

This is a class in using a database system to work with data. So we need to have some agreement on a few terms. For this first unit of class, we really do not need a deep understanding of these concepts. We will develop these over the next few units. But I want you to connect to your database system and get started, so some terminology is required. Remember these are the quick-and-dirty explanations.

Data – values that have meaning to us for some purpose. For example, your name, your student ID, your date of birth, the number of units for this class- these are all data. Some data values are numeric, other data values are dates, and other data values are text. To be useful, we need to organize the data and store it in a way that we can retrieve it efficiently. If we have a lot of data, we will generally store it using a computer system and one of the more useful ways to organize and store data is to use a database.

Database – a collection of data that we can use for more than one purpose. For example, CCSF has a student database that we use for registration, production of transcripts, billing students, and sending reports to the state for funding. Databases are everywhere these days.

Table – the data in the database is stored in structures that we can think of as tables.

Student ID	LastName	FirstName
W0000003	Babbage	Charles
W0000001	Hopper	Grace
W0000002	Teape	Betsy

A table consists of a set of rows, this table has three rows of data. The shaded row at the top contains the table header. A table stores data of a specific type defined by its columns. In the student database, we might have a table for basic student data; each row would contain the data values for a specific student; the columns might have names such as StudentLastName, StudentFirstName, StudentResidencyState, StudentID, and StudentDOB. Each column stores one type of information that we need to keep for a student. We might have another table, ClassesTaken, for the classes a student has taken; in that table the columns might have names such as StudentID, CourseID, Semester, and FinalGrade.

Identifier - If we are storing data about students in a table, we would want to be able to identify a specific student and his data. Most tables will have a column that is used to identify each row of data. With the student table described in the previous paragraph, we would probably use the StudentID as the identifier. What do you think we could use as an identifier for the ClassesTaken table? Consider that a student might take a class more than once.

Query – a question that we ask about the data- such as "for which classes is the student with ID W00000009 registered?" or "show me how many students we have registered for classes this semester who do not live in California".

Schema – this is the collection of tables that you have for a particular purpose. In a company we might have a schema for our customer data and another schema for the products that we sell and another schema for our employees. In some database systems, the terms schema and database mean the same thing; in other database systems, the schema is a subcategory of a database.

If we want to store our data in tables in a database, we will need to have computer programs that help us store and retrieve the data. The main programs are the RDBMS, clients, and application programs.

RDBMS –Relational Database Management System – this is a collection of programs that handle the physical files that make up the database. When we create a table, add rows of data to that table, or look at the data in that table, we issue a command that is transferred to one or more of the programs in the RDBMS which does the actual reading or writing of the data. The RDBMS also handles security and other issues.

Client/Server – a client/server system has two types of software- the server software and the client software. The server software is also called the database engine. These two types of software can be installed on the same physical computer or on different computers that are linked via a network.

Database server/engine – this is the set of programs in the RDBMS that work directly with the data. When you run a query, you are actually passing the query to the database server. In many systems the terms server and engine mean the same thing. In this class, we are using an Oracle 12g database server.

Client – to communicate with the database engine, you need a program to enter your SQL statements and see the results; that is a client program. We are using the SQL*Plus client to issue queries to the dbms and receive the results; we can use another client - SQL Developer -which has a graphical interface. You can use several different clients to work with the same DBMS

Application program- this would be the programs that a typical end user would work with. A typical end-user does not interact directly with the SQL language. These application programs would often be written in languages such as Java, Visual Basic or C#. We do not work at this level in this class.

SQL – a language which is commonly used to communicate with a database. Much of this semester is devoted to learning how to use SQL effectively.

Set-based operations and closure – These are basic concepts of the relational system. When we work with a database using the SQL language our commands are applied to the entire table as a unit; we can think of the table as a set of rows and we think of the query as working on all of the rows at the same time. The result of a command is another table; this is the feature of closure.

The first question listed above was "What classes is the student with ID W00000009 registered for?" The result of that query might be a list of classes showing the ID and the CRN of each class and its meeting days. That looks like a table.

Student ID	Last Name	Class CRN	Class ID	Days
W00000009	Jones	78956	CS 151A	Online
W00000009	Jones	78512	CS 159A	Tues
W00000009	Jones	70236	CS 114D	Thurs

The result of the second query "Show me how many students we have registered for classes this semester who do not live in California" is also a table- even if it has only one row and one column.

OutOfStateStudents
845

With some queries, the result set might be empty. This is still considered a table- it might be called an empty table. Depending on the client, you might not see a column header row such as indicated here.

StudentsUnder12YearsOfAgeEnrolledInCSClasses
no rows selected

Instead you might get a response such as

no rows selected

Table of Contents

1.	First Time Connecting to the CCSF Linux system	1
1.1.	Dunes accounts	1
1.2.	First time login to dunes	2
1.3.	Set up for Oracle	2
1.4.	Directory for storing your scripts	3
2.	Routine Connecting to the CCSF Linux system and oracle	3
2.1.	Routine login to Oracle	4
2.2.	Changing your Oracle password	4
3.	Entering an SQL command.....	5
3.1.	Copy and paste approach for testing single queries	5
4.	File names, Extensions, Paths	5

Most students taking this class will use the CCSF Oracle system to do their class work; that is generally the easiest way. I have included a document on other software choices in this unit.

If you are using the CCSF system, the "setup" is pretty minimal. Our student Oracle system is on the dunes server. When you enroll in this class you will get a dunes account that lets you log into the server. You will also have your account set up so that you have permission to use the Oracle dbms. You need to do a few things to get started. These are things you need to do only once.

- 1) Be certain that you have the software needed to connect to the dunes server - we recommend SSH Secure Shell for windows users. There is a separate document in this unit on getting SSH Secure Shell. You can use other communication programs. You will be doing your work on the dunes/Oracle system; it makes no difference how you connect to dunes.
- 2) The first time you log into dunes, you will have to change your default password. The dunes system was changed in May 2103 and people with previous accounts will still need to handle the initial password change. If you had a dunes account earlier (such as last semester), your password will not be reset.
- 3) You will have to modify a file on your account to access Oracle
- 4) You should create a directory on your dunes account for your files for this class.

This document also discusses a routine log-in that you will use most of the time.

1. First Time Connecting to the CCSF Linux system

1.1. Dunes accounts

In this class, every student has a dunes account that can access Oracle. Dunes is the student server running linux. You can connect to the dunes linux system from the ACRC, from other places on campus, or from your home computer using a communication program such as SSH Secure Shell to connect (see info on SSH Secure Shell in another document for this unit.) Dunes is normally available 24 /7.

When you connect to the linux system, you need a dunes user name and a dunes password; to then log into Oracle you need an Oracle user name and an Oracle password. Your dunes **user name** and your Oracle **user name** are the same. The passwords can be different.

The linux operating system is case-sensitive. This applies to login ids, passwords, and commands. Most entries are made in lower-case letters.

dunes login: We are now using your Gmail userid as the id to your dunes account. You can look up that userid (and your initial Gmail password) inWeb4 /WebStar under "Personal Information."

dunes initial password: For dunes accounts, the initial password pattern is mmmddyy.xx. The first part mmmddyy is based on your birthdate formatted as the three letter month abbreviation , in lower case, followed

by two digits for the day and two digits for the year. Then there is a period, followed by the first two characters of the login. For example, if Martha Graham (with user name mgraham) told CCSF that she was born on July 5, 1935, her initial password would be `jul0535.mg`

This password is used only for your first login and you will be forced to select a new password at your first linux login.

(This format for student passwords is being used for dunes accounts, Windows accounts in the ACRC, Mac server accounts, and Linux accounts.)

Your password is a string of characters that you will select to guard access to your account. A password must be at least six characters long and contains at least two letters and one digit. It is case-specific. You may use the period (.), the hyphen (-), and the underscore (_) in your password. Avoid other special characters since they might have a special meaning to linux.

If you forget your password, you will not be able to logon to the dunes system. Your password is private information and you can change it at any time.

If you forget your CCSF dunes password (or your Oracle password) or have any trouble running SQL*Plus you should call the Help Desk; it should be available 24/7. the phone numbers are 415.239.3711 and Toll Free 844.693.HELP. The email is helpdesk@ccsf.edu. Please note: the help desk does not provide help with the class material. The help desk staff can also reset dunes and Oracle passwords.

I cannot reset a CCSF password.

1.2. First time login to dunes

The first time you log in to dunes, there will be some additional tasks

- Log into dunes. The dunes server is at dunes.ccsf.edu
- Change your default password; the password is not displayed; the system will automatically log you out after this
- Log back into dunes using the new password

The command to log out of linux is **exit**.

(The ACRC has a handout that is used in their Orientations for logging into dunes. It is very detailed and included some steps that are specific to using the ACRC. It might be helpful for students who have never done this before. http://www.ccsf.edu/Services/ACRC/handouts/dunes-1_Login_out.pdf. You could also attend the ACRC orientations.)

1.3. Set up for Oracle

This is done **once only**.

- Download the following 2 files from Canvas – you can click on the links to access them directly
 - [ora_profile](#)
 - [set oraenv](#)
- Upload these 2 files to your root folder on dunes. Your root folder is your default login folder.

You need to edit your profile file so that you can access Oracle.

- Log into dunes
- edit your `.bash_profile` file

To edit your `.bash_profile` file: You need to know how to use a linux editor to do this. nano is the easiest editor to use.

- Log into dunes
- At the dunes level, use an editor to edit the file named `.bash_profile` --- the file name starts with a dot.
- You can use the nano editor by typing
`nano .bash_profile`
- Arrow down to the bottom of the file and append the following line. This line starts with a dot and a space.
`. ora_profile`
- After you enter that line, enter `Ctrl_X` to exit nano
- Enter “y” when prompted to “Save modified buffer...”
- Press Enter to accept the existing file name when prompted for “File Name to Write.”
- You then need to execute that file so that these changes are made. You can do that with the dot command (this has a dot, then a space and then the filename `.bash_profile`)
`. .bash_profile`
- or by logging off and logging in again.
- You can find detailed instructions for the nano editor at [The nano Editor](#)

1.4. Directory for storing your scripts

Create a directory to store your script files. When you create script and spool files on the linux system, those files will be stored in your current directory. It is a good idea to keep all of these files in a separate directory—not at your root directory.

- Log into dunes
- Make a directory for your files; you do this **only once**
`mkdir 151A`
- Change to your directory for your class stuff ; you do this each time you login **before you login into the Oracle system**
`cd 151A`

2. Routine Connecting to the CCSF Linux system and oracle

Once you have set up your account for Oracle access, the steps to get to Oracle will be pretty simple:

- log into dunes
- change to your directory for your files (`cd 151A`)
- log into Oracle
- do your Oracle work
- log out of Oracle (exit)
- log out of dunes (exit)
- You can find instructions on using Linux at [Hills Linux](#) (**this does not mean we are using hills. The documentation for hills is the same as for dunes**).

2.1. Routine login to Oracle

The command to log into Oracle is sqlplus. You will be prompted for your user name and then for your password. Your **initial** Oracle password follows the birth date pattern with the .xx pattern. (The staff at the ACRC and the Help Desk can change your Oracle password if you forget it.)

- Log into dunes
- Change to your directory for your class stuff
- `cd 151A`
- Log into Oracle
- `sqlplus <User ID>@ORCL (ex. sqlplus dsgoldma@ORCL)`
- Supply your login when prompted. It is the same password as your dunes Linux password (see above).
- Supply your password when prompted
- Do something in Oracle so that you know it works. The following statement displays the current date and is enough to let you know that you have successfully gotten into Oracle.
- `select sysdate from dual;`
- Log out of Oracle
- `exit`
- Log out of dunes
- `exit`

Note: it is possible to give the sqlplus command and your login on the same command line; this is OK. It is also possible to include your Oracle password on this command line—this is not a good idea. Anyone using the linux system can give a command to see everyone's command lines. If you put your password on the command line, it is exposed to all users.

When you have logged into SQL*Plus, you are said to have started an Oracle session. The session ends when you log out of SQL*Plus.

To quit SQL*Plus, use the command exit

It is important to log out of Oracle and then log out of linux. If you simply close your communication program, your data might not be saved and your tables might be hung up for a while. Do the exit from Oracle and the exit from linux- it will save you time.

2.2. Changing your Oracle password

You may wish to change your SQL password. The method to accomplish this has changed in the new Dunes Oracle implementation. The new steps are as follows:

- Login to Oracle with SQL*Plus and issues the following commands:
`SQL> alter session set "_ORACLE_SCRIPT"=true;`
`SQL> alter user <User ID> identified by <New Password>`
- If you have special characters in your password you must enclose the entire password in double quotes.
For example,
`SQL> alter user dsgoldma identified by "myp@ssK!ds"`

3. Entering a SQL command

The prompt for SQL*Plus is SQL>. You type in an sql statement at the prompt, hit the enter key and the system responds. For your first command enter the following. Be certain to include the ending semicolon.

```
select sysdate from dual;
```

3.1. Copy and paste approach for testing single queries

Another method to work with the SQL*Plus environment in creating SQL statements in a windows environment is to take advantage of the copy and paste features of windows. Open a second window with Notepad (or Notepad ++) or other text editor. Create and edit your SQL statement in the text document. Then copy it and paste it into your Oracle SQL*Plus session window using copy and paste techniques.

It is likely that your Oracle SQL*Plus session window will appear garbled but it should run. You can give the list command to review the SQL buffer. If this is a Select command, you can use the / command to run the statement again.

This is helpful for testing because you can use the editing capabilities of the text editor when creating the SQL statements and can save that document separate from the Oracle system. You should be able to copy the demo sql from the supplied notes to the client to try them out.

You can also copy the output from the SQL*Plus window to your text document.

You can use this approach to building up the script file for assignments, but you need to use the spool command to create the spooled output for grading.

4. File names, Extensions, Paths

I expect people taking this class to have sufficient experience with their computer system to handle the following. This refers to working with files on your **local computer**- not files on the dunes system. I expect that all files you have on the dunes system are in your 151A directory if this is the only class you are taking that uses the dunes system. The linux system displays the full pathname.

The files we will use have a name which consists of a base name and an extension. For example, for Assignment A01, a student with the last name of Jones will have a file named A01_Jones.sql and another file named A01_Jones.lst. These files differ only in the file name extension (sql and lst are file name extensions.)

Some systems try to hide the file name extensions. Some systems say you do not need file extensions. But for the class assignments you do need to have file names such as A01_Jones.lst and A01_Jones.sql and you need to be able to tell these files apart. So you may want to find out how to do this.

You can do an internet search for something like : Mac file extensions or Windows extensions to get directions for your system.

This is a page I found that shows techniques for windows XP, 7, and 8 (and it has pictures).

<http://www.bleepingcomputer.com/tutorials/how-to-show-file-extensions-in-windows/>

You will probably keep a copy of your assignment files on your local computer ; You will also upload a copy of your script file to dunes in order to run it.

File path- this is a string that says where to find a file on your file system. The full path starts at the root of the disk- or other storage device- and lists the various folders you need to go through to get to the file. You use the

backslash(\) to separate the components of a path. C: represents the C hard drive. (this is the windows version Mac people do a search for Mac file path).

For example

C:\db_scripts\A01_Jones.sql

The next path which is a single string may be to be quoted because it contains spaces.

C:\Documents and Settings\Rose Endres\My
Documents\CS\2016_spring\151A\5_scripts_151A\151A_vets\vetsCreates_151A.sql

There is a place in a command line where you have to write the pathname to a file- which of these would you prefer to write? Keep your script files in a folder with a short pathname.

Table of Contents

1.	The login screen from Linux	1
2.	Exiting SQL*Plus	1
3.	Entering an SQL statement	1
4.	Command line interpretation	2
5.	Some useful SQL*Plus variables.....	2
6.	SQL*Plus commands	4
6.1.	Desc	4
6.2.	Column	4
6.3.	Clear columns.....	5

SQL*Plus is the traditional client used to work with an Oracle database that provides access to your data and lets you create and run SQL statements. You need to be familiar with using SQL*Plus even if you often use more graphical tools.

1. The login screen from Linux

Change to the directory you are using to store your scripts.

