lower case character followed by an upper case character	SELECT REGEXP_REPLACE('George McGovern',
	'([[:lower:]])([[:upper:]])', '\1 \2') CITY FROM DUAL;
	(Produces 'George Mc Govern')

Replace the period with a string (note use of '\')	SELECT REGEXP_REPLACE('We are trying to make the subject easier.','\.',' for you.') REGEXT_SAMPLE FROM DUAL;

Demo	CREATE TABLE t(
	testcol VARCHAR2(10));
	
	INSERT INTO t VALUES ('1');
	INSERT INTO t VALUES ('2 ');
	INSERT INTO t VALUES ('3 new ');
	
	col newval format a10
	
	SELECT LENGTH(testcol) len, testcol origval,
	REGEXP_REPLACE(testcol, '\W+$', ' ') newval,
	LENGTH(REGEXP_REPLACE(testcol, '\W+$', ' ')) newlen
	FROM t;

- The use of regular expressions with a task like string replacement is a much more powerful alternative to the use of a function such as REPLACE, which doesn't support regular expressions

- The regular expression back reference operator can be used as the third parameter to replace a pattern, and to specify grouped expressions within the pattern as part of the replacement

Regular Expressions and CHECK Constraints

- You can create CHECK constraints that use regular expressions

```
CREATE TABLE EMAIL_LIST(
      EMAIL_LIST_ID NUMBER(7) PRIMARY KEY,
      EMAIL1 VARCHAR2(120),
      CONSTRAINT CK_EL_EMAIL1
      CHECK (
      REGEXP_LIKE
         (EMAIL1,'^([[:alnum:]]+)@[[:alnum:]]+.(com|net|org|edu|gov|mil)$')
      )
);
```

- CHECK constraints can use regular expression patterns to define restrictions and requirements on incoming data for a given table

- The REGEXP_LIKE condition is useful in applying the CHECK constraint to a given table

Others functions

REGEXP_COUNT	
Syntax	`REGEXP_COUNT(<source_string>, <pattern>[[, <start_position>], [<match_parameter>]])`

Count's occurrences based on a regular expression	`SELECT REGEXP_COUNT(testcol, '2a', 1, 'i') RESULT FROM test;`
	`SELECT REGEXP_COUNT(testcol, 'e', 1, 'i') RESULT FROM test;`

IMPORTANT: The regular expression functions are just like any other function— and may be used in any SQL statement and clause that accepts any valid SQL function

Controlling User Access (C18)

Differentiate System Privileges (perform a task: CREATE_TABLE) from Object Privileges (on specific objects such UPDATE a table)

- SYSTEM PRIVILEGE: The right to use any given SQL statement and/or to generally perform a task in the database is a system privilege, for example **CREATE_TABLE** (which gives the right to create a table in your user account. Includes ability to ALTER and DROP TABLE. Also includes ability to CREATE, ALTER, and DROP INDEX objects), **CREATE_ANY_TABLE** (which gives the right to create any table, regardless the user account). NOTE: An important system privilege is **CREATE SESSION** which allows a user to connect to DB
- OBJECT PRIVILEGE: The right to use a system privilege to perform some task on a specific existing object in the database is an object privilege, for example the ability to change rows of data in, for example, a table called BENEFITS owned by a user account named EUNICE—is an object privilege.

GRANT privilege TO user_option / REVOKE privilege FROM user

SYSTEM PRIVILEGE	OBJECT PRIVILEGE
• Both system and object privileges are granted to and revoked from users in the database o `GRANT privilege TO user option;` o `REVOKE privilege FROM user;`	

• System privileges may be granted WITH ADMIN OPTION, which provides the ability for the recipient to grant the same privilege to yet another user
`GRANT privilege TO user WITH ADMIN OPTION;`
• When a system privilege is revoked, the revocation **does not cascade**—meaning that it is only revoked from the user from whom it is being revoked, not from other users to whom the revoked user may have extended the privilege
	• The GRANT statement has the “ON”,
	`GRANT SELECT,UPDATE(customer_id, order_total)` **ON** `orders TO hr;-- NOTE THE POSSIBILITY TO UPDATE ONLY OVER 2 COLUMNS`
	• Object privileges may be granted WITH GRANT OPTION, which provides the ability for the recipient to grant the same privilege to yet another user
	`GRANT SELECT, UPDATE` **ON** `WEBINARS TO HENRY WITH GRANT OPTION:`
	• When an object privilege is revoked, the revocation **cascades** (CONTRARILY TO SYSTEM PRIVILEGE)—meaning that it is revoked from the user from whom it is being revoked, as well as from other users to whom the revoked user may have extended the privilege
	`REVOKE SELECT, UPDATE ON WEBINARS FROM`

	`HENRY;`

The **ALL PRIVILEGES** keywords can be used to grant or revoke all privileges to or from a user: as an alternative to granting specific system privileges, a qualified user account, such as SYSTEM or some other DBA qualified account, can issue the following statement:
`GRANT ALL PRIVILEGES TO user;`
This statement has the effect of granting all system privileges to the user. The WITH ADMIN OPTION clause may be used with this as well. Needless to say, this should only be done with great caution
	SAME but The keyword PRIVILEGES is not required when granting object privileges:
	`GRANT ALL (PRIVILEGES) ON WEBINARS TO HENRY;`
	The same is true with REVOKE when used with object privileges:
	`REVOKE ALL (PRIVILEGES) ON WEBINARS FROM HENRY;`
	CAN WE USE THE with grant option ??????

PUBLIC ACCOUNT: is a built-in user account in the Oracle database that represents all users. Any objects owned by PUBLIC are treated as though they are owned by all the users in the database, present and future.

- The GRANT statement will work with the keyword PUBLIC in the place of a user account name

```
GRANT CREATE ANY TABLE TO PUBLIC;
```

Note that if you wish to grant all privileges, you use the keywords ALL PRIVILEGES. But if you wish to grant certain privileges to all users, you do not use the keyword ALL. Instead, you grant to PUBLIC.