The login command is sqlplus; you can include your user name on the command line.

```
The Oracle base remains unchanged with value /oracle/app  
ORACLE_HOME is /oracle/app/product/12.1.0  
ORACLE_SID is cs12  
$ sqlplus rendres  
  
SQL*Plus: Release 12.1.0.1.0 Production on Sat Jun 21 17:42:47 2014  
  
Copyright (c) 1982, 2013, Oracle. All rights reserved.  
  
Enter password:  
Last Successful login time: Wed Apr 30 2014 15:56:55 -07:00  
  
Connected to:  
Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production  
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing opt  
ions  
  
SQL> ■
```

2. Exiting SQL*Plus

The session ends when you log out of SQL*Plus. To quit SQL*Plus, use the command **exit**

3. Entering an SQL statement

One way to enter an SQL query is by typing the SQL statement followed by a semicolon. The **SQL>** in the line below is the system prompt; you do not type that.

```
SQL> select sysdate from dual;
```

```
SYSDATE  
-----  
10-JAN-16  
  
SQL>
```

Notice that the output column headers are displayed in uppercase letters. The column headers may be truncated to fit in the column width defined in the table.

To enter an SQL statement from the SQL*Plus environment, simply type it at the SQL> prompt. The SQL statement can extend over several lines and does not use a continuation character. You can terminate the SQL statement in three ways:

- End the last line with a semicolon. The SQL statement will be executed. The semicolon can be on a line all by itself. For the examples, I have generally ended the SQL statement with a semicolon.
- Enter a slash on a line all by itself. The SQL statement will be executed.
- Enter a blank line. The SQL statement will not be executed, but will stay in the SQL buffer. This is helpful if you notice that you made a mistake and want to re-enter the command.

To execute an SQL statement that is in a buffer, you can

- use a forward slash on a new line
- the run command (this also displays the buffer)

4. Command line interpretation

This section will help you understand some error situations - don't worry about it too much right now.

There are three types of commands you can enter at the SQL*Plus prompt:

- SQL*Plus commands
- SQL commands/statements
- PL/SQL blocks

SQL*Plus determines which type of command you are entering by examining the start of the command line. It sends SQL statements to the SQL engine and PL/SQL blocks to the PL/SQL engine. If the command line is not SQL or PL/SQL, then it is executed by SQL*Plus.

For example:

```
set echo on;      This is an SQL*Plus command
select sysdate from dual;    This is an sql command
```

We will see some PL/SQL later.

Some command lines include both SQL*Plus components and SQL components—in that case, SQL*Plus does its work before sending the SQL or PL/SQL to the database engines.

The results of an SQL statement is returned to SQL*Plus for display. Environment settings in SQL*Plus will influence the final display of the result.

SQL*Plus maintains a buffer called the SQL buffer. It contains the most recent SQL statement that you entered (or the most recent PL/SQL block). Executing the SQL statement does not empty the SQL buffer. SQL*Plus commands do not get stored in the SQL buffer.

You can clear the SQL buffer with the SQL*Plus command CLEAR BUFFER

You can clear the screen buffer with the SQL*Plus command CLEAR SCREEN

This will make more sense as the semester goes on- for now just understand that in the SQL*Plus client you can enter some commands the client executes and some that the database server executes.

5. Some useful SQL*Plus variables

Your SQL & PL/SQL statements work within the SQL*Plus environment and the output that you see is influenced by both the SQL statement and the SQL*Plus environment. You can modify how your SQL*Plus environment works by setting system variables. The values of the system variables remain until you terminate your session; therefore they continue to affect subsequent statements that you run in that session

You can give the SQL*Plus command SHOW ALL to see the values for all of your system variables. You can give the SQL*Plus command SHOW varbName to see the current setting for a particular variable.

We are going to start with only a few SQL*Plus variables that affect the spool files you will produce for assignments.

The usage of these commands is: SET FEEDBACK 15

The Set command, followed by the variable name and any option

System Variable	Values	Description
ECHO	ON OFF	Echo causes the commands in your script files to be displayed on the screen (and put in the spool file). We will use this for assignments.
FEEDBACK	n ON OFF	Feedback produces a line after the SQL statement runs that states how many rows were displayed by the query. The default value is 6; if your query produces fewer than 6 output rows, then the feedback line is not displayed. For assignments, set feedback to 1
PAGESIZE	n	Pagesize sets the number of lines per page. This determines how many lines will be displayed on the screen before the column headers are repeated. This is a physical line count, not a row count. So it includes column headers and blank lines as well as lines that display data. The default value is 24. For assignments, set pagesize to 999
TRIMSPPOOL	ON OFF	Trimspool handles some of the problems with excess spaces at the end of the output lines. For assignments this should be set ON.
LINESIZE	n	The Linesize setting controls the number of characters SQL*Plus displays on a physical line. This can help prevent linewraps with long display lines.
TAB	ON OFF	SQL*Plus can keep spaces in the output or replace them with tab characters. For assignments this should be set OFF.

For our script assignments I want the following SQL*Plus statements included at the top of each script. For the page size, I do mean the literal 999. These commands are in the template you use for assignments.

```
set echo on
set feedback 1
set pagesize 999
set trimspool on
set linesize 200
set tab off
clear columns
```

6. SQL*Plus commands

SQL*Plus was designed with a series of commands that could be used for simple report formatting. Most report formatting should be done in another tool and is not covered in this class. Therefore we need very few SQL*Plus commands.

We have seen a few SQL*Plus command so far: exit, set, clear.

- SQL*Plus commands are not case specific.
- Most SQL*Plus command are short, but if you need to extend an SQL*Plus command over two lines use the hyphen (-) as the continuation character.
- Many SQL*Plus commands keywords have an abbreviated version.

6.1. Desc

Suppose you have a table you created a few weeks ago and you want to see the names of the columns and their data types. You can use the SQL*Plus command Describe (desc). We will create this table in this unit- see document First Table.

SQL> desc zoo_2016;

Name	Null?	Type
Z_ID	NOT NULL	NUMBER(38)
Z_NAME		VARCHAR2(25)
Z_TYPE	NOT NULL	VARCHAR2(25)
Z_COST	NOT NULL	NUMBER(7,2)
Z_DOB	NOT NULL	DATE
Z_ACQUIRED	NOT NULL	DATE

SQL> _

This gives you a quick view of the table structure- showing the names of the columns, if they are nullable or not (more on this later in the semester) and their data type. If you compare this to the Create table statement, you will see a few differences:

- Some data type names are different- I use the type Integer and internally Oracle Database changed that to Number(38)
- The constraint I set up for the z_cost value is not shown here.

6.2. Column

The column command in SQL*Plus has a lot of options. For assignment scripts, you may use only three of these options. In the zoo_2016 table we have a column for the animal name that was set up to store 25 characters. And you have already seen that our rows are rather wide. I decide that I want the animal names to be in a column of width 10. I can set this up as an SQL*Plus column command as shown here

```
column z_name format a10
column z_name format a10 word_wrapped
column z_name format a10 trunc
```

I provide the column name and a format- here the format is A10 which says this is a width 10 alphanumeric(string) column. Note that this does not use the table name.

But what happens if I have an animal named 'Sally Robinson'- that is longer than 10 characters. I can use a second option word_wrapped to say that I want the data wrapped with line breaks between words. And there is a third option Trunc

The following shows a Select command with three different column options. For class you do not need to worry about these options but they can be helpful to make the output more readable.

-- this is the default with 25 spaces for the name

```
SQL> select z_id, z_name, z_type
  2  from zoo_2016
  3  where z_id in (52,85);

  Z_ID Z_NAME          Z_TYPE
----- -----
  85  Sally Robinson   Giraffe
  52  Dewey             Giraffe

2 rows selected.

SQL> column z_name format a10
SQL> /
      Z_ID Z_NAME          Z_TYPE
----- -----
  85  Sally Robi Giraffe
        nson

  52  Dewey             Giraffe

2 rows selected.

SQL> column z_name format a10 word_wrapped
SQL> /
      Z_ID Z_NAME          Z_TYPE
----- -----
  85  Sally      Giraffe
        Robinson

  52  Dewey             Giraffe

2 rows selected.

SQL> column z_name format a10 trunc
SQL> /
      Z_ID Z_NAME          Z_TYPE
----- -----
  85  Sally Robi Giraffe
  52  Dewey             Giraffe

2 rows selected.
```

6.3. Clear columns

This SQL*Plus command will clear the columns settings you have in your session. You are required to use this in your scripts (it is in the template) and that means that if you use any column commands, they need to be in the script after this command.

Table of Contents

1. Selecting columns.....	1
2. Selecting all columns	2
3. Column aliases.....	2
4. Sorting the output display.....	3

In this discussion we will examine a few features of the Select statement. These are:

- selecting individual columns
- selecting all columns
- using column aliases
- sorting the rows displayed

1. Selecting columns

The first few queries use only two clauses: the FROM clause to identify the table that supplies the data and the SELECT clause to identify the columns to be returned. For these queries, all rows from the table are returned. This set of demos uses the zoo_2016 table. Your data set might be different depending on the rows you inserted. You indicate which columns you want displayed and the order of the columns by listing the column names in the Select clause.

Demo 01: You can display the columns in any order. Note that row for the animal with no name displays a blank cell with this client.

```
Select z_type, z_name
From zoo_2016;
```

Z_TYPE	Z_NAME
Giraffe	Sam
Armadillo	Abigail
Lion	Leon
Lion	Lenora
Giraffe	Sally Robinson
Zebra	Huey
Zebra	Dewey
Zebra	Louie
Horse	
Giraffe	Dewey
Giraffe	Arnold
Giraffe	
Giraffe	
Giraffe	Geoff
armadillo	Anders
armadillo	Anne
Lion	Leon
Lion	
Lion	

Demo 02: Display dates and numeric values. The default display for dates uses a format with a two digit day, a three letter month and a 2 digit year.

```
Select
    z_dob
  , z_cost
  , z_name
From zoo_2016;
```

Z_DOB	Z_COST	Z_NAME
15-MAY-14	5000	Sam
15-JAN-13	490	Abigail
25-FEB-10	5000	Leon
25-MAR-14	5000	Lenora
15-MAY-14	5000.25	Sally Robinson
02-JUN-13	2500.25	Huey
02-JUN-14	2500.25	Dewey
02-JAN-13	2500.25	Louie
10-JAN-15	490	
06-JUN-13	3750	Dewey
15-MAY-14	5000	Arnold
15-MAY-13	5000	
15-MAY-02	5000	
15-MAY-02	5000	Geoff
15-JAN-13	490	Anders
15-JAN-13	490.01	Anne
25-FEB-13	1850	Leon
25-FEB-13	1850	
25-FEB-13	1850	
25-FEB-13	1850	

2. Selecting all columns

The symbol * is used to indicate that all columns should be returned. This is inefficient if you do not need to see all of the columns but is helpful for a quick look at a small table.

Using Select * can be a bad idea with embedded SQL if the table design is changed. Embedded SQL refers to SQL statement that might be included inside other units of code. You also have to consider that someone might reorder the column positions in the table and then your query produces a different result.

Demo 03: Display all columns, all rows.

```
Select *
From zoo_2016
;
```

Z_ID	Z_NAME	Z_TYPE	Z_COST	Z_DOB	Z_ACQUIRE
23	Sam	Giraffe	5000	15-MAY-14	15-MAY-14
25	Abigail	Armadillo	490	15-JAN-13	15-APR-13
56	Leon	Lion	5000	25-FEB-10	25-MAR-10
57	Lenora	Lion	5000	25-MAR-14	31-MAR-14
85	Sally Robinson	Giraffe	5000.25	15-MAY-14	15-MAY-14
43	Huey	Zebra	2500.25	02-JUN-13	02-JUN-14
44	Dewey	Zebra	2500.25	02-JUN-14	02-JUN-14
45	Louie	Zebra	2500.25	02-JAN-13	02-JAN-13
. . . rows omitted					

3. Column aliases

By default, the column headers are the attribute names displayed in uppercase letters. Column aliases can be used to supply different headers for the output display. Column aliases are limited to 30 characters.

Notice in the demos below that the column aliases are in upper case.

You can also use the SQL*Plus column command to affect the final display. We will not use this command during most of the semester as we are focusing on the SQL proper.

Demo 04: Display column headers other than the attribute names. The word AS is optional and may be omitted. Most people include the word AS for the column aliases.

```
Select
  z_id
, z_dob AS BirthDate
, z_cost AS Price
, z_name AS NAME
From zoo_2016
;
```

Z_ID	BIRTHDATE	PRICE	NAME
23	15-MAY-14	5000	Sam
25	15-JAN-13	490	Abigail
56	25-FEB-10	5000	Leon
57	25-MAR-14	5000	Lenora
85	15-MAY-14	5000.25	Sally Robinson
43	02-JUN-13	2500.25	Huey
44	02-JUN-14	2500.25	Dewey
45	02-JAN-13	2500.25	Louie
. . . rows omitted			

Demo 05: The use of double quotes for your aliases allows you to use spaces or special characters in the header and also preserves the case. Note that the default column width that SQL*Plus uses for a date column truncates our column alias.

```
Select
  z_id
, z_dob AS "Date of Birth"
, z_cost AS "Price $"
, z_name AS "Name"
From zoo_2016
;
```

Z_ID	Date of B	Price \$	Name
23	15-MAY-14	5000	Sam
25	15-JAN-13	490	Abigail
56	25-FEB-10	5000	Leon
57	25-MAR-14	5000	Lenora
85	15-MAY-14	5000.25	Sally Robinson
43	02-JUN-13	2500.25	Huey
44	02-JUN-14	2500.25	Dewey
45	02-JAN-13	2500.25	Louie
. . . rows omitted			

4. Sorting the output display

If you want to control the order in which the rows are displayed, you use an ORDER BY clause.

You can order by

- a column
- a column alias
- the numeric position of the column in the Select (not always a good idea)
- a calculated column expression (we will discuss this in the next unit)

If you have two columns with the same alias and try to sort by the alias, you will get an error message.

Demo 06: Controlling the order in which the rows are displayed. This is sorted by price with the lower values first; this is an ascending sort which is the default sort order.

```
Select
  z_id
, z_cost AS "Price"
, z_name As "Name"
From zoo_2016
ORDER BY z_cost;
```

Z_ID	BirthDate	Price	Name
25	15-JAN-13	490	Abigail
370	15-JAN-13	490	Anders
47	10-JAN-15	490	
371	15-JAN-13	490.01	Anne
374	25-FEB-13	1850	
375	25-FEB-13	1850	
373	25-FEB-13	1850	
372	25-FEB-13	1850	Leon
44	02-JUN-14	2500.25	Dewey
45	02-JAN-13	2500.25	Louie
43	02-JUN-13	2500.25	Huey
52	06-JUN-13	3750	Dewey
56	25-FEB-10	5000	Leon
23	15-MAY-14	5000	Sam
259	15-MAY-02	5000	
258	15-MAY-13	5000	
257	15-MAY-14	5000	Arnold
57	25-MAR-14	5000	Lenora
260	15-MAY-02	5000	Geoff
85	15-MAY-14	5000.25	Sally Robinson

Demo 07: Using DESC to specify a descending sort.

```
Select
  z_id
, z_cost AS "Price"
, z_name As "Name"
From zoo_2016
ORDER BY z_cost DESC;
```

Z_ID	BirthDate	Price	Name
85	15-MAY-14	5000.25	Sally Robinson
56	25-FEB-10	5000	Leon
57	25-MAR-14	5000	Lenora
259	15-MAY-02	5000	
258	15-MAY-13	5000	
23	15-MAY-14	5000	Sam
260	15-MAY-02	5000	Geoff
257	15-MAY-14	5000	Arnold
52	06-JUN-13	3750	Dewey
45	02-JAN-13	2500.25	Louie
44	02-JUN-14	2500.25	Dewey
43	02-JUN-13	2500.25	Huey
374	25-FEB-13	1850	
375	25-FEB-13	1850	
373	25-FEB-13	1850	
372	25-FEB-13	1850	Leon
371	15-JAN-13	490.01	Anne
370	15-JAN-13	490	Anders
25	15-JAN-13	490	Abigail
47	10-JAN-15	490	

If two rows have the same value for z_cost, then we have not specified an exact order for those rows

Demo 08: This is a two level sort. The first sort key is the z_type. If the z_type of two rows match, then the cost is used for the second sort level.