You can only grant access to one object at a time:

```
GRANT ALL PRIVILEGES ON WEBINARS,TABLE2,TABLE TO HENRY; -- wrong only 1 object
```

System privileges and object privileges cannot be granted together in a single GRANT statement

```
GRANT CREATE TABLE, SELECT ON OE.ORDERS TO HENRY – wrong, mixed privileges is not allowed
```

Grant Privileges on Tables: Any user with the system privilege CREATE TABLE can create a table. The table, once created, is owned by the user who created it. The owner does not require any explicitly granted privileges on the table. The table owner can use DML to add rows, change data in the table, query the data in the table, and remove rows from the table. But other users do not have that privilege automatically. Other users must have explicitly granted priviliges on the object—which, in this case, is a table.(Note: The exception, of course, is those users who have the system privileges that allow them to run any DML statements on any table in the database, regardless of who owns it—those system privileges, as we saw in the last section, include SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, and DELETE ANY TABLE.)

Suppose user 'LISA' it's the owner of table WEBMINARS

```
GRANT SELECT ,UPDATE  ON WEBMINARS TO HENRY
```

Henry must include the schema prefix in order to SELECT or update WEBMINARS

```
SELECT * FROM LISA.WEBINARS;
```

However, the user SYSTEM can grant LISA to create a public synomym

```
GRANT CREATE PUBLIC SYNONYM TO LISA;
```

Now Lisa can create the public synonym and therefore HENRY will be able to SELECT or update just doing 'SELECT * FROM WEBMINARS'

```
CREATE PUBLIC SYNONYM WEBINARS FOR LISA.WEBINARS;
```

Remember name priority (chapter 10):

- First, SQL looks in the local namespace, which contains objects owned by the user account: tables, views, sequences, private synonyms, and something called user-defined types—which are beyond the scope of the exam.
- Next, SQL looks in the database namespace, which contains users, roles, and public synonyms.
- Object privileges correspond to DML statements, and to DDL statements that are relevant to existing objects
- When a user has been granted access to an object, the object name will require a schema name prefix to be correctly identified
- A PUBLIC SYNONYM can provide an alternative name for the schema prefixed version of the granted object

Typical Question: If you grant privileges on a table, then drop the table, the privileges are dropped with the table. If you later recreate the table, you must also grant the privileges again. However, if you restore a dropped table with the FLASHBACK TABLE . . . BEFORE DROP statement, you will recover the table, its associated indices, and the table's granted privileges, and you will not need to grant the privileges again.

View Privileges in the Data Dictionary: USER_SYS_PRIVS, DBA_SYS_PRIVS, USER_TAB_PRIVS,…

- A variety of data dictionary views provide information about system and object privileges
- Users may see privileges granted to them, or granted by them to others, by querying the data dictionary

Data Dictionary View	Explanation
USER_SYS_PRIVS	System privileges granted to current user
DBA_SYS_PRIVS	System privileges granted to users and roles
USER_TAB_PRIVS	Grants on objects for which the user is the grantor, grantee, or owner

ALL_TAB_PRIVS	Grants on objects for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
DBA_TAB_PRIVS	Grants on all objects in the database
ALL_TAB_PRIVS_RECD	Grants on objects for which the user, PUBLIC, or enabled role is the grantee
SESSION_PRIVS	Privileges that are enabled to the user

SELECT PRIVILEGE, ADMIN_OPTION FROM USER_SYS_PRIVS ORDER BY PRIVILEGE;

Typical Question: When inspecting data dictionary views like DBA_TAB_PRIVS or DBA_SYS_PRIVS to see what privileges have been granted to a particular user account, you can check the GRANTEE column for the appropriate USER name. However, don't forget to also check for rows where GRANTEE = 'PUBLIC'; these privileges are also available to your user account.

ROLES = collection of privileges and other roles

CREATE ROLE CRUISE_ANALYST; GRANT SELECT ON SHIPS TO CRUISE_ANALYST (WITH ADMIN OPTION)

Grant Roles

- A role is created with the CREATE ROLE statement. A ROLE is a database object that you can create, and to which you can assign system privileges and/or object privileges

```
CREATE ROLE CRUISE_ANALYST;

GRANT SELECT ON SHIPS TO CRUISE_ANALYST;

GRANT SELECT ON PORTS TO CRUISE_ANALYST;

GRANT SELECT ON EMPLOYEES TO CRUISE_ANALYST;
```

Once they are created, we can grant these roles to user accounts in the database:

```
GRANT CRUISE_OPERATOR TO LISA;

GRANT CRUISE_ANALYST TO HENRY;
```

- Roles may be granted WITH ADMIN OPTION, which provides the ability for the recipient to grant the same role to yet another user

```
GRANT CRUISE_OPERATOR TO LISA WITH ADMIN OPTION;
```

- Roles exist in a namespace outside of an individual user account. A role may be granted to another role

Typical Question: Let's say you create an object, then grant a privilege on that object to a role, and then grant the role to a user. If you drop the object, then you also drop the granted object privilege to the role. However, the role still exists, and the grant of the role to the user still exists. If you subsequently re-create the object, then grant the object privilege to the role once again, then you've re-created the situation before the object was dropped—in other words, you do not need to re-create the role, nor grant the role to the user once again, since neither was affected by the act of dropping the object on which the privilege had originally been granted.

```
CREATE TABLE X (...); -- creation of table T
CREATE ROLE RX; -- creation of Role RX
GRANT RX TO USER1 -- grant role rx to 'PEPE'
GRANT INSERT ON X TO RX; -- grant permission to insert to table X to rolex RX
DROP TABLE X;
-- AT THIS POINT ROLE STILL EXISTS AND STILL IS ASSIGNED TO USER 'PEPE'
CREATE TABLE X(...)
GRANT INSERT ON X TO RX; -- grant permission to insert to table X to rolex RX
-- no need to recreate the role, nor to assign it to user 'PEPE'
```

Typical Question: If a user has been granted a privilege directly and "indirectly" via ROLE, then if the privilege is REVOKE, the user still "keeps" that privilege via ROLE, for example:

```
CREATE ROLE r1;
GRANT SELECT, INSERT ON oe.orders TO r1;
GRANT r1 TO scott;
GRANT SELECT ON oe.orders TO scott;
REVOKE SELECT ON oe.orders FROM scott; -- SCOTT would be able to query the OE.ORDERS
```

`In other words:` A privilege may be granted directly as a privilege, or indirectly as part of a role. If you intend to DROP a privilege from a USER, use the data dictionary to determine if that same privilege is granted to a ROLE that is also granted to the same USER—if so, then the privilege you dropped directly is still granted to the USER indirectly through the ROLE.