The other thing to note here is that the case of the z_type values is considered. Oracle is case sensitive on sorting character data and uppercase letters sort before lowercase. You can see that z_type value of 'Armadillo' sorts first and that z_type value of 'armadillo' sorts after 'Zebra'.

```
Select
  z_type As "Type"
 , z_cost AS "Price"
 , z_name As "Name"
 From zoo_2016
 ORDER BY z_type, z_cost;
```

Type	Price	Name
Armadillo	490	Abigail
Giraffe	3750	Dewey
Giraffe	5000	Sam
Giraffe	5000	Geoff
Giraffe	5000	
Giraffe	5000	
Giraffe	5000	Arnold
Giraffe	5000.25	Sally Robinson
Horse	490	
Lion	1850	Leon
Lion	5000	Lenora
Lion	5000	Leon
Zebra	2500.25	Huey
Zebra	2500.25	Dewey
Zebra	2500.25	Louie
armadillo	490	Anders
armadillo	490.01	Anne

Demo 09: This is a two level sort. The first sort key is the z_type and it is ascending. The second sort key z_cost uses a descending sort.

```
Select
  z_type As "Type"
 , z_cost AS "Price"
 , z_name As "Name"
 From zoo_2016
 ORDER BY z_type, z_cost desc;
```

Type	Price	Name
Armadillo	490	Abigail
Giraffe	5000.25	Sally Robinson
Giraffe	5000	Sam
Giraffe	5000	Geoff
Giraffe	5000	
Giraffe	5000	
Giraffe	5000	Arnold
Giraffe	3750	Dewey
Horse	490	
Lion	5000	Leon
Lion	5000	Lenora
Lion	1850	

Lion	1850
Lion	1850 Leon
Lion	1850
Zebra	2500.25 Huey
Zebra	2500.25 Louie
Zebra	2500.25 Dewey
armadillo	490.01 Anne
armadillo	490 Anders

Demo 10: The Oracle default is that nulls sort as a high-valued data item. We have some animals with no name value. They are sorting at the end of this display

```
Select
  z_type As "Type"
 , z_name As "Name"
 From zoo_2016
 ORDER BY z_name;
```

Type	Name
Armadillo	Abigail
armadillo	Anders
armadillo	Anne
Giraffe	Arnold
Giraffe	Dewey
Zebra	Dewey
Giraffe	Geoff
Zebra	Huey
Lion	Lenora
Lion	Leon
Lion	Leon
Zebra	Louie
Giraffe	Sally Robinson
Giraffe	Sam
Lion	
Giraffe	
Lion	
Horse	
Lion	
Giraffe	

Demo 11: You can specify a NULLS FIRST or NULLS LAST option. This is a NULLS FIRST sort and the nulls appear at the start of the result set.

```
Select
  z_type As "Type"
 , z_name As "Name"
 from zoo_2016
 ORDER BY z_name NULLS FIRST;
```

Type	Name
Lion	
Lion	
Lion	
Giraffe	
Giraffe	
Horse	
Armadillo	Abigail
armadillo	Anders
armadillo	Anne
Giraffe	Arnold
Giraffe	Dewey

Zebra	Dewey
Giraffe	Geoff
Zebra	Huey
Lion	Lenora
Lion	Leon
Lion	Leon
Zebra	Louie
Giraffe	Sally Robinson
Giraffe	Sam

Demo 12: Using a NULLS FIRST sort with the names in descending order. The Nulls First/Last option places the nulls at the start or at the end of the result set.

```
select
    z_type as "Type"
    , z_name as "Name"
from zoo_2016
ORDER BY z_name DESC NULLS FIRST
;
```

Type	Name
Giraffe	
Lion	
Giraffe	
Lion	
Lion	
Horse	
Giraffe	Sam
Giraffe	Sally Robinson
Zebra	Louie
Lion	Leon
Lion	Leon
Lion	Lenora
Zebra	Huey
Giraffe	Geoff
Zebra	Dewey
Giraffe	Dewey
Giraffe	Arnold
armadillo	Anne
armadillo	Anders
Armadillo	Abigail

Demo 13: You can sort on a date value.

```
Select
    z_id
    , z_dob as "BirthDate"
    , z_name as "Name"
From zoo_2016
ORDER BY z_dob DESC
;
```

Z_ID	BirthDate	Name
47	10-JAN-15	
44	02-JUN-14	Dewey
23	15-MAY-14	Sam
85	15-MAY-14	Sally Robinson
257	15-MAY-14	Arnold
57	25-MAR-14	Lenora
52	06-JUN-13	Dewey

```

43 02-JUN-13 Huey
258 15-MAY-13
373 25-FEB-13
374 25-FEB-13
372 25-FEB-13 Leon
375 25-FEB-13
371 15-JAN-13 Anne
370 15-JAN-13 Anders
25 15-JAN-13 Abigail
45 02-JAN-13 Louie
56 25-FEB-10 Leon
260 15-MAY-02 Geoff
259 15-MAY-02

```

Demo 14: Oracle allows you to sort by an alias. But if this is a quoted alias, then the sort key must also be a quoted alias.

```

Select
  z_id
, z_dob  as "Date of Birth"
, z_name as "Name"
From zoo_2016
ORDER BY "Date of Birth"
;

```

Z_ID	BirthDate	Name
259	15-MAY-02	
260	15-MAY-02	Geoff
56	25-FEB-10	Leon
45	02-JAN-13	Louie
25	15-JAN-13	Abigail
370	15-JAN-13	Anders
371	15-JAN-13	Anne
375	25-FEB-13	
372	25-FEB-13	Leon
374	25-FEB-13	
373	25-FEB-13	
258	15-MAY-13	
43	02-JUN-13	Huey
52	06-JUN-13	Dewey
57	25-MAR-14	Lenora
85	15-MAY-14	Sally Robinson
257	15-MAY-14	Arnold
23	15-MAY-14	Sam
44	02-JUN-14	Dewey
47	10-JAN-15	

Demo 15: Oracle allows you to sort by the column number. This is generally considered poor style since it is easy to rearrange the column in the select and forget to adjust the Order By clause. You want to write SQL that is easier to write correctly and harder to write incorrectly.
This will sort by the z_type values then by the z_name values.

```

Select
  z_id
, z_type
, z_name
From zoo_2016
ORDER BY 2,3
;

```

Z_ID	Z_TYPE	Z_NAME
25	Armadillo	Abigail
257	Giraffe	Arnold
52	Giraffe	Dewey
260	Giraffe	Geoff
85	Giraffe	Sally Robinson
23	Giraffe	Sam
258	Giraffe	
259	Giraffe	
47	Horse	
57	Lion	Lenora
56	Lion	Leon
372	Lion	Leon
374	Lion	
375	Lion	
373	Lion	
44	Zebra	Dewey
43	Zebra	Huey
45	Zebra	Louie
370	armadillo	Anders
371	armadillo	Anne

Demo 16: You can sort on calculated columns, either by using the alias or repeating the calculation as the sort key. We discuss calculation later; this is included here for completeness. extract (Month...) returns the numerical value of the month.

```
select
  z_id
, extract ( month from z_dob ) as "Birth Month"
, z_dob
, z_name as "Name"
from zoo_2016
order by extract ( month from z_dob ) ;
```

Z_ID	Birth Month	Z_DOB	Name
45	1	02-JAN-13	Louie
25	1	15-JAN-13	Abigail
371	1	15-JAN-13	Anne
370	1	15-JAN-13	Anders
47	1	10-JAN-15	
56	2	25-FEB-10	Leon
374	2	25-FEB-13	
375	2	25-FEB-13	
373	2	25-FEB-13	
372	2	25-FEB-13	Leon
57	3	25-MAR-14	Lenora
258	5	15-MAY-13	
259	5	15-MAY-02	
260	5	15-MAY-02	Geoff
23	5	15-MAY-14	Sam
85	5	15-MAY-14	Sally Robinson
257	5	15-MAY-14	Arnold
52	6	06-JUN-13	Dewey
43	6	02-JUN-13	Huey
44	6	02-JUN-14	Dewey

Table of Contents

1. Creating and using a script file	1
2. Creating a spool file.....	1
3. Creating and handling the script and spool files.....	2
4. Creating your script on hills.....	3

1. Creating and using a script file

For interactive SQL you are limited to a single SQL statement that is placed in the SQL buffer and then executed. A script file is simply a text file that can contain multiple SQL statements and SQL*Plus commands that you wish to save as a file and then run. A script file is an external file, an operating system level file—it is not part of your schema. If you are doing all of your work on the CCSF Linux system, then your script files will be stored as text files on your Linux account but not in your Oracle schema.

Most of the class work will have you create a script file of the various steps involved in the assignment.

You can create script files using any text editor. It is traditional to use the extension `.sql` for Oracle script files. You can create the script file on your local computer as you develop and test the queries. Then you can upload your script file to your hills Linux account to do the final run.

You run the script file with the `START` command or use the symbol `@` to execute the script file. These commands are given from the `SQL>` prompt **inside the SQL*Plus client**.

```
START filename.sql
@ filename.sql
```

When your script finishes executing, the last SQL statement from the script is left in the buffer.

2. Creating a spool file

The technique to create spool files for this class is given here. The command are given from within the SQL*Plus client. (If you have Linux experience you might be thinking of the Linux spool process- this is similar but not quite the same. For this class I do*not* want a Linux spool. I also do not want a log file.)

Assume that you have a file named `demo_1.sql` that is stored in your 151A directory. This file- the script file- contains the sql queries you want to run. And you have changed to that directory before you started SQL*Plus.

You give a command to start the spooling process. Then use the `@` command to run the script file. Stop the spooling process by giving the `spool off` command. This produces a text file in your Linux account - in the 151A directory- that you can then print or send to me for grading.

It is traditional to use the extension `LST` for spool files. The commands you use are:

```
SQL> spool demo_1.LST
SQL> @demo_1.sql
-- your script will run here
SQL> spool off
```

For this to work as shown here, your script file must be in the directory you were in when you started SQL*Plus. Remember the sequence of steps I asked you to use.

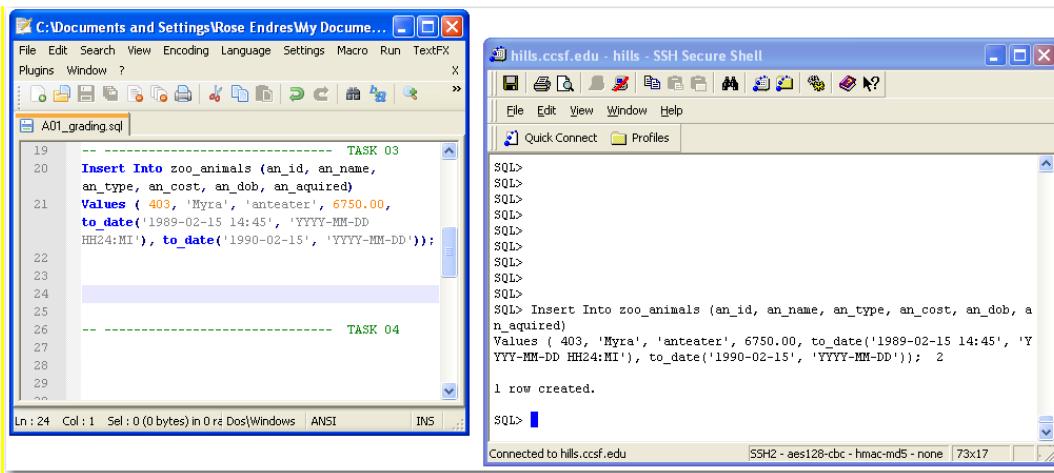
- log into hills
- change to your directory for your files- 151A
- log into Oracle

If you do this and save your script files in that directory, you can use the command above. Your spool file will be written to the same directory. The file names are case specific.

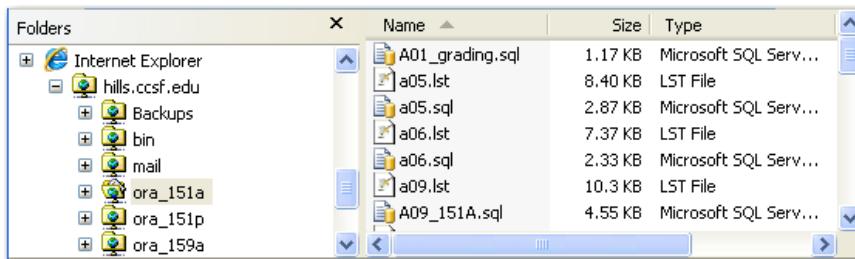
3. Creating and handling the script and spool files

The most common way to do this is to open a text editor on your local computer and start to build the script file locally; open the script file for this assignment in a text editor. Open another window to connect to hills and Oracle.

Write the SQL statement in the editor window and copy and paste to the SQL*Plus window and run it. If the query is correct, resave the script file. This way you can work on the script for a while and come back to it later. You can save an incomplete or incorrect query in your script and correct it later.



When you have all of the queries written and tested, then you need to **upload your script file to hills** to run it as a script. This shows the ssh secure shell ftp client.



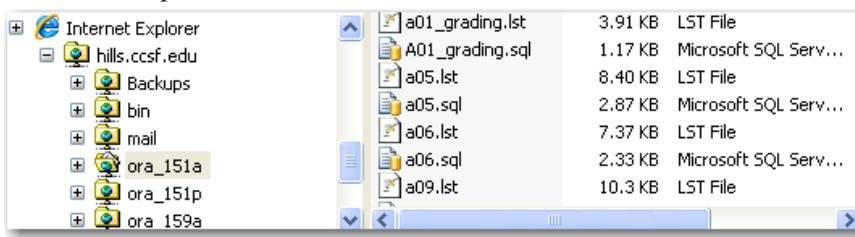
Now you can run that script file from SQL*Plus

```

SQL> spool a01_grading.lst
SQL> @A01_grading.sql

```

Refresh the ftp client window



When you are ready to turn in your assignment, drag the LST file to your local computer and then zip the two files together and load the zip file to Insight.

4. Creating your script on hills

I have suggested creating your script files on your local computer and uploading them to hills to use to create the spool file. For most people that is the easiest way.

But if you are comfortable working directly on the hills system and know how to use a text editor there, you can upload the assignment template file to your hills account and make copies of it and edit the script file on hills. I am not going to discuss linux editors in this class.

Table of Contents

1. Assignment Template	1
2. Modifying the script file	2
3. Make copies of script file	3
4. Filling the script.....	3
5. Testing your script-to-spool process.....	3
6. Turning in the assignment	3
7. File name problems I have seen in the past and do not want to see again	4

This document discusses some of the specific rules for assignment script and spool files.

In order to grade your assignments, I need to see both the SQL that you write and the result of running that SQL. Most assignments consist of 8-10 separate SQL queries. It is likely that you will have to experiment with creating many of those statements. I expect you to make mistakes as you learn these techniques- but I do not want to see all of the mistakes that you make while developing the statements. The way to handle this is to build a file of correct SQL query statements and then run the file as a unit. That file is called a **script file**. When you run the script file, you can tell SQL*Plus to keep a copy of the results- that file is called a **spool file**. With this approach you can turn in an assignment with no typing errors.

1. Assignment Template

Rule 1: Use the template file that I have provided. This file is included on the download page and will work for almost every assignment. If you do not use the template file you are apt to repeatedly lose points on assignments. There is a 10 point penalty for not using the template and additional penalties for skipping parts of the template in your script.

These are the first few lines of the template file:

```

set echo on
set feedback 1
set pagesize 999
set trimspool on
set linesize 200
set tab off
clear columns

replace this line with a comment giving your name; I do *not* want your id,
assignment number, etc.- just your name

/* TASK 00 */
select user, sysdate
from dual;

/* TASK 01 */
replace this line with your SQL for task 01

/* TASK 02 */

/* TASK 03 */

```

- Leave the set commands in the script
- Then include a comment with your full name. Do not include any other header comments. I know which class and assignment this is- you do not need to tell me; I do **not** need your student id, email, the current

date, etc. Some instructors want a more complete documentation style- The documentation for this class is very simplistic. Note that it says to "replace this line" and you need to have your name as a comment.

```
/* your name */
```

how hard could that be? Be certain to have a space after the /*

- There is a Task 00. Leave this in the script without any change.
how hard could that be?
- Include the task number as a comment for each task in the assignment. These are in the template. Do not change the Task comments. I want this style and wording of the task comments.

It is essential that you look at the output file produced. Would you be able to grade this? Can you find the SQL used for a particular task followed by its result? When I grade assignments, I grade all assignments at the same time- first grading all of the task1 steps, then all of the task 2 steps etc. So I need to be able to find your sql and your results quickly.

Since this is a script/spool process, the files you turn in should have no mistyped commands. When you read your results file, if you see a mistyped SQL command, you should correct the script and reload it up to dunes and then rerun it. It takes very little time to run an assignment script to an output file. Typing errors which remain in a results file will cost you points. You can experiment with the queries one at a time until you get them correct and then add them to your script file.

If the SQL query is not included in the spool file, you will lose major points.

If the various steps are not correctly numbered, you will lose major points.

Sometime you will find a query that you cannot do. Include the comment for the Task number and a comment that you skipped that task.

```
/* TASK 08 */  
-- Omitted
```

2. Modifying the script file

These are some ideas about how to handle the mechanics of creating the scripts for the assignments.

You should create a folder/directory on your **local** system to save your work. The name of my directory is c:\db_scripts.

Download the template file into that directory and change the name of the file to A01_yourLastName.SQL . Obviously I do not mean to literally use the letters yourLastName, but I did not want to make 100 different copies of this document -one for each student. If you have a common last name, then use A01_yourLastName_yourFirstname.SQL.

Open the file in a text editor and REPLACE the direction line with a comment giving your name. For example, if I were a student, I would use the following line after the set commands..

```
/* Rose Endres */
```

Then there are three lines; the first is a comment and the second and third is a command that will display some information; leave this in the script.

```
/* TASK 00 */  
select user, sysdate  
from dual;
```

Next follows a set of comments for task numbers that corresponds to the tasks in the assignment. Leave these comments in the script and add your SQL after the appropriate task number. Do not change the comment style or wording.

```
/* TASK 01 */
```

Save your file. Be certain that the file name extension is SQL. Since you are working with a Linux system, the file names will be case sensitive. I do not care about the case of the file name- just be consistent.

3. Make copies of script file

Now that you have modified the template with your name, you can go ahead and make 16 copies- one for each assignment. Change the file names to match the assignments.

A01_yourLastName.SQL
A02_yourLastName.SQL
A03_yourLastName.SQL
A04_yourLastName.SQL

4. Filling the script

You can open the script file in a text editor and fill in the SQL commands for each task and test them in SQL*Plus.

Save the script as you work. If you have troubles with one of the tasks, you can skip it temporarily and go on to the next. You can do some of the tasks, save the file and take a short break and then come back to work on other tasks. You need that script file to run the assignment, so it makes sense to me to build it up this way.

You should test the script to spool process occasionally as you build the scripts but the sql you execute in the SQL*Plus client window should run the same way with the script-to-spool process. But be certain to test that early enough that if you have a problem you can fix it in time to turn in the assignment on time. (When we discuss SQL Developer- this will not be true- SQL developer is a different client.)

I do get people who turn in a spooled file that is mostly empty and they do not get a chance to correct this.
READ YOUR SPOOL FILE. I have to read it; it is only fair that you read it also.

The spooled file should contain

- the set commands
- your name as a comment
- Task 00 as provided in the template
- the task number for each task as a comment as provided in the template
- the sql query(queries) needed at each step
- the output for each step

5. Testing your script-to-spool process

See the direction in 01-06 Section 3 for Running script. Read your LST file. If it has error messages or other errors, correct the script and resave it and reupload it to dunes then rerun the script-to-spool.

6. Turning in the assignment

After you have written and tested your script and have created the spooled file and have read it for possible problems, then it is time to zip the two files. You need to download the files to your local computer. You can use the windows menu (send to compressed folder) or other file compression techniques that open with 7-Zip. The compressed file should use your name (such as A01_endres.zip). Do not zip at the folder level- just zip the two files.

7. File name problems I have seen in the past and do not want to see again

You turn in files with the name A01.txt and A01.1st-- you lose 10 points for having the wrong file extensions and 10 points for the files not including your name. (did you notice that the extension was a digit 1 and not the letter l?)

You turn in files with the name A01.sql and A01.lst-- -- you lose 10 points for the files not including your name.

There are two people in class with the same last name. You can check the participants listing in Insight and if you have the same last name as another student, please use the naming pattern

A01_yourLastName_yourFirstName.SQL

The slq file and the lst files are named correctly but the zip file has a name such as A01.zip. You lose 10 points.

Following directions is actually important. Use your creativity in writing queries- not in naming files.

8. Another problem I see occasionally.

Your script file looks ok but your spool file is incomplete- it is cutoff. This occurs when you do not give the **spool off** command. the first part of the spool file is saved but it is incomplete. READ YOUR SPOOL FILE.

Table of Contents

1. Server software choices	1
1.1. Using the CCSF Oracle installation	1
1.2. Using a local Oracle installation	1
1.3. Using a different Oracle installation	1
1.4. Why is this a choice?	1
2. Client software choices	2
2.1. SQL*Plus.....	2
2.2. SQL Developer.....	2
2.1. Other clients.....	2
2.2. Using graphical clients	2
3. Communication software choices.....	2
4. Text editor	2

In this class, we focus on using interactive Oracle SQL using Oracle 12g. To work with our data we need a server (Oracle 12g) and a client. Your simplest choice is to use a communication program to log into your CCSF student linux account and then log into the SQL*Plus client using the CCSF student Oracle installation. But you do have some other choices.

1. Server software choices

1.1. Using the CCSF Oracle installation

You can access the student installation of Oracle from your CCSF Unix account. This installation runs on our dunes Linux system. Your student account is automatically created for you if you are enrolled in this class. This is generally the easiest approach since the Oracle server is set up and running and your account already exists.

On the CCSF Oracle system, each student gets one user account which is considered to be their database and all of their tables are created in that database. Oracle also refers to this as your schema. When you log into your Oracle account, you are in your private schema; other students cannot see your tables. I cannot see your tables either, since my Oracle account has the same privileges as your Oracle account.

1.2. Using a local Oracle installation

You can also download Oracle software from the Oracle site and use this for your assignments. There is a free downloadable Oracle 11g Express Edition which was a rather simple install on a Windows system. It is also available as a Linux version. This would allow you to create different users if you wanted.

I do not provide assistance in installing and configuring Oracle on your computer.

Having troubles doing a local installation will not be accepted as a reason for late assignments.

If you do this be certain you have the 12c version- not the older 10g version of Oracle express

1.3. Using a different Oracle installation

Occasionally students use an Oracle system that is set up on their job for their assignments. I do not recommend this since you could make errors that damage your company's data. A good dba would not want you doing class assignments on a production server.

1.4. Why is this a choice?

The assumption for this class is that each student has a separate user account, which is a separate database. As long as you are using Oracle 12g (or 11g) and install the tables using the scripts I give you, your assignments should work the same way.

2. Client software choices

2.1. SQL*Plus

We start with SQL*Plus, which is a command line Oracle client tool that lets you create, edit and run interactive SQL queries and PL/SQL blocks. You can use SQL*Plus to create and run script files. The result of the queries that you run is determined by both the SQL statement that is executed by the Oracle server and the current settings of the SQL*Plus environment. It is important to recognize the difference between SQL*Plus commands and SQL statements. SQL*Plus is the client available on the CCSF system. SQL*Plus is available on most Oracle installations.

Even if you commonly use a graphical client (such as SQL developer) you need to be able to use SQL*Plus.

The script-spool process for assignments needs to be run from within SQL*Plus.

2.2. SQL Developer

You may want to use SQL Developer to develop some queries so that you see a graphical client. We will discuss this in a few weeks. In general, I will not know which client you are using to develop your queries. You should be comfortable with both a command line interface (SQL*Plus) and a graphical interface (SQL Developer). SQL Developer is available as a free download from Oracle. You can connect to the CCSF Oracle server using SQL Developer.

SQL Developer has a few different client level rules than SQL*Plus. Some of the commands you are required to use for scripts do not work in SQL Developer because they are client commands, not SQL commands. You need to always double check your script and spooled results for assignments. (the major cause of errors in the that SQL Developer client will allow blank line in the middle of a query and SQL*Plus will not- that would cause an error in your script.)

2.1. Other clients

There are other clients that let you create queries and run them against an Oracle database. If you find them helpful in developing queries, please use them. If they are free, please tell other people in class about them.

2.2. Using graphical clients

You should consider using graphical clients if it helps you understand how the queries are created. Many graphical client use color coding for syntax and provide lists of relevant table and column names.

3. Communication software choices

You can use many different communication tools to get to the CCSF Linux system. You want a tool that opens a client window where you can enter commands to log into your account and to start up SQL*Plus.

You also will be transferring some files between your local computer and your linux account. This is usually done with an FTP client.

For Windows users we commonly recommend the use of SSH Secure Shell; Mac users commonly use Fugo (See the ACRC handouts for that). Any other tool that lets you connect to your Oracle account and use SQL*Plus and that lets you transfer plain text files can be used.

4. Text editor

It is a good idea to have a text editor to use when writing your queries. If you are using the SQL*Plus client, it can be easier to enter the SQL query into the text editor and then copy and paste it into the SQL*Plus client. There is an editor for the SQL*Plus client, but it is not intuitive.

Many people find NotePad++ a good text editor for Windows systems. It is a free download. MS Word is NOT a good choice- it is not a text editor.

Table of Contents

1. The SQL buffer.....	1
2. Editing the SQL buffer using a text editor	1
3. Editing the SQL buffer using the Line Editor	1
3.1. Walkthrough.....	1
3.2. Rules for the SQL*Plus line editor	3

SQL*Plus is a command line interface that provide access to your data via SQL statements. The basic SQL*Plus features were described in a previous document. This document is included for people who want to know how to use the SQL*Plus line editor. **It is not intuitive and you do not need to use it for class.**

1. The SQL buffer

SQL*Plus maintains a buffer called the SQL buffer.

- The SQL buffer contains the most recent SQL statement that you entered.
- SQL*Plus commands do not get stored in the SQL buffer.
- Executing the SQL statement does not empty the SQL buffer.
- You can re-execute the command in the SQL buffer with the / command.
- You can edit the SQL statement in the SQL buffer

2. Editing the SQL buffer using a text editor

You can give the edit command which loads the contents of the SQL buffer into your designated editor. The editor on the Unix system might be pico or vi. When you exit the editor, you load the file back into the SQL buffer. The default name of the file is afiedt.buf; if you are using this file simply to edit the **buffer**, you can keep that filename. You might see that name in your file system.

3. Editing the SQL buffer using the Line Editor

For small changes you might find the SQL*Plus line editor useful.

This first thing to be aware of is that this is a line editor and changes are made to the current line only. This is not the way we are used to in the windows world- or in most visual editors. As you add and delete lines from the SQL buffer, the lines get renumbered. The line numbers are supplied automatically and are not part of your SQL statement. Use the List command to redisplay the buffer. The current line is indicated by a * after the line number.

3.1. Walkthrough

First let's go through a simple walkthrough. Using the zoo_2015 table, you enter and execute the following sql statement.

```
SQL> select z_name, z_type
  2  from zoo_2015
  3  where z_type ='Lion'
  4  order by z_name;

Z_NAME          Z_TYPE
-----          -----
Lenora          Lion
Leon           Lion

2 rows selected.
```

You can now give the List command (abbreviation l) and see the content of the buffer

```
SQL> 1
  1 select z_name, z_type
  2 from zoo_2015
  3 where z_type ='Lion'
  4* order by z_name
SQL>
```

Suppose I want to change the SQL statement in the buffer and select for zebras instead of lions. I need to edit line 3. I type a 3 and line 3 becomes the current line.

```
SQL> 3
  3* where z_type ='Lion'
SQL>
```

The following shows a change command. The / characters are used as delimiters and the command says to change the first occurrence of the pattern Lion to Zebra on the current line. We get feedback on the change.

```
SQL> c/Lion/Zebra
  3* where z_type ='Zebra'
```

Now use the slash command and we find the zebras.

```
SQL> /


| Z_NAME | Z_TYPE |
|--------|--------|
| Dewey  | Zebra  |
| Huey   | Zebra  |
| Louie  | Zebra  |


3 rows selected.
```

If you like this, you must be a Unix person at heart. Try a few other changes.

The del command deletes the current line

The i command lets you insert a new line after the current line

```
SQL> 3
  3* where z_type ='Zebra'
SQL>
SQL> del
SQL> 2
  2* from zoo_2015
SQL>
SQL> i where z_type = 'Giraffe'
SQL>
SQL> L
  1 select z_name, z_type
  2 from zoo_2015
  3 where z_type = 'Giraffe'
  4* order by z_name
SQL>
SQL> /


| Z_NAME         | Z_TYPE  |
|----------------|---------|
| Dewey          | Giraffe |
| Sally Robinson | Giraffe |
| Sam            | Giraffe |


3 rows selected.

SQL>
```

3.2. Rules for the SQL*Plus line editor

Remember this is an editor for the SQL*Plus client. If you are using another client, this does not apply.

The SQL*Plus line editor is editing the SQL buffer—it is only doing an edit of the last SQL query you entered and does not directly provide the ability to save this as a file.

You cannot go back up through a command history.

The semicolon (;) that you use to run a query is not stored in the buffer. Do not add a semicolon to the last line when you edit the buffer. After you have edited the buffer, you can list it with the List command and run it with either the slash command or the run command.

Below is a table of the SQL*Plus Line Editor commands. "m" and "n" refer to line numbers in the buffer; the asterisk (*) is used to indicate the current line; "text", "old" and "new" are text strings.

Errors often make the line with the error the current line. Use the list command and then select the line you want for the current line.

Editing Command	Abbrv	Function
RUN	R	display and execute the SQL statement in the buffer
CLEAR BUFFER	CL BUFF	delete all lines from the buffer
LIST	L	list all lines in the buffer; this also moves the current line pointer to the last line in the buffer
LIST n	L n	list line n in the buffer and makes that line the current line
LIST m n	L m n	list lines m through n in the buffer
LIST *	L *	list the current line in the buffer
LIST LAST	L LAST	display the last line and make that the current line
LIST n	L n	display line n through the last line and make that the current line
n		list line n and make it the current line; this does not work for a range of lines
n text		replaces line n with the indicated text to insert a line before the first line use 0 text
CHANGE /old/new/	C/old/new/ C/old/new	change text on the current line. The separator character can be anything except a letter or digit or space. Generally, you do not need the last separator character. Change affects only the first occurrence of that text in the current line. Change is not case-specific for the old text. In the change command, you can use ellipses in the old pattern. C/...old/new/ changes everything on the current line up to and including old with new. C/old.../new/ changes everything from old to the end of the line. C/old1...old2/new/ changes everything from old1 to the next occurrence of old2.
CHANGE /text/	C/text/	remove the indicated text from the current line
APPEND text	A text	append the indicated text to the current line; don't forget any blanks that you need; use one space after Append before your text.
DEL		delete the current line
DEL n		delete line n
DEL m n		delete lines m through n

DEL LAST		delete the last line
INPUT	I	insert a series of line after the current line; terminate input with a blank line
INPUT text	I text	insert the text as a line; use this to insert a single line.

Table of Contents

1. Terms.....	1
2. Relational databases	1
3. Client, application	1
4. DBMS features	2
4.1. Physical data independence.....	2
4.2. Logical data independence	2
4.3. Data integrity.....	2
4.4. Query optimization	2
4.5. Concurrency control.....	3
4.6. Database security.....	3
4.7. Backup and recovery	3
5. Business entities and database components.....	3
6. Relational databases and relations and relationships	5

In the first unit, we had a brief intro to some database terms. In this unit and the next few units we will explain these types of terms in a bit more depth. This document will seem rather dry, but go though it and come back next week and read it again. After a reading or two (or three) you should be able to understand these terms in context.

1. Terms

Application	Data type	Relational database
Attribute	Database	Relationship
Base table	DBMS	Row
Client	Foreign key	Schema
Column	Normalization	Virtual table
Data independence	Primary key	
Data integrity	Relation	

2. Relational databases

A **database** is a collection of related data organized for some purpose. Some people will use the term database for any type of storage including paper files. Most people will assume that the database is a computer system managed by a collection of programs called a database management system (dbms).

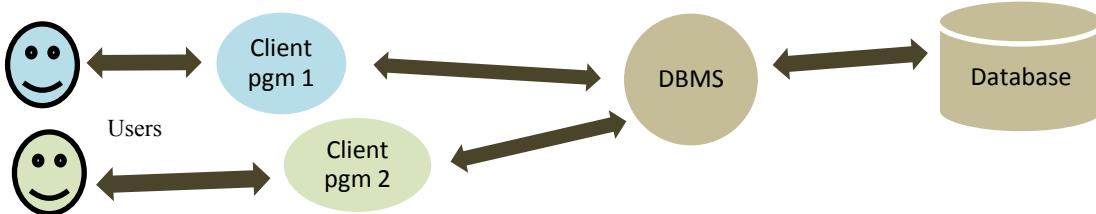
A **relational database** stores data in structures called **relations** which can be represented by two-dimensional tables. A relation is a mathematical concept- but we can think of it as a table. We will discuss relations in a bit more detail later in this document.