Distinguish Between Privileges and Roles

- A privilege granted directly to a user exists independently from a privilege granted to a role, If you revoke a privilege directly from a user who also has been granted a role containing the same privilege, the role remains unchanged and the user still has privileges by way of the role. The same situation is true with regard to revoking privileges directly from roles; if you revoke a privilege from a role that a user already has through a direct grant, the direct grant stays in force

- In summary: remember that "privileges" may refer to either system privileges or object privileges, which are very different. Roles consist of some combination of one or more system and/or object privileges and/or other roles.

SQL Functions

Table 4-9: Oracle-Supported Functions

Function	Description
abs(number)	Returns the absolute value of ***number.***
acos(number)	Returns the arc cosine of ***number*** ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians.
ascii(string)	Returns the decimal value in the database character set of the first character of ***string***; returns an ASCII value when the database character set is 7-bit ASCII; returns EBCDIC values if the database character set is EBCDIC Code Page 500.
asin(number)	Returns the arc sine of ***number*** ranging from -1 to 1. The resulting value ranges from **-**π /2 to π/2 and is expressed in radians.
atan(number)	Returns the arctangent of any ***number.*** The resulting value ranges from **-**π /2 to p/2 and is expressed in radians.
atan2(number,nbr)	Returns the arctangent of ***number*** and ***nbr.*** The values for ***number*** and ***nbr*** are not restricted, but the results range from **-**π to π and are expressed in radians.
avg([DISTINCT] expression) over (analytics)	Returns the average value of ***expr***. It can be used as an aggregate or analytic function (analytic functions are beyond the scope of this text).
bfilename(`directory',' filename')	Returns a ***BFILE*** locator associated with a physical LOB binary ***filename*** on the server's filesystem in ***directory.***
ceil(number)	Returns smallest integer greater than or equal to ***number.***
chartorowid(char)	Converts a value from a character datatype (***CHAR*** or ***VARCHAR2*** datatype) to ***ROWID*** datatype.
chr(number [USING NCHAR_CS])	Returns the character having the binary equivalent to ***number*** in either the database character set (if ***USING NCHAR_CS*** is not included) or the national character set (if ***USING NCHAR_CS*** is included).
convert(char_value, target_char_set, source_char_set)	Converts a character string from one character set to another; returns the ***char_value*** in the ***target_char_set*** after converting ***char_value*** from the ***source_char_set.***
corr(expression1, expression2) over (analytics)	Returns the correlation coefficient of a set of numbered pairs (***expressions*** 1 and 2). It can be used as an aggregate or analytic function (analytic functions are beyond the scope of this text).
cos(number)	Returns the cosine of ***number*** as an angle expressed in radians.
cosh(number)	Returns the hyperbolic cosine of ***number.***
count	Returns the number of rows in the query; refer to the earlier section on ***COUNT*** for more information.
covar_pop(expression1, expression2) over (analytics)	Returns the population covariance of a set of number pairs (***expressions*** 1 and 2). It can be used as an aggregate or analytic function (analytic functions are beyond the scope of this text).
covar_samp(expression1, expression2)	Returns the sample covariance of a set of number pairs (***expressions*** 1 and 2). It can be

over(analytics)	used as an aggregate or analytic function (analytic functions are beyond the scope of this text).
cume_dist( ) ([OVER (query)] ORDER BY...)	The cumulative distribution function computes the relative position of a specified value in a group of values.
dense_rank( ) ([OVER (query)] ORDER BY...)	Computes the rank of each row returned from a query with respect to the other rows, based on the values of the ***value_exprs*** in the ***ORDER_BY_clause.***
deref(expression)	Returns the object reference of ***expression***, where ***expression*** must return a ***REF*** to an object.
dump(expression [,return_ format [, starting_at [,length]]])	Returns a ***VARCHAR2*** value containing a datatype code, length in bytes, and internal representation of ***expression***. The resulting value is returned in the format of ***return_ format***.
empth[B | C]lob( )	Returns an empty LOB locator that can be used to initialize a LOB variable. It can also be used to initialize a LOB column or attribute to empty in an ***INSERT*** or ***UPDATE*** statement.
exp(number)	Returns ***E*** raised to the ***number*** ed power, where E = 2.71828183.
first_value(expression) over (analytics)	Returns the first value in an ordered set of values.
floor(number)	Returns largest integer equal to or less than ***number.***
grouping(expression)	Distinguishes null cause by a super-aggregation in ***GROUP BY*** extension from an actual null value.
hextoraw(string)	Converts ***string*** containing hexadecimal digits into a raw value.
instrb(string1, string2, [start_a[t, occurrence]])	The same as ***INSTR***, except that ***start_at*** and the return value are expressed in bytes instead of characters.
lag(expression [,offset][,default]) over(analytics)	Provides access to more than one row of a table at the same time without a self join; refer to the vendor documentation for more information.
last_value(expression) over (analytics)	Returns the last value in an ordered set of values; refer to the vendor documentation for more information.
lead(expression [,offset][,default]) over(analytics)	Provides access to more than one row of a table at the same time without a self join. Analytic functions are beyond the scope of this text.
lengthb(string)	Returns the length of ***char*** in bytes; otherwise, the same as ***LENGTH.***
ln(number)	Returns the natural logarithm of ***number***, where the ***number*** is greater than 0.
log(base_number, number)	Returns the logarithm of any ***base_number*** of ***number***.
make_ref({table_name | view_name} , key [,...n])	Creates a reference (***REF***) to a row of an object view or a row in an object table whose object identifier is primary key-based.
max([DISTINCT] expression) over (analytics)	Returns maximum value of ***expression***. It can be used as an aggregate or analytic function (analytic functions are beyond the scope of this text).
min([DISTINCT] expression) over (analytics)	Returns minimum value of ***expression***. It can be used as an aggregate or analytic function (analytic functions are beyond the scope of this text).

new_time(date, time_zone1, time_zone2)	Returns the date and time in ***time_zone2*** when date and time in ***time_zone1*** are ***date***. ***Time_zones*** 1 and 2 may be any of these text strings: • AST, ADT: Atlantic Standard or Daylight Time • BST, BDT: Bering Standard or Daylight Time • CST, CDT: Central Standard or Daylight Time • EST, EDT: Eastern Standard or Daylight Time • GMT: Greenwich Mean Time • HST, HDT: Alaska-Hawaii Standard Time or Daylight Time • MST, MDT: Mountain Standard or Daylight Time • NST: Newfoundland Standard Time • PST, PDT: Pacific Standard or Daylight Time • YST, YDT: Yukon Standard or Daylight Time
nls_charset_decl_len(bytecnt, csid)	Returns the declaration width (***bytecnt***) of an ***NCHAR*** column using the character set ID (***csid***) of the column.
nls_charset_id(text)	Returns the NLS character set ID number corresponding to ***text.***
nls_charset_name(number)	Returns the ***VARCHAR2*** name for the NLS character set corresponding to the ID ***number.***
nls_initcap(string [,'nlsparameter'])	Returns ***string*** with the first letter of each word in uppercase and all other letters in lowercase. The ***nlsparameter*** offers special linguistic sorting features.
nls_lower(string, [,'nlsparameter'])	Returns ***string*** with all letters lowercase. The ***nlsparameter*** offers special linguistic sorting features.
nlssort(string [,'nlsparameter'])	Returns the string of bytes used to sort ***string***. The ***nlsparameter*** offers special linguistic sorting features.
nls_upper string [,'nlsparameter'])	Returns ***string*** with all letters uppercase. The ***nlsparameter*** offers special linguistic sorting features.
ntile(expression) over (query_partition ORDER BY...)	Divides an ordered data set into a number of buckets numbered 1 to ***expression*** and assigns the appropriate bucket number to each row.
percent_rank( ) over (query_partition ORDER BY...)	Similar to the ***CUME_DIST*** analytical function. Rather than return the cumulative distribution, it returns the percentage rank of a row compared to the others in its result set. Refer to the vendor documentation for more assistance.
power(number, power)	Returns ***number*** raised to the nth ***power***. The base and the exponent can be any numbers, but if ***number*** is negative, ***power*** must be an integer.
rank (value_expression) over (query_partition ORDER BY ...)	Computes the rank of each row returned from a query with respect to the other rows returned by the query, based on the values of the ***value_expression*** in the ***ORDER_BY_clause***.
ratio_to_report (value_exprs) over (query_partition)	Computes the ratio of a value to the sum of a set of values. If ***values_expr*** is null, the ratio-to-report value also is null.
rawtohex(raw)	Converts a ***raw*** value to a string (character datatype) of its hexadecimal equivalent.
ref(table_alias)	***REF*** takes a table alias associated with a row from a table or view. A special reference value is returned for the object instance that is bound to the variable or row.
reftohex(expression)	Converts argument ***expression*** to a character value containing its hexadecimal equivalent.

regr_ xxx(expression1, expression2) over (analytics)	Linear regression functions fit an ordinary-least-squares regression line to a set of number pairs where ***expression1*** is the dependent variable and ***expression2*** is the independent variable. The linear regression functions are: • REGR_SLOPE: returns the slope of the line • REGR_INTERCEPT: returns the y-intercept of the regression line • REGR_COUNT: returns the number of non-null pairs fitting the regression line • REGR_R2: returns the coefficient of determination for the regression • REGR_AVGX: returns the average of the independent variable • REGR_AVGY: returns the average of the dependent variable • REGR_SXX: calculates ***REGR_COUNT(exp1, exp2) * VAR_POP(exp2)*** • REGR_SYY: calculates ***REGR_COUNT(exp1, exp2) * VAR_POP(exp1)*** • REGR_SXY: calculates ***REGR_COUNT(exp1, exp2) * COVAR_POP(exp1, exp2)*** These can be used as aggregate or analytic functions.
replace(string, search_string [,replacement_string])	Returns ***string*** with every occurrence of ***search_string*** replaced with ***replacement_string.***
row_number ( ) over (query_partition ORDER BY ...)	Assigns a unique number to each row where it is applied in the ordered sequence of rows specified by the ***ORDER_BY_clause***, beginning with 1.
rowidtochar(rowid)	Converts a ***rowid*** value to ***VARCHAR2*** datatype, 18 characters long.
rpad(string1, number [, string2])	Returns ***string1***, right-padded to length ***number*** with the value of ***string2***, repeated as needed. ***String2*** defaults to a single blank.
sign(number)	When ***number*** < 0, returns -1. When ***number*** = 0, returns 0. When ***number*** > 0, returns 1.
sin(number)	Returns the sine of ***number*** as an angle expressed in radians.
sinh(number)	Returns the hyperbolic sine of ***number.***
sqrt(number)	Returns square root of ***number***, a nonnegative number.
stddev([DISTINCT] expression) over (analytics)	Returns sample standard deviation of a set of numbers shown as ***expression.***
stdev_pop(expression) over (analytics)	Computes the population standard deviation and returns the square root of the population variance.
seddev_samp(expression) over (analytics)	Computes the cumulative sample standard deviation and returns the square root of the sample variance.
substrb(extraction_string [FROM starting_position] [FOR length])	***SUBSTRB*** is the same as ***SUBSTR***, except that the arguments ***m starting_position*** and ***length*** are expressed in bytes, rather than in characters.
sum([DISTINCT] expression) over (analytics)	Returns sum of values of ***expr*** ; refer to vendor documentation for assistance with analytics and the ***OVER*** subclause.
sys_context (`namespace','attribute '	Returns the value of ***attribute*** associated with the context ***namespace***, usable in both SQL and PL/SQL statements.