3. Client, application

These are terms that can have several different definitions. The way I will use them is that the client is a purpose that lets us work with the database. The client lets us enter SQL statements and see the results. The client lets us build database objects such as tables. We can also run scripts via the client. But as you will see over the semester, coding SQL statements is not the appropriate way for most end-users to get results from a database. We do not expect students enrolling in classes to do this by coding Insert statements. For an end-user we want to provide a more specialized interface- probably using menus and text boxes and buttons. The way an end-user interacts with a database is generally through a specialized application that provides an interface appropriate to the end-user. We can have an application that handles student enrolling in classes and a different application that allows staff to build the semester schedule of classes. The application program might use SQL statements to work with the database. In this class, we do not build the application level software. We work at the SQL level.

4. DBMS features

The **DBMS** is a collection of programs that manage the database. When we issue an SQL select statement, we enter it in a client program which sends the SQL statement to a translator program which changes our text into a query that can be processed. Then another part of the dbms takes that request and gets the data from the physical data in storage. Now the DBMS send the data back to the **client** program which may need to do some additional steps to format it for our display.



4.1. Physical data independence

One of the advantages of using a dbms is that the end-user and, to a large extent, the developer does not need to know how the data is stored physically on the persistent storage devices - often hard drives. The storage details are more in the realm of the database administrator (dba). For example, the data stored on the disk might be organized in a particular physical order to make retrieving data more efficient. The SQL to retrieve the data does not logically depend on that ordering. If the physical order of the data in persistent storage is changed, the SQL application does not need to be changed. If a developer wants to store a date value, she should not be concerned about how the data value is stored in terms of the bit patterns. In reality, a developer often has to know something about the storage of data to write the most efficient queries and to decide on the proper data types to use.

4.2. Logical data independence

With logical data independence we are talking about the ability of the dbms to access data even if the organization of the underlying data is altered. For example, suppose you define a table that stores the student's street address and their zip code as two attributes. It should not make any difference to your code whether the data is stored with the address first in the persistent storage or the zip code first. In a relational database the order of the attributes in the table can have no logical significance. Suppose I decide to add a new column to the `zoo_2016` table and allow that column to be null; your SQL code for A01 should still work.

4.3. Data integrity

The dbms is responsible for protecting the integrity of the data. This has several aspects. If the developer defines an enrollment date as a date value, the dbms should reject an attempt to store a value such as Feb 31, 2009 since that is not a valid date. If the developer defines an attribute as an integer number between 1 and 15, then the dbms should reject an attempt to store a value such as 51 since it is out of range.

Systems vary in the proper response in some situations. Suppose the developer defines a last name attribute as a string field with a maximum of 15 characters. What should the dbms do with an attempt to enter a value that is longer than 15 characters? One approach is to reject that value and another approach is to store only the first 15 characters of that value.

4.4. Query optimization

It might be that our query is rewritten in some way by the translator program to make it more efficient. This is one of the jobs of the dbms (optimization) and the dbms should not change the logic of your request. The dbms generates a plan of execution as to how it will get the data from the tables. For our first sets of queries this will be a pretty simple plan- but as we work with queries that get data from several different tables and as our tables get bigger, this execution plan will become more important. We do not focus on query efficiency in this class as our job to create a query to return a correct result set; but in a database with tables with thousands of rows of data, efficiency is much more important.

4.5. Concurrency control

In our class assignments you will be the only user accessing your tables; in more realistic situations a dbms will have to handle multiple users working with the data at the same time. For example, we might have two students trying to enroll in the last open seat for a class. The dbms handles this but the way that a developer writes code can affect this situation. This feature, concurrency control, is only briefly discussed in this class. We discuss this topic more in the database programming classes.

4.6. Database security

Database security- in terms of user access, is handled by authentication and authorization. Authentication means identifying a user- generally via a login and password. Authorization deals with a set of privileges that the user has been granted.

When we log into the database, the dbms checks our user name and password to see if we have the right to login. Then it checks what types of work we are allowed to do- can we create tables? Or just read tables that already exist? Can we change the data in the tables?

4.7. Backup and recovery

This is another topic that is more dba oriented. Our tables for class are not storing mission critical data (except for your turning in assignments on time). We have a set of SQL statements to build the database and load the tables with preset data. So you can rebuild the database if needed. Hopefully you will save your scripts for the queries you create.

But for an online real time system, backup and recovery is critical; CCSF would be in trouble if the enrollment system failed and we had to ask students to enroll in their classes again. You would not be happy if your bank's database failed and you had no way to determine the status of your bank accounts. A major dbms will include features to do backups of data while the system is running and help with recovery of the data. For a mission critical system this has to be a lossless recovery- not a backup from the last day of the previous month.

5. Business entities and database components

When we use a database in a company there is always some purpose for which the database was constructed. A **business entity** is something for which we want to keep data. Examples of entities are: students, customers, books in a library.

One of the early steps in designing a database is to decide on

- the entities that are important to our company
- the attributes we need to save for these entities
- what type of data the attributes store
- how the data values for one entity are related to other entities
- what rules we have about the data

The database that we are using for the next few weeks deals with a veterinary clinic and the exams the clinic does for animals. The entities we will consider are:

- the animals the clinic sees
- the clients who are responsible for those animals
- the clinic staff
- the exams the vets perform

These entities will be handled in the database by creating tables.

For an entity, the pieces of data we stored can be called **attributes**. Examples of attributes for an animal seen at the clinic include:

- the animal's name
- the type of animal
- the date the animal was born
- the client who is responsible for that animal
- the date of any exam
- the details of the exams for that animal

We will need to make some decisions about the attributes to be sorted. For this database I am simplifying some of these decisions. For example, I think we can agree that each animal has one value for its animal type- the animal might be a dog or a bird. For the clinic we can say that each animal has exactly one value for its animal type and it must have that value.

But we might not insist that the animal has a name. Maybe the vet is treating a new litter of puppies and the puppies do not yet have a name. But we might decide that we will only store one name for each animal.

We will expect that most animals will have multiple exams- so we will need a way to store multiple sets of exam data for each animal.

Suppose this is a representation of part of the table for animals.

an_id	an_type	an_name	an_dob	cl_id
10002	cat	Gutsy	2010/04/15	3560
11015	snake	Kenny	2012/02/23	4534
11025	bird		2012/02/01	4534
12035	bird	Mr Peanut	1995/02/28	3560
21004	snake	Gutsy	2011/05/12	5699

The attributes are stored in **columns** and each **row** in the table includes the attribute values for a specific entity (an **entity instance**). A column in a table has a name and a **data type**. For the animal table example, we could have the following

an_id(integer), an_type(character), an_name (character), an_dob(date) , cl_id (integer).

Each row in the table should say one true thing about an animal. The first row says "We have an animal named Gutsy which is a cat. The animal was born April 15, 2010 and is associated with the client with client id 3560. We are using the value 10002 as an id for this animal."

In this section of the table we have 5 rows- so we are saying 5 true things about animals. Notice that the following represents the same 5 true things.

an_id	cl_id	an_name	an_dob	an_type
11025	4534		2012/02/01	bird
12035	3560	Mr Peanut	1995/02/28	bird
10002	3560	Gutsy	2010/04/15	cat
21004	5699	Gutsy	2011/05/12	snake
11015	4534	Kenny	2012/02/23	snake

We can also define rules for the data, called **constraints** which limit the data values that can be entered. We might want a rule that the an_type values are limited to a certain set of values. We might want a rule that the values for an_id has to be a positive number.

The collection of all of the rows in the current table is called the **entity set**. An individual row can be considered an **entity instance**. We need to be able to distinguish the individual rows in our table, so each table should have a **primary key**- an attribute or collection of attributes that uniquely identifies that entity instance. No two rows in the same table can have the same value for the primary key and no row can be missing a primary key value. Here we will define the primary key for this table as the an_id attribute.

Traditional **normalization** rules say that each attribute should store a single value; this is often interpreted as storing a single simple scalar value. Normalization rules say that all attributes in a row should be determined by the primary key of that entity instance. We will come back to the concept of normalization later in the semester.

Part of database design is deciding how many tables we have and what data is stored in each table. When we store data in tables, we need to organize it efficiently. Often this means that we split the data into several tables and then build **relationships** between the tables.

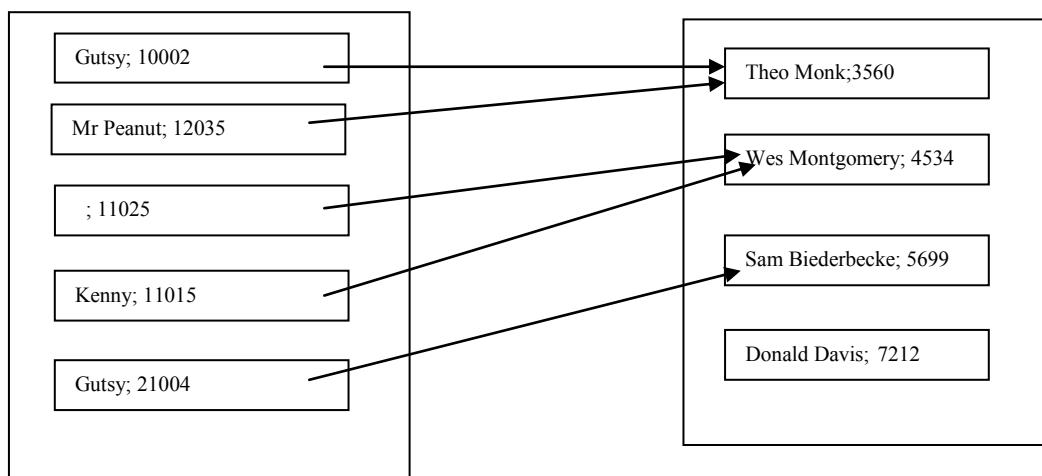
The term **base table** refers to a table that we define with the Create Table statement. Base tables hold data values. The term **virtual table** refers an in-memory data structure that appears to hold the data.

6. Relational databases and relations and relationships

The relational database model has been used for more than 40 years (a very long time for a computer model). The relational database model was developed by Dr Codd in the late 1960's and is the underlying model for the majority of database systems in use.

In the relational model, we think of the data as being stored in **tables** (relations) - with data being stored as a series of rows and columns. A relation stores a set of values that are related to the same entity. The data in one table will usually be associated with data in other tables in the database. The association between two tables is called a **relationship**. Relationships are implemented by the two related tables having data values in common.

In our vets database we have a table with client information. Each client gets a cl_id value which is the primary key for that table. We also store the client's name and address data in the clients table. The value for the cl_id attribute in the animals table has to match a value for cl_id in the clients table. In the animals table, the cl_id attribute is called a foreign key. Then we can navigate from a particular animal in the animal table to the related client in the client table. We can also navigate from a particular client in the client table to the animal in the animal table for that client



When we talk about a database as a logical thing, we think of a collection of tables that are logically related to each other. We may include other types of objects with the database- such as views and processing routines.

In summary: data values are stored in columns. Columns are grouped together into a row that represents an entity instance. Rows are grouped together into a table which represents a business entity that we care about. Tables are grouped together into a schema- a collection of tables. Schemas can be grouped together into a database.

Table of Contents

1. Assignment Rules	1
2. A note about the demos	1
3. Characteristics of SQL.....	1
4. Demos	2
5. SQL syntax guidelines.....	4
5.1. Identifiers	5
6. Statement terminator	5

SQL is a language for working with relational databases. These are some of the general characteristic of the SQL language and how to write statements using the SQL language. There are some differences in writing SQL statements for the various dbms; for this class I will emphasize the features that all versions of SQL have in common so that you could more easily transfer your knowledge to another dbms. But we will address the specifics of using Oracle SQL - as opposed to MySQL or T-SQL.

1. Assignment Rules

I have posted a document for the Assignment rules which lists the rules in effect for all assignments starting with A02. This include files names, use of the template and the layout for the SQL queries.

2. A note about the demos

With most of the document files, I will also include a text file that includes the sql used for the demos. The purpose of the demo file is to give you sample sql queries that you could run and modify to try experiments. It also saves typing. You can open the demo file in a text editor and copy the sql into a client window. If you are using a gui client you can probably open the demo file in the gui client. The demo files are not intended to be run as a script- the queries are intended to be run one at a time and thought about.

Often I will display part or all of the result produced by the sql query in these documents. It is possible that the actual values may differ from the values that you get with the current dataset. The data set that I use when creating these document has the same tables as I provide, but sometimes the inserted data is different. That is ok since the query should work correctly on any data set that is currently in the tables.

The result that I post were created using ether the SQL*Plus or the SQL Developer client. It is possible that if you use a different client that the result you see may have a different format- the data might be displayed in a different way. The display of a Null often differs with clients- with some clients, the cell is left empty; with other clients, the cell contains the display NULL or *Null*. Numeric values might be displayed with different numbers of digits after the decimal point. These are formatting issues- not logical issues.

3. Characteristics of SQL

- SQL works with tables (either base tables or virtual tables) and it produces virtual tables. This feature is called closure. Base tables are tables you create with the Create Table statement; the data in a base table is stored in persistent storage. A virtual table is a collection of rows and columns that the computer has in memory but it is not stored to persistent storage. The result of your query is a virtual table.
- SQL is a declarative language. You do not tell the SQL engine step by step how to produce the result. Instead you write a statement that describes the desired output table. The SQL statement that you write is passed on to the DBMS, which processes it and returns the result to the client for display.
- The same SQL statements can be used by end-users, database programmers, and database administrators. The same SQL statements can be used interactively, collected in batch files (script files), or embedded into application programs.
- There are several basic categories of SQL statements.

Query statements are the Select statement used to display data

Data Manipulation Language (DML) statements are used to manage the data within the database- this includes modifying the data.

Data Definition Language (DDL) statements are used to create and modify the design of the database objects- such as tables and relationships.

Transaction Control statements are used to make changes to the database permanent or to roll them back

Data Control Language (DCL) statements are used to assign privileges to users.

- SQL is a redundant language. There are often several different ways to accomplish the same goal.
- The SQL language has an ANSI standard; most implementations of SQL follow the standard to some degree but also add additional features and have some variations in the way that SQL is written. In this class I will emphasize the standard techniques but will also address dbms variations.

4. Demos

These are examples of SELECT queries. They are based on the zoo_2016 table. You might have additional rows in your table. You can run the queries against your tables to see the actual output. The alignment of the data might be different depending on the client you use.

The method I will usually use for displaying the SQL and the output is shown here. The SQL statement is presented in the Courier New font. The output is boxed. You should be able to copy and paste the SQL statements into your client and run them and then try variations on the SQL. In some cases, I have reduced the column widths and limited the number of rows displayed to save space. When I introduce new SQL keywords I will show them in caps; keywords we have already used will be in lower case or Initcase.

Demo 01: All rows are displayed. The attribute names are used for the column headers; the column width is determined by the data being displayed.

```
SELECT z_name, z_cost
FROM zoo_2016;
```

Z_NAME	Z_COST
Sam	5000
Abigail	490
Leon	5000
Lenora	5000
Sally Robinson	5000.25
Huey	2500.25
Dewey	2500.25
Louie	2500.25
	490
Dewey	3750
Arnold	5000
	5000
	5000
Geoff	5000
Anders	490
Anne	490.01
Leon	1850
	1850
	1850
	1850

20 rows selected.

Demo 02: Select specific columns; add a column alias, and add a criterion to limit the rows that are displayed.

```
SELECT z_name, z_cost "Price more than 3K", z_type
FROM zoo_2016
WHERE z_cost > 3000;
```

Z_NAME	Price more than 3K Z_TYPE
Sam	5000 Giraffe
Leon	5000 Lion
Lenora	5000 Lion
Sally Robinson	5000.25 Giraffe
Dewey	3750 Giraffe
Arnold	5000 Giraffe
	5000 Giraffe
	5000 Giraffe
Geoff	5000 Giraffe

9 rows selected.

Demo 03: Select specific columns and add a criterion to limit the rows that are displayed.

```
SELECT
  z_dob
 , z_type
 , z_name
FROM zoo_2016
WHERE z_type = 'Giraffe';
```

Z_DOB	Z_TYPE	Z_NAME
15-MAY-14	Giraffe	Sam
15-MAY-14	Giraffe	Sally Robinson
06-JUN-13	Giraffe	Dewey
15-MAY-14	Giraffe	Arnold
15-MAY-13	Giraffe	
15-MAY-02	Giraffe	
15-MAY-02	Giraffe	Geoff

7 rows selected.

Demo 04: Rewrite the literal to match a different pattern. Oracle does string comparisons using case-specific rules.

```
SELECT z_dob, z_type, z_name
FROM zoo_2016
WHERE z_type = 'giraffe';
```

No rows selected

Demo 05: Sometimes a query does not return any rows. In that case, it does not display a header in the SQL*PLUS client.

```
SELECT z_name, z_cost "Price more than 20K", z_type
FROM zoo_2016
WHERE z_cost > 20000;
```

no rows selected

In the SQL Developer client, the display is the header but no data rows

Z_NAME	Price more than 20K	Z_TYPE
--------	---------------------	--------

Things to notice about the result table for the query result, so far:

- This is a text display.

- Column headers are the attribute names or aliases but may be truncated to fit the column width.
- Text columns are left justified; numeric columns are right justified in SQL*Plus and left justified in SQL Developer.
- Numeric values are not automatically formatted with a certain number of digits after the decimal point.
- The column widths for the text columns are determined by the defined data types.
- The column widths for the numeric columns are determined by the default value for numeric columns and the column header width.
- The appearance / format of the display is influenced by the client you are using.

5. SQL syntax guidelines

Even though there is an SQL standard, individual database systems use different dialects of the language. The version used with Oracle differs somewhat from SQL used in SQL Server and SQL used in Microsoft Access and SQL used in MySQL and from earlier versions of Oracle SQL. The SQL that you learn in this class will help you when you need to use SQL in other relational database systems.

- SQL is a free-form language. This means you could write query 2 as

```
SELECT z_name, z_cost "Price more than 2K", z_type FROM zoo_2016 WHERE z_cost  
> 2000;
```

But it is a lot easier to read if you start the keywords SELECT, FROM, and WHERE on new lines. For the class assignments, it is a rule that you start these keywords on new lines.

- SQL statements begin with a keyword - such as SELECT- and are composed of one or more clauses which begin with keywords such as FROM or WHERE. You should avoid using the SQL keywords as table or column names.
 - Although you will commonly see SQL keywords written in upper case, you can use either upper or lower case for SQL keywords and for table and column names.
 - You use commas to separate lists of items- such as the columns to be displayed.
 - Literals
 - If you use a constant (a literal) in an SQL statement, it may need to be delimited.
 - Numbers do not use delimiters.
 - Do not put commas or dollar signs in numeric literals.
 - Text literals are enclosed in quotes (')
 - In Oracle, dates should be enclosed in single quotes; the default syntax for Oracle date follows the pattern 11-AUG-07. We can use other date formats.
 - In Oracle, text comparisons are case-sensitive.
 - You can use comments in your SQL statement. There are three forms of comments
 - The single line comment is indicated by two hyphens followed by a space. This is the ANSI standard comment.
 - The multi-line comment is delimited by /* comment */ Start with a slash, an asterisk and a space. It is essential to have a space after the /* or SQL*Plus will misinterpret that line.
- To avoid problems with comments:
- Do not put comments near the start of your SQL statement.
 - Do not put comments on the same line after the semicolon of your SQL statement.
 - Do not put a semicolon, hyphen, or ampersand (&) in your comments.

5.1. Identifiers

A table exists within a schema which exists within a database. (When you log into your Oracle account you are already in your schema). A column exists within a table. So if we want to refer to a column, we need a multi-part name.

Within your schema, the name of the table is: `zoo_2016`

The name of the attribute storing the names of our animals is: `zoo_2016.z_name`

When we are in a query that uses the `zoo_2016` table in the From clause, we can refer to the attribute as `zoo_2016.z_name` or just as `z_name`. Using `zoo_2016.z_name` is called "qualifying" the column name.

- A qualified column name is one that includes the name of the table- such as `zoo_2016.z_name`.
- If your query uses only a single table, you do not need to qualify any column name.
- You need to qualify column names only if the SQL statement includes two or more tables and that column name appears in more than one of these tables.

If your query uses multiple tables, your query might be more efficient if you qualify all of the column names

Demo 06: Any of these will work

```
select z_name, z_cost  
from zoo_2016;  
  
select zoo_2016.z_name, z_cost  
from zoo_2016;  
  
select zoo_2016.z_name, zoo_2016.z_cost  
from zoo_2016;
```

6. Statement terminator

Each client needs to have a way of knowing when your sql statement is complete and you want to run it. This might be indicated by a statement terminator. Often the client uses a semicolon for this. So you will commonly see SQL written as

```
Select * from zoo_2016;
```

The semicolon is not actually part of the statement but it might be required by the client in order to run the statement.

The ANSI default character used is the semicolon. In the SQL*Plus client, the use of the semicolon is required.

You can have a blank line in the midst of an SQL statement in the SQL Developer client but not in the SQL*Plus client. Since SQL*Plus is used so commonly, avoid blank lines in your queries.

Table of Contents

1. Selecting distinct output rows.....	1
1.1. Sorting and distinct	2
2. Using RowNum.....	3

1. Selecting distinct output rows

The keyword DISTINCT can be placed after SELECT to specify that any duplicate copies of an output row will not be displayed. The decision is based on the uniqueness of the rows to be displayed, not the uniqueness of rows in the table.

Using DISTINCT takes extra processing time and should be avoided unless it is necessary to avoid duplicate output lines. If you are writing a query that uses a single table and displays the primary key, do not include DISTINCT.

If you are using DISTINCT, then you can sort only by columns listed in the Select clause.

Demo 01: Display one output row per row in the table

```
select z_type
      from zoo_2016;
Z_TYPE
-----
Giraffe
Armadillo
Lion
Lion
Giraffe
Zebra
Zebra
Zebra
Horse
Giraffe
Giraffe
Giraffe
Giraffe
armadillo
armadillo
Lion
Lion
Lion
Lion
20 rows selected.
```

Demo 02: Display one output row for each different value of z_type. Note that we get one row for 'armadillo' and another row for 'Armadillo', these are considered distinct because Oracle is case specific.

```
select DISTINCT z_type
      from zoo_2016;
Z_TYPE
-----
Zebra
armadillo
Lion
Giraffe
Armadillo
Horse
6 rows selected.
```

Demo 03: Display one output row for each different combination of values for z_type and z_cost .

```
select DISTINCT z_type, z_cost
from zoo_2016
order by z_type, z_cost ;

```

Z_TYPE	Z_COST
Armadillo	490
Giraffe	3750
Giraffe	5000
Giraffe	5000.25
Horse	490
Lion	1850
Lion	5000
Zebra	2500.25
armadillo	490
armadillo	490.01

10 rows selected.

Please note that Distinct is not a function **and it is inappropriate to use parentheses with Distinct**. You will often see queries that use the syntax Select Distinct (z_type) from zoo_2016, but those parentheses are simply parentheses you could use around any expression. You can write a query such as this where the parentheses are also legal but meaningless. Select (z_type) from zoo_2016;

1.1. Sorting and distinct

Return to the query

```
select DISTINCT z_type from zoo_2016;
```

Can we sort the output? With some dbms, the way a Distinct operation is implemented the output is commonly sorted.

Demo 04: So we will do a descending sort

```
select distinct z_type
from zoo_2016
order by z_type desc;
```

Z_TYPE
armadillo
Zebra
Lion
Horse
Giraffe
Armadillo

What if we want to display the animal type and sort the output by the animal name? Before we try this, think about what this means. We are displaying one row that represents all of the zebra, one row that represents all of the giraffes. If we sort by z_name- how would the rows be returned? If we change the order by clause to sort by the name, we get an error message.

Demo 05:

```
select distinct z_type
from zoo_2016
order by z_name desc;
order by z_name desc
*
ERROR at line 3:
ORA-01791: not a SELECTed expression
```

You will often find this to be the case. You try to run a query that seems to make sense at first and find that it is blocked by the system. Most of the time, if you reflect on this, you will see the reasoning behind the decision to disallow the action.

2. Using RowNum

RowNum is a pseudo column that can be used to number and limit the rows as they are returned to SQL*Plus. You can also think of RowNum as a function that returns a row number value. The use of RowNum is limited to Oracle databases.

The tests that work correctly are limited to testing if RowNum is < value or <= value. Think of RowNum as counting off the rows as they are passed into the result set until the row count reaches the desired number.

RowNum is applied to the rows **before** any Order By or Distinct. This means the results might not be what you had hoped to achieve.

Demo 06: These are the rows from the zoo_2016 table sorted by price with the most expensive first.

```
select z_id, z_name, z_cost
from zoo_2016
order by z_cost desc;
```

Z_ID	Z_NAME	Z_COST
85	Sally Robinson	5000.25
56	Leon	5000
57	Lenora	5000
259		5000
258		5000
23	Sam	5000
260	Geoff	5000
257	Arnold	5000
52	Dewey	3750
45	Louie	2500.25
44	Dewey	2500.25
43	Huey	2500.25
374		1850
375		1850
373		1850
372	Leon	1850
371	Anne	490.01
370	Anders	490
25	Abigail	490
47		490

20 rows selected.

Demo 07: Now we try to get the five most expensive animals using RowNum

```
select z_id, z_name, z_cost
from zoo_2016
where ROWNUM <=5
order by z_cost desc;
```

Z_ID	Z_NAME	Z_COST
85	Sally Robinson	5000.25
23	Sam	5000
56	Leon	5000
57	Lenora	5000
25	Abigail	490

5 rows selected.

It runs but these are not the most expensive animals- we have many animals at \$5000. That is not what we wanted. What RowNum does is take the first 5 rows that come from the zoo_2016 table and are passed to the

result set and then it sorts those 5 rows. We will eventually solve the problem of the five most expensive items. What you need to remember at this time is that **Rownum is a Where clause test and that is processed before the Order By clause.**

A few more examples might help.

Demo 08: First show the different animal types in the zoo_2016 table sorted. I got 6 rows

```
select DISTINCT z_type
from zoo_2016
order by z_type;
```

Z_TYPE

Armadillo
Giraffe
Horse
Lion
Zebra
armadillo

6 rows selected.

Demo 09: Then add a where rownum <= 2 filter; the output looks ok. I got two rows.

```
select DISTINCT z_type
from zoo_2016
where rownum <=2
order by z_type;
```

Z_TYPE

Armadillo
Giraffe

Demo 10: But If I change this to rownum <=8, I got only four rows. Why didn't I get the Horse or armadillo?

```
select DISTINCT z_type
from zoo_2016
where rownum <=8
order by z_type;
```

Z_TYPE

Armadillo
Giraffe
Lion
Zebra

4 rows selected.

With `Where rownum<=8` why do I get only four rows? The From clause and the Where clause were executed first and 8 rows were returned, but then Distinct was applied to those 8 rows and apparently there were only 4 distinct z_type values in those rows. (you might get a different set of rows.)

Demo 11: Including RowNum in the select may help you see what is happening. I removed the DISTINCT.

We get 8 rows coming from the table in whatever order the dbms delivers them. After the 8 rows are retrieved, the Order By clause is applied- here the sort is by the price. We do get rownum values 1, 2, 3, 4, 5,6,7, and 8.

```
select ROWNUM, z_id, z_cost, z_type
from zoo_2016
where ROWNUM <=8
order by z_cost desc;
```

ROWNUM	Z_ID	Z_COST	Z_TYPE
5	85	5000.25	Giraffe
3	56	5000	Lion
1	23	5000	Giraffe
4	57	5000	Lion
6	43	2500.25	Zebra
7	44	2500.25	Zebra
8	45	2500.25	Zebra
2	25	490	Armadillo

8 rows selected.

If I now manually looked at the rows and got the distinct values for z_type from that display, I get Giraffe, Lion, Zebra, Armadillo

Some other tests might help you see how this works. All of these return 0 rows. This is due to the way that Oracle generates the rownum values as rows are pulled from the table.

Demo 12:

```
select ROWNUM, z_id, z_name, z_cost
from zoo_2016
where rownum = 5;
```

Demo 13:

```
select ROWNUM, z_id, z_name, z_cost
from zoo_2016
where rownum between 5 and 10;
```

Demo 14:

```
select ROWNUM, z_id, z_name, z_cost
from zoo_2016
where rownum >= 5;
```

The tests that work as we probably want are limited to testing if RowNum is **equal to or less than a value**. RowNum is not the same as a Top or Limit keyword that you might be familiar with from another dbms.

One common use of RowNum is to limit the output to a few rows to see what type of data we have in a large table.

Table of Contents

1.	Data types and literals.....	1
1.1.	Character strings	1
1.2.	Numbers.....	2
1.3.	Temporal Data.....	2
1.4.	Literals.....	2
2.	Null	2
3.	Select from the DUAL table.....	3
4.	Manipulating numeric data.....	3
5.	Concatenation of strings	6
6.	Testing calculations with nulls	6
6.1.	The Coalesce function	7
7.	Functions we have seen	8

Data can be manipulated with operators, such as the concatenation operator for text and the arithmetic operators for numbers. Another way to manipulate data is with built-in functions which we will start looking at here. These manipulations result in new values being displayed in the query output or used in the Where clause or Order By clause. The queries shown here do not change the data in the tables.

Oracle will attempt to do automatic conversion of values from one data type to another. Relying on automatic conversion is risky. You should not assume that your data values are always clean or that the conversions work as you anticipate.

1. Data types and literals

There are several basic data types available for data to be stored in a table or to be used in expressions. The basic types are:

- String types
- Numeric types
- Date and time types

For our tables we will use only a few types of data.

1.1. Character strings

String types have two general types

- Char(99)
- Varchar2 (99)

CHAR is used for fixed length strings. We use CHAR (2) to store state abbreviations since they all have a 2 letter. If you are storing data where all of the data has the same number of characters- such as SSN, ISBN13, some product codes, then CHAR is appropriate. If necessary, the data will be end-padded with blanks to the stated length when stored.

VARCHAR2 is used for variable length strings. We could use VARCHAR2 (25) to store customer last names assuming we won't need to store a name longer than 25 characters.

For both of these - if you are defining a table column with char or varchar and try to insert a value longer than the defined length, you will get an error. Char defaults to a length of 1 if you do not state a length; varchar requires a length.

1.2. Numbers

Numeric types are divided into two categories

- Exact numbers
 - Numbers that store integers- like 15, 0, -35
 - Fixed point numbers which store numbers with a set number of digits- such as a number that can have a total of 6 digits and two of those digits are after the decimal. So that type could store a value such as 1234.56 or 0.3 but not a value such as 34.5678
- Approximate numbers
 - Floating point numbers that store values such as 5.876E2

The details of the data types is the type of stuff that you can look up in a book or manual when you need to know the range of each type. What you do need to know about numeric types is:

- Integer types are exact representation of numbers.
- Integers are faster for calculations
- Fixed decimal types may take more space and more processing time. A type such as decimal(6,2) can hold numbers from -9999.99 to +9999.99. The first value is the number of digits and the second the number of digits after the decimal point
- Floating point numbers are stored as approximation of the value; these vary with the overall size of the values they can store and the numbers of digits of accuracy. These types might be appropriate for storing a value such as a weight or distance.

1.3. Temporal Data

Date and Time data types vary more between the various dbms.

Oracle has a date type which stores a point in time as a date and time value.

1.4. Literals

Literals are constants- they have a specific value that does not change.

For string literals, enclose the data value within single quotes: 'San Francisco', 'cat'; if you need to include a quote within your literal, use two single quotes; 'San Francisco''s favorite cat';

For numeric literals, do not use delimiters; '97' is not a number- it is a string. Do not use commas or percent symbols etc. You can use a leading + or - symbol and a single decimal point. You can also use E notation.

These are valid numeric literals: 5 -58.89 + 0.002 2.3E5 (which is equal to 23000)
2.3E-3 (which is equal to 0.00 23)

For date literals you use the ANSI standard date expression: date '2009-05-14'. You can also use the Oracle traditional default format of to_date('06-JUL-09')

2. Null

If a column in a row has no value then we say the column is null. We use a null when we do not know the value or when a value would not be meaningful. A null is not the same thing as a value of 0 or a space. Oracle has a rule that it treat a zero length string as a null. A zero length string is a string that has no characters. For example in the vt_animals table we have a column an_type for the kind of animal this is; this is defined as Not Null. We expect that the vet can figure out what type of animal this is. We also have a column an_name; this is defined as null which means we could have an animal where we do not know the animals name. Maybe the animal doesn't have a name yet. I doubt the vet would refuse to treat a puppy because the client hasn't decided on a name yet.