[,length])	
sys_guid( )	Generates and returns a globally unique identifier (***RAW*** value) made up of 16 bytes.
tan(number)	Returns the tangent of ***number*** as an angle expressed in radians.
tanh(number)	Returns the hyperbolic tangent of ***number***
to_lob(long_column)	Usable only by ***LONG*** or ***LONG RAW*** expressions, it converts ***LONG*** or ***LONG RAW*** values in the column ***long_column*** to LOB values. It is usable only in the ***SELECT*** list of a subquery in an ***INSERT*** statement.
to_multi_byte(string)	Returns ***string*** with all of its single-byte characters converted to their corresponding multi-byte characters.
to_single_byte(string)	Returns ***string*** with all of its multi-byte characters converted to their corresponding single-byte characters.
translate(`char_value', `from_text', `to_text')	Returns ***char_value*** with all occurrences of each character in ***from_text*** replaced by its corresponding character in ***to_text***; refer to the section "CONVERT and TRANSLATE" earlier in this chapter for more information on ***TRANSLATE.***
translate (text USING [CHAR_CS | NCHAR_CS])	Converts ***text*** into the character set specified for conversions between the database character set or the national character set.
uid	Returns an integer that uniquely identifies the session user who logged on. No parameters are needed.
user	Returns the name of the session user who logged on in ***VARCHAR2.***
userenv(option)	Returns information about the current session in ***VARCHAR2.***
value(table_alias)	Takes as a table alias associated with a row in an object table and returns object instances stored within the object table.
var_pop(expression) over (analytics)	Returns the population variance of a set of numbers after discarding the nulls in the ***expression*** number set. Analytic functions are covered in the vendor documentation.
var_samp(expression) over (analytics)	Returns the sample variance of a set of numbers after discarding the nulls in the ***expression*** number set. Analytic functions are covered in the vendor documentation.
variance([DISTINCT] expression) over (analytics)	Returns variance of ***expression*** calculated as follows: • 0 if the number of rows in ***expression*** = 1 • ***VAR_SAMP*** if the number of rows in ***expression*** > 1
vsize(expression)	Returns the number of bytes in the internal representation of ***expression***. When ***expression*** is null, it returns null.

http://www.datadisk.co.uk/html_docs/oracle/

http://coskan.wordpress.com/2009/03/05/what-i-learned-during-oracle-sql-expert-exam-study-part-1/

http://coskan.wordpress.com/2009/03/18/what-i-learned-during-oracle-sql-expert-exam-study-part-2/

http://www.giannakidis.info/post/51891048487/1z0-047-oracle-database-sql-expert-beyond-the-exam

The base syntax for declaring a variable is:

identifier [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expr];

An example of a PL/SQL block declaring several variables is:

```
DECLARE
v_serial NUMBER; -- SQL data type
v_part_name VARCHAR2(20); -- SQL data type
v_in_stock BOOLEAN; -- PL/SQL-only data type
v_price NUMBER(6,2); -- SQL data type
v_part_desc VARCHAR2(50); -- SQL data type
v_count PLS_INTEGER NOT NULL := 0; -- PL/SQL-only data type
BEGIN
NULL;
END;
```

```
VARIABLE bv_return_val VARCHAR2(20)
VARIABLE bv_return_code NUMBER
```

The above bind variables can then be referenced from within a PL/SQL block by preceding the variable name with a colon:

```
BEGIN
SELECT emp_job
INTO :bv_return_val
FROM employees
WHERE emp_id=18;
:bv_return_code := 0;
EXCEPTION
WHEN OTHERS THEN
:bv_return_code := 1;
END;
```

The %TYPE attribute lets you declare a variable to be of the same data type as a previously declared variable or column. A variable using %TYPE does not inherit the initial value of the referenced item. If the referencing item specifies NOT NULL or inherits NOT NULL from the referenced item, an initial value must be specified. However, note that PL/SQL variables declared using %TYPE do not inherit the NOT NULL constraint from database columns.

DECLARE
v_last_name employees.emp_last%TYPE;
BEGIN
SELECT emp_last
INTO v_last_name
FROM employees
WHERE emp_id = 9;
DBMS_OUTPUT.PUT_LINE('v_last_name = ' ||
v_last_name);
END;
	DECLARE
	v_salary NUMBER(6,2);
	v_commission v_salary%TYPE;
	v_bonus v_salary%TYPE;
	BEGIN
	NULL;
	END;

```
CREATE SEQUENCE seq_ocp_temp;
DECLARE
v_seqval NUMBER;
BEGIN
v_seqval := seq_ocp_temp.NEXTVAL;
DBMS_OUTPUT.PUT_LINE('v_seqval: ' || v_seqval);
END;
```

```
/
```

If a variable name used in the parent is redeclared by one of the nested subprograms, then inside that particular subprogram,both variables are in scope, but only the local variable is visible

```
<<outer>>
DECLARE
v_a VARCHAR2(1) := 'A';
BEGIN
DECLARE
v_a VARCHAR2(1) := 'a';
BEGIN
DBMS_OUTPUT.PUT_LINE('Inner Block, variable v_a: ' || v_a);
DBMS_OUTPUT.PUT_LINE('Inner Block, global variable outer.v_a: ' ||
outer.v_a);
END;
END;
Inner Block, variable v_a: a
Inner Block, global variable outer.v_a: A
```

PL/SQL does not have direct support for Data Definition Language (DDL) SQL statements or Data Control Language SQL statements. However, DDL and DCL statements can be executed from within PL/SQL using dynamic SQL.

```
DECLARE
v_max_sal NUMBER;
BEGIN
SELECT MAX(salary)
INTO v_max_sal
FROM hr.employees;
DBMS_OUTPUT.PUT_LINE('v_max_sal :' ||
v_max_sal);
END;
v_max_sal :24000
```

```
DECLARE
TYPE EmpRec IS RECORD (
v_first hr.employees.first_name%TYPE,
v_last hr.employees.last_name%TYPE,
v_id hr.employees.employee_id%TYPE);
r_emprec EmpRec;
BEGIN
  SELECT first_name, last_name,
employee_id
 INTO r_emprec
 FROM hr.employees
WHERE employee_id = 122;
DBMS_OUTPUT.PUT_LINE ('Employee #' ||
r_emprec.v_id ||
' = ' || r_emprec.v_first ||
' ' || r_emprec.v_last);
END;
Employee #122 = Payam Kaufling
```

```
DECLARE
v_value NUMBER := 5;
BEGIN
CASE v_value
WHEN 4 THEN
DBMS_OUTPUT.PUT_LINE('v_value = 4');
WHEN 5 THEN
DBMS_OUTPUT.PUT_LINE('v_value = 5');
WHEN 6 THEN
DBMS_OUTPUT.PUT_LINE('v_value = 6');
END CASE;
END;
```

```
DECLARE
v_value NUMBER := 5;
BEGIN
CASE
WHEN v_value = 4 THEN
DBMS_OUTPUT.PUT_LINE('v_value = 4');
WHEN v_value = 5 THEN
DBMS_OUTPUT.PUT_LINE('v_value = 5');
WHEN v_value =6 THEN
DBMS_OUTPUT.PUT_LINE('v_value = 6');
END CASE;
END;
```