Nulls will need special handling in various situation we will deal with over the semester. For this unit you need to know that nulls propagate in some expressions. For example, if any of the operands in an arithmetic expression is Null, then the calculated value is Null.

Whether or not you should allow nulls in database tables will always be an issue. Some people believe that you should never allow nulls in your tables. Certainly it is easier to work with tables that cannot accept nulls.

However you might have troubles if you always insist on non-null attributes. Do you really want to turn away sales because a customer says they have no first name? Are you positive that you can legally refuse to hire someone because they have no phone number just because your employee table has a not null phone number attribute?

You might make a good argument that the order tables should never have a missing quantity or price. But perhaps you are setting up an order and the price is negotiable- you might want to store the data that you have about the customer and the items to be ordered and then fill in the price later. Perhaps the customer does not know if they want to buy 4000 refrigerators or 4500 until they know what the negotiated price is.

3. Select from the DUAL table

Sometimes we want to just see how an expression works. We can use a Select statement with an expression but Oracle requires that every Select statement include a From clause with a table. We do not really want to see data from any table but we need to use a table to have a valid statement.

Oracle provides a special table named DUAL that consists of a single row and a single column. The column name is DUMMY and the value is 'X'.

You can use this to advantage in trying out calculations, functions, etc.

```
desc DUAL
```

Name	Null?	Type
DUMMY		VARCHAR2 (1)

Demo 01: Display the contents of the dual table

```
select *
from dual;
```

D
X

Demo 02: Display literals using the dual table. If you do not use a column alias, the expression will be used for the header.

```
select 'Hello world', 'Hello world' as Greeting, 2015 as Year, 'Paul'
from dual
;
```

'HELLOWORLD GREETING	YEAR 'PAU
-----	-----
Hello world Hello world	2015 Paul

4. Manipulating numeric data

Oracle uses the standard arithmetic operators: +, -, * and /. Arithmetic expressions can be used in the Select clause, the Where clause and the Order By clause.

Oracle follows the standard rules for arithmetic precedence. Multiplication and division are carried out before addition and subtraction. Use parentheses to change the order of evaluation.

With the expression: $5 + 3 * 8$ which has no parentheses, the multiplication is done first. $5 + 24 = 29$.

With the expression: $(5 + 3) * 8$ which has parentheses, the operation in parentheses is done, $8 * 8 = 64$.

Demo 03: Use the dual table for testing arithmetic with literals as arguments.

```
select
  5 * 3 as Col1
, 5 + 8   as Col2
, 5 + 3 * 8 as Col4
, (5 + 3) * 8 as Col5
, (5 + 3) / 3
from dual;
```

COL1	COL2	COL4	COL5	(5+3) / 3
15	13	29	64	2.66666667

If any of the operands in an arithmetic expression is Null, then the calculated value is Null. Nulls propagate in arithmetic.

For working with calculations, I have created a small test table. You can create test tables easily to try out ideas. I put these tables in the a_testbed database.

Demo 04: Create the Table; see the demo for the inserts.

```
create table z_tst_calc (
  item_id integer primary key
, quantity integer not null
, price   decimal (6,2)
);
```

ITEM_ID	QUANTITY	PRICE
101	1	125.12
102	5	30
103	10	101.05
104	1	75.5
105	12	33.95

Demo 05: Using a calculated column. Price * quantity gives a value commonly called the extended cost.

```
select item_id
, price
, quantity
, price * quantity as extendedcost
from z_tst_calc;
```

ITEM_ID	PRICE	QUANTITY	EXTENDED COST
101	125.12	1	125.12
102	30	5	150
103	101.05	10	1010.5
104	75.5	1	75.5
105	33.95	12	407.4

Demo 06: Another calculation. We repeat the calculations of quoted_price * quantity_ordered for the last column. You cannot use an alias as an operand in a calculation in the Select clause.

```
select item_id
, price
, quantity
, price * quantity as extendedcost
, price * quantity * 1.085 as ExtCostWithTax
from z_tst_calc;
```

ITEM_ID	PRICE	QUANTITY	EXTENDEDCOST	EXTCOSTWITHTAX
101	125.12	1	125.12	135.7552
102	30	5	150	162.75
103	101.05	10	1010.5	1096.3925
104	75.5	1	75.5	81.9175
105	33.95	12	407.4	442.029

Demo 07: You cannot meaningfully use the column alias in the Select to continue the calculations. Suppose you want to add a \$5 handling fee per item line.

The following does not run; you get an error message that ExtendedCost is an invalid identifier

```
select item_id
, price
, quantity
, price * quantity as extendedcost
, extendedcost + 5 as ExtCostWithShipping
from z_tst_calc;
```

Demo 08: We will get to functions soon; this uses the Round function. Here it rounds the expression to two digits after the decimal point.

```
select item_id
, price
, quantity
, price * quantity as extendedcost
, round(price * quantity* 1.085, 2) as ExtCostWithTax
from z_tst_calc;
```

ITEM_ID	PRICE	QUANTITY	EXTENDEDCOST	EXTCOSTWITHTAX
101	125.12	1	125.12	135.76
102	30	5	150	162.75
103	101.05	10	1010.5	1096.39
104	75.5	1	75.5	81.92
105	33.95	12	407.4	442.03

Demo 09: Using a calculated value as a sort key. We could also use the clause: Order by ExtCostWithTax

```
select item_id
, price
, quantity
, price * quantity as extendedcost
, round(price * quantity* 1.085,2) as ExtCostWithTax
from z_tst_calc
order by round(price * quantity* 1.085,2) ;
```

Demo 10: We are having a \$50.00 off sale. This might not be such a good idea! We have negative prices

```
select item_id
, price
, quantity
, price * quantity as extendedcost
, (price- 50) * quantity as saleccost
from z_tst_calc;
```

ITEM_ID	PRICE	QUANTITY	EXTENDEDCOST	SALECCOST
101	125.12	1	125.12	75.12
102	30	5	150	-100
103	101.05	10	1010.5	510.5
104	75.5	1	75.5	25.5
105	33.95	12	407.4	-192.6

5. Concatenation of strings

Concatenating strings means to put the strings together one after the other. You use the concatenation operator `||` to concatenate strings.

Concatenation does not add spaces between the strings being put together. Include spacing by concatenating in a literal space. `' ' || ' '`

Oracle will concatenate a null with a non-null string to give the string. This is different than some other dbms.

Demo 11: Concatenating strings. Oracle concatenates a null as if it were a zero-length string. With Oracle nulls do not propagate in concatenation. The last row displayed here does not have a value for the `an_name` column. The space at the start of the concatenated column value is the literal space being concatenated into the expression.

```
select an_name
, an_type
, an_name || ' is a ' || an_type as Animal
from vt_animals;
-- Selected rows
```

AN_NAME	AN_TYPE	ANIMAL
Burgess	dog	Burgess is a dog
Pinkie	lizard	Pinkie is a lizard
Pinkie	dog	Pinkie is a dog
Yoggie	hedgehog	Yoggie is a hedgehog
	bird	is a bird

Demo 12: You can concatenate values of different data types and Oracle will do type casts. Later we will see ways to improve the appearance of this output.

```
select srv_desc || ' $' || srv_list_price As "ServicePrice"
from vt_services
where rownum <= 5;
```

ServicePrice
Dental Cleaning-Canine \$50
Routine Exam-Canine \$80
Dental Cleaning-Other \$100
Feline PCR Series \$75
Dental Cleaning-Feline \$45

6. Testing calculations with nulls

What you need to know for now is how nulls are treated in calculations. We will set up a small test table to look at these calculations. We will use an id column, a nullable column that stores strings, a nullable column that stores integers, and a nullable column that stores floats.

Demo 13:

```
create table z_tst_nulls (
  col_id      int          not null primary key
, col_string  varchar2(10) null
, col_int     int          null
, col_float   float        null
);
```

We need very few rows to test this.

```
insert into z_tst_nulls values (1, 'abc', 10, 10.567);
insert into z_tst_nulls values (2, 'abc', null, 20.222);
insert into z_tst_nulls values (3, null, 30, null);
insert into z_tst_nulls values (4, null, null, null);
```

```
select *
from z_tst_nulls ;
```

COL_ID	COL_STRING	COL_INT	COL_FLOAT
1	abc	10	10.567
2	abc		20.222
3		30	
4			

Demo 14: Now a few expressions

```
select col_id
, 'XYZ' || col_string as stringTest
, 25 + col_int as intTest
, 25 + col_float as floatTest
from z_tst_nulls;
```

COL_ID	STRINGTEST	INTTEST	FLOATTEST
1	XYZabc	35	35.567
2	XYZabc		45.222
3	XYZ	55	
4	XYZ		

Comparing these two selects, we can see that nulls propagate in arithmetic but not in string concatenation.

6.1. The Coalesce function

Since nulls create problems for us, a dbms has a function to substitute another value for the null. This function is coalesce. For now we will use coalesce with two arguments- the first will be a column name and the second argument is a value to use if the column value is null.

Demo 15: Using Coalesce where both arguments are of the same data type

```
select col_id
, coalesce(col_string, 'DataMissing') as stringTest
, coalesce(col_int, -20) as intTest
, coalesce(col_float, 29.95) as floatTest
from z_tst_nulls;
```

COL_ID	STRINGTEST	INTTEST	FLOATTEST
1	abc	10	10.567
2	abc	-20	20.222
3	DataMissing	30	29.95
4	DataMissing	-20	29.95

Demo 16: Note that you have to return a numeric value for the numeric columns.

```
select col_id
, coalesce(col_string, 'DataMissing') as stringTest
, coalesce(col_int, 'DataMissing') as intTest
from z_tst_nulls;
```

```
, coalesce(col_int, 'DataMissing') as intTest
*
ERROR at line 3:
ORA-00932: inconsistent datatypes: expected NUMBER got CHAR
```

Part of the reason for doing this is to encourage you to create small tests to find out how your dbms works. It is generally quicker to set up a small test table like this than to page through a book or try an internet search.

One of the problems with internet searches is that many web pages do not tell you which version of the software they are using, which system settings are in place- and that does not even count the web pages that are just wrong. Some web pages don't even seem to say which dbms they are using. You certainly should be able to check some things out on web pages- but you still need to verify anything you find.

7. Functions we have seen

These are some functions I have used so far that seem easy to work with.

<code>extract(Month from a date value)</code>	-- get the month number
<code>extract(Year from a date value)</code>	-- get the year
<code>Round(number, position)</code>	-- rounds at the indicated position
<code>Coalesce(exp1, exp2, exp3)</code>	-- get the first not-null value

Examples

```
select extract(Month from date '1015-05-12')
, extract (year from date '1015-05-12')
from dual;
```

<code>EXTRACT(MONTHFROMDATE'1015-05-12')</code>	<code>EXTRACT(YEARFROMDATE'1015-05-12')</code>
-----	-----
5	1015

```
select Round(458.873), Round(458.873, 0), Round(458.873, 2), Round(458.87,-2)
from dual;
```

<code>ROUND(458.873)</code>	<code>ROUND(458.873,0)</code>	<code>ROUND(458.873,2)</code>	<code>ROUND(458.87,-2)</code>
-----	-----	-----	-----
459	459	458.87	500

```
select coalesce(345,34), coalesce(345,null)
, coalesce(null,34), coalesce(null, null)
from dual;
```

<code>COALESCE(345,34)</code>	<code>COALESCE(345,NULL)</code>	<code>COALESCE(NULL,34)</code>	C
-----	-----	-----	-----
345	345	34	-

```
select coalesce(null, null, 56, null)
from dual;
```

<code>COALESCE(NULL,NULL,56,NULL)</code>

56

Table of Contents

1.	Attributes and domains.....	1
2.	Relations.....	1
3.	Keys	2
3.1.	Candidate, Primary Keys	2
3.2.	Foreign Key.....	2
4.	Relationships	2
5.	Database and table design.....	3

This document discusses database terms and concepts from a design point of view. This is not a database design class (that is CS 159A); but we cannot talk about databases and tables without talking about design.

We need to talk about collections of tables/relations and the associations/relationships between these tables and part of this is terminology.

1. Attributes and domains

An attribute is a characteristic of an entity that we need to store; an attribute is represented as a column in a table. We will limit the values that can be stored in an attribute. The first way is simply the data type that we use when we define the table. In the vt_staff table, the stf_id is defined as an integer; so we can use a staff id of 1567, but we cannot use a staff id value of 45.67 nor can we store a staff id value of 'Sect'. Look at the vt_staff.stf_job_title attribute; it is limited to the four values that are listed in the definition. You can find other rules about the allowed data values as you read the Create Table statements. The term domain is sometimes used to describe the legitimate values for an attributes; some people use the term 'type' for this concept.

2. Relations

A **RELATION** is a collection of tuples (the theoretical term for what we commonly call a table row). A relation is represented as a two-dimensional table. A relation has the following characteristics:

- Each relation has a name that is unique within the schema. We cannot create two tables with the same name in the same schema.
- Each row contains information about a single entity instance. We should not have a row in a table that includes information about an client and also information about their animals. These are two separate entities and the data is stored in two tables.
- No two rows in a table are identical. It would not be helpful to have two rows in the client table for the same client. We use a **primary key** to avoid duplicate rows.
- Each column contains attribute data values. A column is defined with a name that represents an attribute we want to store and we should use that column for that purpose.
- All of the values in a column are from the same domain.
- Each column has a name that is unique within the relation. We cannot create two columns in the same table with the same name. We can use the same name for columns in different tables and we often do this for the columns that form the relationships between tables.
- A table has to have at least one column.
- Each cell contains a single data value. If we want to keep salary history for our staff, we would have to set up a new table for this. We should not try to keep salary history as a list of values in the salary attribute.
- The relation may not have repeating columns. Suppose we decided to keep track of our staff phone numbers and we wanted to store multiple phone numbers for each staff person. It would not be appropriate to create columns such as phone_1, phone_2, phone_3 etc.
- The order of the rows has no logical significance. We cannot have any logic in our table concepts that tries to reflect that a row in a table has some meaning because it follows another row. We can display

- the rows in a certain order, but we do not store the rows in any particular order. Tables reflect sets which are not ordered.
- The order of the attributes has no logical significance. We are not required to have the attribute for the primary key be the first column in the table, although it often is the first. In fact we cannot meaningfully speak of the first attribute in a relation. We speak of attribute by the name of the attribute. (Tables actually have an ordering to their columns and we do use that in SQL sometimes. An SQL table is not strictly speaking a relation.)

3. Keys

We talk a lot about keys in databases; there are several types of keys.

3.1. Candidate, Primary Keys

A candidate key is an attribute or a collection of attributes which is never null and is always unique within its table. In the vt_staff_pay table, we have two candidate keys, the stf_id and the stf_ssn. The primary key is the key that we chose as the identifier and define as such when we create the table. A table has only one primary key.

A composite key is made up of more than one attribute. The vt_exam_details table has a primary key made up of two attributes (ex_id and line_item.).

The Entity Integrity Rule says that each table must have a primary key, which cannot be null. If this is a composite primary key, then no component of the key can be null.

3.2. Foreign Key

Foreign keys are defined with respect to two tables. The foreign key is the attribute in the child table that has values matching a candidate key's values (normally the primary key) in the parent table. The foreign key must have the same underlying domain as the associated candidate key. The foreign key values must either be null or have values that match existing values in the parent table.

The foreign key and the referenced candidate key must have the same data type. A foreign key can be nullable.