EXIT -- Will unconditionally end a loop immediately and pass control
to the first statement in the block following the loop end.
 EXIT WHEN -- When a given condition is true, this will end a loop immediately and pass control to the first statement in the block following the
loop end.
 CONTINUE -- Will unconditionally end the current iteration of a loop
and pass control to the start of the loop for the next iteration.
 CONTINUE WHEN -- When a given condition is true, this will unconditionally end the current iteration of a loop and pass control to the start
of the loop for the next iteration.

```
DECLARE
v_ndxo PLS_INTEGER := 0;
v_ndxi PLS_INTEGER;
BEGIN
<<outer_loop>>
LOOP
  v_ndxi := 1;
  v_ndxo := v_ndxo + 1;
  <<inner_loop>>
  LOOP
    CONTINUE outer_loop WHEN (MOD(v_ndxo, 2) = 0);
   DBMS_OUTPUT.PUT_LINE('v_ndxo: ' ||
TO_CHAR(v_ndxo) ||
 ' --- v_ndxi: ' || TO_CHAR(v_ndxi));
   v_ndxi := v_ndxi + 1;
   EXIT inner_loop WHEN v_ndxi > 2;
  END LOOP inner_loop;
 EXIT WHEN v_ndxo > 6;
END LOOP outer_loop;
END;
```

The FOR loop executes one or more statements when a loop index is in a given range. The basic structure of a FOR loop is:
[label] FOR index IN [REVERSE] lower_bound..upper_bound LOOP
statements
END LOOP [label];
When the REVERSE keyword is not present, the loop index begins at the lower bound and increments by one for each iteration until it reaches the upper bound. When the REVERSE keyword is used, the index runs from the upper bound to the lower, decrementing by 1. For PL/SQL for loops, the lower number is always to the left. Reversing the order of the numbers will result in a loop that never runs.

COMPOSITE DATA TYPES:

PL/SQL RECORDS: Record variables allow you to store multiple separate but related pieces of information in a single construct. There are three ways to define a record:

1) Define a RECORD type and then declare a variable using that type.

```
DECLARE

TYPE r_emprectyp IS RECORD (
emp_id NUMBER,
emp_first VARCHAR2(30),
emp_last VARCHAR2(30)
);
  r_emp_rec1 r_emprectyp;
BEGIN
  r_emp_rec1.emp_id := 103;
  r_emp_rec1.emp_first := 'John';
  r_emp_rec1.emp_last := 'Jones';
```

2) Use %TYPE to declare a record variable as a previously declared record variable type.

```
<previous example>    r_emp_rec2 r_emp_rec1%TYPE;
```

3) Use %ROWTYPE to declare a record variable to match part or all of a row in a database table or view

- The record fields do not inherit the constraints of the columns. A NOT NULL column will not produce a NOT NULL field.

- The record fields do not inherit initial values of the corresponding columns. Record values will all default to NULL.

```
DECLARE
r_ctry_rec hr.countries%ROWTYPE;
BEGIN
  -- Assign values to fields:
  r_ctry_rec.country_id := 'US';
  r_ctry_rec.country_name := 'United States';
  r_ctry_rec.region_id := 2;
```

over partial table or view

```
DECLARE
CURSOR c_emp IS
SELECT first_name, last_name, job_id
FROM hr.employees;
r_emp_jobs c_emp%ROWTYPE;
BEGIN
r_emp_jobs.first_name := 'Fred';
r_emp_jobs.last_name := 'Rogers';
r_emp_jobs.job_id := 'IT_PROG';
DBMS_OUTPUT.PUT_LINE (r_emp_jobs.first_name);
DBMS_OUTPUT.PUT_LINE (r_emp_jobs.last_name);
DBMS_OUTPUT.PUT_LINE (r_emp_jobs.job_id);
END;
```

COLLECTIONS (three types)

*-**Associate array (INDEX BY table):** An associative array (formerly called PL/SQL table or index-by table) is a set of key-value pairs. Each key is a unique index, used to locate the associated value with the syntax variable_name(index). The data type of index can be either a string type or PLS_INTEGER. Indexes are stored in sort order, not creation order. For string types, sort order is determined by the initialization parameters NLS_SORT and NLS_COMP. Example:*

```
DECLARE
TYPE t_region IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
t_reg_tab t_region;
BEGIN
 t_reg_tab(1) := 'Southwest';
 t_reg_tab(2) := 'Northwest';
 t_reg_tab(3) := 'Southeast';
 t_reg_tab(4) := 'Northeast';
FOR v_Lp IN 1..4 LOOP
  DBMS_OUTPUT.PUT_LINE('t_reg_tab(' || v_Lp || ') is: ' ||t_reg_tab(v_Lp));
END LOOP;
```

*-**Nested Tables:** A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and the database. Within PL/SQL, nested tables are like one dimensional arrays whose size can increase dynamically.*

```
SET SERVEROUTPUT ON;
DECLARE
TYPE location_type IS TABLE OF locations.city%TYPE;
offices location_type;
table_count NUMBER;
BEGIN
  offices := location_type('Bombay', 'Tokyo','Singapore', 'Oxford');
  FOR i in 1.. offices.count() LOOP
    DBMS_OUTPUT.PUT_LINE(offices(i));
  END LOOP;
END;
/
```

***VARRAY types**: Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, thereby allowing varrays to be stored in database tables.*

In summary: A variable-size array (VARRAY) is similar to an associative array, except that a VARRAY is constrained in size.

```
TYPE location_type IS VARRAY(3) OF locations.city%TYPE;
offices location_type;
```

```
A.      when you want to retrieve an entire row from a table and perform calculations

B.      when you know the number of elements in advance and the elements are usually accessed
sequentially

C.      when you want to create a separate lookup table with multiple entries for each row of the main
table, and access it through join queries

D.      When you want to create a relatively small lookup table, where the collection can be  constructed
on memory each time a subprogram is invoked.

Answer A for :records
Answer B for :varrays
Answer C for :nested tables
Answer D for :associative arrays
```

- **Implicit cursor** -- Implicit cursors, also known as SQL cursors, are constructed and managed by PL/SQL. PL/SQL
- **Explicit cursor** -- Explicit cursors are constructed and managed by usercode.Explicit cursors are declared and defined in a block. They will have a name and be associated with a query.

Implicit Cursor	**Explicit Cursor**
Implicit cursors, also known as SQL cursors, are constructed and managed by PL/SQL	Explicit cursors are constructed and managed by usercode.Explicit cursors are declared and defined in a block. They will have a name and be associated with a query.

`BEGIN`
`UPDATE hr.employees`
`SET salary = salary * 1.1`
`WHERE job_id = 'IT_PROG';`
`IF SQL%ISOPEN THEN`
`DBMS_OUTPUT.PUT_LINE('Implicit cursor is open');`
`END IF;`
`IF SQL%FOUND`
	`DECLARE`
	`CURSOR c_itp IS`
	`SELECT first_name, last_name`
	`FROM hr.employees`
	`WHERE job_id = 'IT_PROG'`
	`AND salary = 4800;`
	`BEGIN`
	`IF c_itp%ISOPEN THEN`
	`DBMS_OUTPUT.PUT_LINE('Explicit cursor is open before loop');`
	`END IF;`

%ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT	

Example of explicit cursor

```
DECLARE
CURSOR c_emps IS
SELECT first_name, last_name, salary
FROM hr.employees
WHERE job_id = 'IT_PROG’ ORDER BY last_name;
 v_firstname hr.employees.first_name%TYPE;
 v_lastname hr.employees.last_name%TYPE;
 v_salary hr.employees.salary%TYPE;
BEGIN
 OPEN c_emps;
LOOP
 FETCH c_emps
 INTO v_firstname, v_lastname, v_salary;
 EXIT WHEN c_emps%NOTFOUND;
 DBMS_OUTPUT.PUT_LINE( v_lastname || ', ' || v_firstname || ' makes ' || v_salary );
END LOOP;
CLOSE c_emps;
END;
```

Cursors with parameters

```
DECLARE
CURSOR c_emps (p_job VARCHAR2, p_years NUMBER) IS
SELECT *
FROM hr.employees
```

```
WHERE job_id = p_job
AND MONTHS_BETWEEN(SYSDATE, hire_date) > (p_years * 12);
v_emps hr.employees%ROWTYPE;
BEGIN
OPEN c_emps('IT_PROG', 6);
LOOP
FETCH c_emps
INTO v_emps;
EXIT WHEN c_emps%NOTFOUND;
DBMS_OUTPUT.PUT_LINE( v_emps.first_name || ' ' ||
v_emps.last_name ||
' was hired on ' ||
TO_CHAR(v_emps.hire_date, 'DD-MON-YY'));
END LOOP;
CLOSE c_emps;
```

-The standard locking behavior is designed to provide the maximum concurrency.

- If there is a requirement to have exclusive access to data during a transaction,

- **LOCK TABLE** -- Explicitly locks entire tables. (not in the exam)
- **SELECT with the FOR UPDATE clause** -- Explicitly locks specific rows of a table. (The NOWAIT, WAIT, or SKIP LOCKED clause of the SELECT FOR UPDATE statement can be used to alter this behavior). : DECLARE

```
v_emp_id hr.employees.employee_id%TYPE;
v_job_id hr.employees.job_id%TYPE;
v_salary hr.employees.salary%TYPE;
CURSOR c_empsal IS
  SELECT employee_id, job_id, salary FROM hr.employees FOR UPDATE;
 BEGIN
 OPEN c_empsal;
  LOOP
   FETCH c_empsal INTO v_emp_id, v_job_id, v_salary;
   IF v_job_id = 'IT_PROG' THEN
  UPDATE hr.employees SET salary = salary * 1.05 WHERE employee_id = v_emp_id;
  END IF;
EXIT WHEN c_empsal%NOTFOUND;
END LOOP;
CLOS
```

```
DECLARE
v_job_id hr.employees.job_id%TYPE;
v_salary hr.employees.salary%TYPE;
CURSOR c_empsal IS
  SELECT job_id, salary FROM hr.employees FOR UPDATE;
BEGIN
 OPEN c_empsal;
  LOOP
   FETCH c_empsal
   INTO v_job_id, v_salary;
   IF v_job_id = 'IT_PROG' THEN
   UPDATE hr.employees SET salary = salary * 1.05
WHERE CURRENT OF c_empsal; -- only FOR UPDATE cursors
   END IF;
   EXIT WHEN c_empsal%NOTFOUND;
  END LOOP;
 CLOSE c_empsal;
END;
```

---

1) **Pre-Defined exceptions**: Because they have names, it is possible to create exception handlers for them.
2) **Non Pre-defined exceptions**: Oracle has hundreds of internally defined exceptions (ORA-xxxxx errors). Actually the pre-defined ones are a small fraction of the internally defined. We can declare a name for them so that you can write an exception handler specifically for that exception condition.

```
DECLARE
x_snapshot_too_old EXCEPTION;
PRAGMA EXCEPTION_INIT(x_snapshot_too_old, -1555);
BEGIN
...
EXCEPTION
WHEN x_snapshot_too_old THEN
```

```
...
END;
```

3) User-defined exceptions:

```
CREATE OR REPLACE PROCEDURE submit_timesheet (p_ts_date DATE)
IS
x_weekend_date EXCEPTION;
BEGIN
IF TO_CHAR(p_ts_date, 'DY') IN ('SAT', 'SUN') THEN
RAISE x_weekend_date;
END IF;
EXCEPTION
WHEN x_weekend_date THEN
DBMS_OUTPUT.PUT_LINE ('Cannot submit timesheet for weekend dates.');
END;
BEGIN
```

RAISE_APPLICATION_ERROR-> PARA ASIGNAR CODE Y MESSAGE, se tratara como si fuera non_predefined

```
PRAGMA EXCEPTION_INIT (exception_name, error_code)
```

RAISE_APPLICATION_ERROR is invoked using the following syntax:

```
RAISE_APPLICATION_ERROR (error_code, message[, {TRUE | FALSE}]);
```

The error_code must be an integer in the range -20000..-20999. The message
returned must be a character string of no more than 2048 bytes. When TRUE is
specified, PL/SQL puts error_code on top of the error stack.

IN -- This is the default parameter mode and need not be explicitly specified. It is used to pass a value to the subprogram. Formal IN parameters act like constants – they are read-only to the subprogram.
OUT -- OUT variables must be specified in the declaration of the subprogram and are used to return a value to the invoker.
IN OUT -- variables must be specified in the declaration of the subprogram and are used to pass an initial value to the subprogram and return an updated value to the invoker

Only a function heading can include the following:
□ **DETERMINISTIC** option -- Helps the optimizer avoid redundant function invocations.
□ **PARALLEL_ENABLE** option -- Enables the function for parallel
execution, making it safe for use in slave sessions of parallel DML evaluations.
□ **PIPELINED** option -- Makes a table function pipelined, for use as a row source.
□ **RESULT_CACHE** option -- Stores function results in the PL/SQL function result cache (appears only in declaration).
□ **RESULT_CACHE** clause -- Stores function results in the PL/SQLfunction result cache (appears only in definition).

```
DROP FUNCTION age_in_dog_years;
```

When a PL/SQL block contains two nested subprograms that call each other, then one of the two requires a forward declaration. It is not possible to invoke a procedure before it has been declared. If subprogram A calls subprogram B and vice versa, neither can be placed first. A forward declaration declares the subprogram, but does not define it.

```
DECLARE
v_verse NUMBER := 1;
PROCEDURE tweedledum(p_param1 NUMBER);
PROCEDURE tweedledee(p_param2 NUMBER)
IS
BEGIN
CASE v_verse
WHEN 2 THEN
DBMS_OUTPUT.PUT_LINE('Agreed to have a battle;');
v_verse := 3;
WHEN 4 THEN
DBMS_OUTPUT.PUT_LINE('Had spoiled his nice new rattle.');
v_verse := -99;
END CASE;
IF v_verse != -99 THEN
tweedledum(v_verse);
END IF;
END;
```

As mentioned in the previous section -- the first time that a package is called by a session, it is instantiated for that session. Part of that instantiation is setting the package state. Any variables, constants, and cursors that a package declares in either the specification or the body make up the package state.

EXECUTE IMMEDIATE has these clauses:

 INTO -- Used to designate variables to hold the results of a SELECT statement that returns a single row

 BULK COLLECT INTO -- Used to designate variables to hold the results of a SELECT statement that returns multiple rows.

 USING -- Allows you to designate incoming or outgoing bind arguments to the dynamic SQL.

```
EXECUTE IMMEDIATE sql_statement
[INTO {variable
[,variable] ... record}]
[USING [IN|OUT|IN OUT] bind_argument
[, [IN|OUT|IN OUT] bind_argument] ... ];
```

The OPEN FOR, FETCH, and CLOSE statements can be used in lieu of EXECUTE IMMEDIATE.

```
DECLARE
TYPE t_emp_tab IS TABLE OF hr.employees%ROWTYPE;
t_tab t_emp_tab;
c_emps SYS_REFCURSOR;
BEGIN
  OPEN c_emps FOR 'SELECT * FROM hr.employees';
  FETCH c_emps
  BULK COLLECT INTO t_tab;
CLOSE c_emps;
```

In particular, there is no support in NDS for dynamic SQL statements that have an unknown number of inputs or outputs. In addition, there are some tasks that can only be performed using DBMS_SQL. The DBMS_SQL package is owned by the SYS schema and compiled with AUTHID CURRENT_USER. This means subprograms in the package that are

---

Subprograms:

A local subprogram (also called a nested subprogram) is created within the declaration section of a PL/SQL block. It is accessible only to the block in which it is declared. Local subprograms can be used when there is a given set of code that must be performed multiple times within a block, but has no utility outside that block.

```
CREATE PROCEDURE emp_years(p_emp_id NUMBER)
AS
v_row hr.employees%ROWTYPE;
FUNCTION yrs_employed(p_hiredate DATE)
RETURN NUMBER
AS
BEGIN
RETURN TRUNC((SYSDATE - p_hiredate)/365);
END yrs_employed;
BEGIN
SELECT *
INTO v_row
FROM hr.employees
WHERE employee_id = p_emp_id;
DBMS_OUTPUT.PUT_LINE(v_row.first_name || ' ' ||
v_row.last_name || ': ' ||
yrs_employed(v_row.hire_date) ||
' years.');
END emp_years;
BEGIN
emp_years(106);
END;
```

Subprograms execute with the privileges of their owner, by default. This allows indirect access to database objects and more granular data security. Users only need to be granted the privilege to execute the procedure and not on the objects accessed by the subprogram

The two options for authorization are:

1) CURRENT_USER
2) DEFINER

```
CREATE OR REPLACE PROCEDURE create_dept (
   v_deptno NUMBER,
   v_dname  VARCHAR2,
   v_mgr    NUMBER,
   v_loc    NUMBER)
AUTHID CURRENT_USER AS
BEGIN
   INSERT INTO departments VALUES (v_deptno, v_dname, v_mgr, v_loc);
END;
/
CALL create_dept(44, 'Information Technology', 200, 1700);
```

```
CREATE PROCEDURE eoy_bonus (p_emp_id NUMBER, p_amount NUMBER)
AS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  UPDATE hr.employees
  SET salary = salary + p_amount WHERE employee_id = p_emp_id;
  COMMIT;
END eoy_bonus;
```

NOCOPY **asks** the compiler to pass the corresponding actual parameter by reference instead of by value

```
   PROCEDURE parametros_x_referencia (
      p_productos   IN OUT NOCOPY   productos_ct
   );
```

PARALLEL_ENABLE / DETERMINISTIC

```
DECLARE
TYPE t_firstTyp IS TABLE OF hr.employees.first_name%TYPE;
TYPE t_lastTyp IS TABLE OF hr.employees.last_name%TYPE;
TYPE t_salTyp IS TABLE OF hr.employees.salary%TYPE;
t_first t_firstTyp;
t_last t_lastTyp;
t_sal t_salTyp;
BEGIN
UPDATE hr.employees
SET salary = salary * .95
WHERE job_id = 'PU_CLERK'
RETURNING first_name, last_name, salary
BULK COLLECT INTO t_first, t_last, t_sal;
DBMS_OUTPUT.PUT_LINE ('Updated ' || SQL%ROWCOUNT || ' rows:');
FOR v_Lp IN t_first.FIRST .. t_first.LAST LOOP
DBMS_OUTPUT.PUT_LINE (t_first(v_Lp) || ' ' ||
t_last(v_Lp) || ' salary ' ||
t_sal(v_Lp));
END LOOP;
END;
```