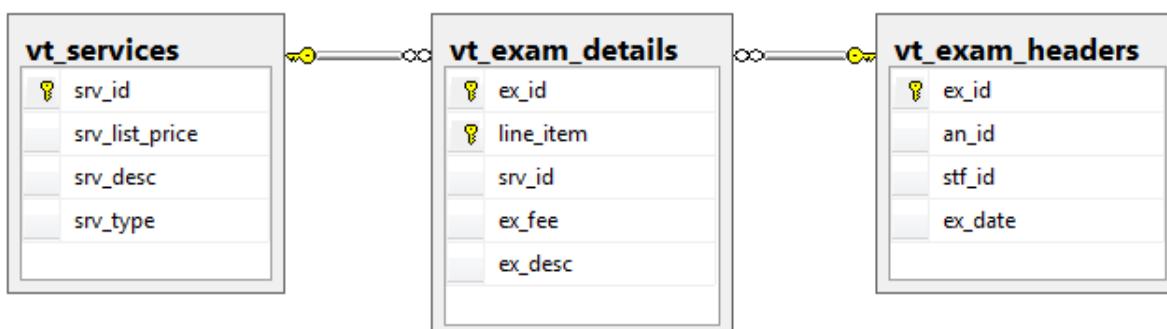
The foreign key and the associated candidate key can have different names.

A row in the parent table can have zero, one, or more associated rows in the child table.

It is possible that the parent table and the child table are actually the same table.

If the primary key of a table is composite, then any related foreign key must also be composite.

Take another look at the table vt_exam_details. The column ex_id refers to the ex_id in the vt_exam_headers table so that we know that this detail row belongs to a specific exam row. The table vt_exam_details also has a column srv_id that refers to the table vt_services; that way we know that the service we have for that detail row is a service the database knows about.



4. Relationships

A Relationship is an association between two or more relations based on data values in common between the tables. Relationships can be classified as One-to-One, One-to-Many, or Many-to-Many.

We have mentioned that we could have a table with data about clients and another table with data about animals. For our database, each client could have multiple animals. The relationship between the clients table and the animals table is One-to-Many. The term "many" does not mean that we must have multiple rows on the many side- only that we could. A one-to-many relationship allows a situation where we could have a row for a client and no animals for that client.

The terms parent and child are often used to describe the tables in a one to many (1:N) relationship. For example, one client may have many animals. In this case, the client table is the parent and the animals table is the child. One animal may be associated with multiple exams. In this relationship, the parent is the vt_animals table and the child is the vt_exam_header table.

We decided to put some the data about our staff in the vt_staff table and part into the vt_staff_pay table for privacy reasons. The relationship between these tables is One-to-One. Each staff person has at one row in the vt_staff table and one row in the vt_staff_pay.

We cannot create a Many-to-Many relationship in our database but logically we do have a Many-to-Many relationship between our entities. Each service that the vet can supply can be used on multiple exams and each exam can include multiple services. (The vt_exam_details table handles that relationship.)

5. Database and table design

Good database design depends on good table design. Good table design reduces redundancy and avoids errors in adding new rows, deleting current rows and updating existing data.

These are some general rules for designing good tables for a relational database. We will formalize these later, when we discuss normalization.

- Store each fact one time only. One purpose of a database is to control redundancy, so you do not want to store facts more than once.
- Each attribute should contain a single value. For example, if you are storing information about exams and an exam includes 10 services, you do not store all the service ids in a single attribute.
- Each attribute should be atomic. Do not have an attribute named Address that contains the street address, city, state, country, and postal code. This should be five separate attributes.
- Each row should have an attribute or combination of attributes that is a unique identifier- a primary key. Naturally occurring attributes, such as an animal name attribute, are not good primary keys because you cannot guarantee that they will be unique.
- The primary key should not be subject to change. For example, a phone number is generally not a good primary key since people change their phone numbers and the phone company can reassign phone numbers to other people.
- Every attribute in a row should contain information about the entity identified by the primary key. For example, a table with information about clients will store the client's name and address values — but not information on the animals that they have. That information will be stored in related tables.
- In general, every table in the database will have a relationship with at least one other table.
- Usually, a well-designed database will have many narrow tightly focused tables- rather than a few tables that try to store a lot of different columns of data.
- Database logic is based on set logic- a set is an unordered collection of items, all of the same type. A well designed table is a set of rows.
- SQL allows you to create tables which have duplicate rows; therefore SQL tables correspond to logical constructs called multi-sets or bags.
- A table is a set of rows; a row is a set of columns; a column is an atomic value which has a data type.
- With object-relational databases, the column value might be an object rather than a simple data value.

The discussion here has focused on the tables for the vets database. It would be a **Very Good Idea** if you think about these concepts in terms of the AltgeldMart database tables.

Table of Contents

1. Row filters	1
2. Testing for nulls.....	2
3. The IN list test	3
3.1. Simple In lists.....	3
3.2. In lists that contain nulls	5
3.3. In lists that contain row values	5
3.4. Testing a literal against an in list	6
4. The BETWEEN test.....	6
4.1. Gotchas with Between	8
5. Direct comparison operators	9
5.1. Tests for exact matches.....	9
5.2. Tests for non-matches.....	10
5.3. Tests for inequalities	10
6. Tests that require conversions.....	10

1. Row filters

Most of our queries so far have returned all of the data from the table in the From clause. We are working with very small tables. Imagine the output if we had thousands of rows in our tables. We seldom want to see all of the data in a table. Rather we want to see only a subset of the data- a subset that matches some search condition. If you are using the CCSF WebStars system to see your current schedule- you do not want to see everyone's schedule- just yours.

To filter the output, we add a WHERE clause to the SQL statement after the From clause. The Where clause specifies which rows should be returned. The Where clause contains a logical expression (a predicate or test) that is applied to each row in the table. If the logical expression evaluates to true for a row from the table (if that row passes the test), that row is returned in the result. The row is not returned if the predicate evaluates as False or as Unknown. We will cover a variety of comparison operators and conditional expressions in this unit and in the following units. We will also investigate the concept of Unknown/Null.

These are Row Filters; the test in the Where clause is applied to each row in turn.

Our query model is now

```
Select col_expressions
  From table_expression
 WHERE predicate
 ORDER BY sort_keys;
```

In this unit, we will look at several filters: we will discuss more filters soon.

- the Is Null and Is Not Null filters
- the In list filter
- the Between filter
- the comparison operators

A few general rules for creating filters

- You cannot test a column alias in a Where clause; you need to test using the column name.
- Character literals are enclosed in single quotes: 'CA', 'Anderson'
- Tests against character data are case sensitive in Oracle- so if I am looking for values that match 'CA', the values 'Ca' and 'ca' will not match

- In tests against character literals, leading and trailing blanks are significant. The value ' CA' or 'CA ' does not match the value 'CA'.
- Numbers are not enclosed in quotes; do not include punctuation such as commas or dollar signs when you write numeric literals
- Dates are more complex; for now write date literals in the standard Oracle default format: '30-JUN-2002'. Note that this is a string literal; Oracle will cast it to a date for date testing. You can also use the date expression: date '2002-06-30'

2. Testing for nulls

You may want to write your queries to skip any rows where certain column values are null. The way to test for this is to add a Where clause after the From clause that filters for the nulls.

Testing for missing values requires the use of the IS NULL operator. Think of IS NULL and IS NOT NULL as single operators. You use the same operator IS NULL for any data type that you are testing.

If you try to test using the syntax = NULL, you will get no rows returned. This is not flagged as an error by the dbms but it does not filter for nulls.

Oracle currently treats a character value with a length of zero as null. This would be a string entry entered as " - a pair of delimiters with nothing between. However, this has not always been true for all Oracle products. You may have to test this on whatever applications you are running.

Demo 01: Test for empty attributes by using the test IS NULL. You use the same test for attributes of any data type. This tests for a null order_shipping_mode_id which is a char(6) attribute.

```
select order_id, order_date, sales_rep_id, shipping_mode_id
from oe_orderHeaders
WHERE shipping_mode_id IS NULL;

```

ORDER_ID	ORDER_DAT	SALES REP_ID	SHIPP
550	02-AUG-15		
551	03-AUG-15		
116	12-NOV-15	155	
117	28-NOV-15	150	
118	28-NOV-15	150	
119	28-NOV-15	155	
2120	02-JAN-16		
2121	03-JAN-16		

Demo 02: Test for empty attributes by using the test IS NULL. This is testing an integer column.

```
select order_id, order_date, sales_rep_id
from oe_orderHeaders
WHERE sales_rep_id IS NULL;

```

ORDER_ID	ORDER_DAT	SALES REP_ID
550	02-AUG-15	
551	03-AUG-15	
115	08-NOV-15	
2120	02-JAN-16	
2121	03-JAN-16	
2225	09-MAR-16	
3227	01-MAR-16	
525	09-MAY-16	
527	01-MAY-16	

Demo 03: We can use IS NOT NULL to find rows that have a data value.

```
select order_id, order_date, shipping_mode_id
from oe_orderHeaders
WHERE shipping mode id IS NOT NULL;
```

ORDER_ID	ORDER_DATE	SHIPPING_MODE_ID
378	14-JUN-15	USPS1
411	01-AUG-15	FEDEX1
412	01-AUG-15	UPSGR
413	07-AUG-15	USPS1
414	20-AUG-15	UPSEXP
415	23-AUG-15	UPSEXP
552	12-AUG-15	FEDEX1
605	05-SEP-15	UPSGR

Demo 04: Note what happens if you use the test = null instead of Is null

```
select order_id, order_date, shipping_mode_id
from oe_orderHeaders
WHERE shipping mode id = NULL;
no rows selected
```

The null indicator does not equal any value in the table. So testing with = Null or <> Null is not a true test and that filter will return no rows. The use of Nulls in tables is important but null testing is not always obvious.

3. The IN list test

Suppose we want to display all customers named Morris or Morse or Morise. This is testing against a specific set of values and we can use an IN list for this. The list of values is enclosed in parentheses and the values are separated by commas.

3.1. Simple In lists

Demo 05: Using the IN list for text values. Each text value is enclosed in quotes.

```
select customer_id
, customer_name_last
, customer_name_first
from cust_customers
where customer_name_last in( 'Morise', 'Morris', 'Morse' )
;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
401250	Morse	Samuel
402100	Morise	William
404950	Morris	William
409010	Morris	William
408777	Morise	Morris

5 rows selected.

Demo 06: Using the IN list for numeric values. It is not an error to have a value in the list which does not match any rows in the table. It is not an error to have a in the list appear more than once.

```
select order_id
, order_date
, customer_id
from oe_orderHeaders
where order_id IN ( 107, 95, 125, 107, 445 );

```

ORDER_ID	ORDER_DATE	CUSTOMER_ID
107	10-JUN-15	401250
95	14-JUN-15	401250
125	14-JUN-15	401250
107	14-JUN-15	401250
445	14-JUN-15	401250

107 02-OCT-15	403050
125 09-DEC-15	409160

Demo 07: You can use the NOT IN test to exclude specified data values.

```
select prod_id, prod_name, catg_id
from prd_products
where catg_id NOT IN ('HW', 'PET')
;
```

PROD_ID	PROD_NAME	CATG_I
1010	Weights	SPG
1020	Dartboard	SPG
1030	Basketball	SPG
1040	Treadmill	SPG
1050	Stationary bike	SPG
1060	Mountain bike	SPG
4569	Mini Dryer	APL
1120	Washer	APL
. . .	selected rows	

Demo 08: You can use have a list that contains only one item.

```
select job_id, job_title, max_salary
from emp_jobs
WHERE max_salary IN (120000);
```

JOB_ID	JOB_TITLE	MAX_SALARY
16	Programmer	120000

Demo 09: You can use a NOT IN list that contains only one item.

```
select job_id, job_title, max_salary
from emp_jobs
WHERE max_salary NOT IN (120000);
```

JOB_ID	JOB_TITLE	MAX_SALARY
1	President	100000
2	Marketing	75000
4	Sales Manager	60000
8	Sales Rep	30000

The above two queries may look like they should return all rows in one or the other of these queries. But we have eight rows in the jobs table; one was returned with the IN (12000) test and four with the NOT IN (12000) test. What happened to the other three rows? Neither the IN test nor the NOT IN test return the rows where the max_salary is null. For that you need to test with the IS NULL test.

Demo 10: Test with Is Null

```
select job_id, job_title, max_salary
from emp_jobs
where max_salary is null;
```

JOB_ID	JOB_TITLE	MAX_SALARY
32	Code Debugger	
64	DBA	
128	RD	

3.2. In lists that contain nulls

If you try putting a Null in a list you will find that it does not match a row with a null. The rule is that a row is returned if the Where predicate evaluates as True; a null in the table does not match a null in the list. A null value does not match another null value. The logical value of a null matching a null is Unknown; the value of the filter expression must be True for the row to be returned.

Demo 11: Trying to use Null in a list.

```
select job_id, job_title, max_salary
from emp_jobs
where max_salary IN (120000, null );

```

JOB_ID	JOB_TITLE	MAX_SALARY
16	Programmer	120000

1 row selected.

What may seem more surprising is the following query, which returns no rows. The rule is that a row is returned if the where predicate evaluates as True; the row is not returned if the predicate evaluates as False or as Unknown. Here we are negating the Unknown value which is still unknown.

Demo 12: Nulls always get interesting

```
select job_id, job_title, max_salary
from emp_jobs
where max_salary NOT IN (120000, null );

```

no rows selected

3.3. In lists that contain row values

You can test constructed rows with an In test. In Oracle you can test row expressions that refer to a two part value. The row value is enclosed in parentheses. The row values are enclosed in parentheses for the In list. The two columns we are comparing are also enclosed in parentheses.

Demo 13: Suppose we wanted to find employees in dept 30 with manager 101. We could use the following. The row values is (30, 101) and it is enclosed in parentheses for the In list. The two columns we are comparing are also enclosed in parentheses.

```
select emp_id, name_last, dept_id, emp_mng
from emp_employees
where (dept_id, emp_mng) IN ( (30, 101) );

```

EMP_ID	NAME_LAST	DEPT_ID	EMP_MNG
108	Green	30	101
205	Higgs	30	101
203	Mays	30	101

Demo 14: Now suppose we wanted to find employees in dept 30 with manager 101 and also employees in dept 35 with manager 101. We have one IN list that contains two row values.

```
select emp_id
, name_last
, dept_id
, emp_mng
from emp_employees
where (dept_id, emp_mng) in( (30, 101), (35, 101) );

```

EMP_ID	NAME_LAST	DEPT_ID	EMP_MNG
--------	-----------	---------	---------

108	Green	30	101
205	Higgs	30	101
162	Holme	35	101
200	Whale	35	101
203	Mays	30	101

3.4. Testing a literal against an in list

Commonly we think of testing a column against an In list of literals. But with some tests we can reverse that type of thinking. Suppose we are looking for a customer with the name Morise, but we do not know if that is the customer's first or last name. That happens fairly frequently when people fill out forms.

Demo 15: This query looks for the literal Morise in two columns

```
select *
from cust_customers
where 'Morise' in (customer_name_first, customer_name_last);
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST	CREDIT_LIMIT
402100	Morise	William	750
408777	Morise	Morris	7500

This syntax is not as obvious in meaning as testing a column against a list of values. Avoid this for a simple In list. The following is poor style.

```
select job_id, job_title, max_salary
from emp_jobs
where 120000 IN (max_salary);
```

4. The BETWEEN test

To test data against a range of values, use BETWEEN. The Between test is an **inclusive** test. If the row being tested matches an end point of the range, the test has a true value, and the row will get into the output display. The range should be an increasing range. If you test using a descending range WHERE salary BETWEEN 72000 and 3000, the query will run but no rows will be returned.

Demo 16: Using BETWEEN with a numeric range.

```
select emp_id
, name_last AS "Employee"
, salary
from emp_employees
where salary BETWEEN 65000 AND 70000
order by salary;
```

EMP_ID	Employee	SALARY
200	Whale	65000
104	Ernst	65000
160	Dorna	65000
109	Fiet	65000
103	Hunol	69000

5 rows selected

Demo 17: Using NOT BETWEEN to exclude values in the range.

```
select emp_id
, name_last AS "Employee"
, salary
from emp_employees
where salary NOT BETWEEN 10000 AND 65000
order by salary;
```

EMP_ID	Employee	SALARY
103	Hunol	69000
205	Higgs	75000
206	Geitz	88954
146	Partne	88954
162	Holme	98000
101	Koch	98005
204	King	99090
100	King	100000
161	Dewal	120000

Demo 18: Using BETWEEN with character range.

```
select emp_id
, name_last AS "Employee"
from emp_employees
where name_last BETWEEN 'J' and 'T'
order by name last;
```

EMP_ID	Employee
100	King
204	King
101	Koch
203	Mays
146	Partne
145	Russ
207	Russ

You need to be careful with character range tests. If we had an employee with a last name composed of just the letter T, that employee would be returned. But the employee with the name Tuck is not returned.

Demo 19: Using BETWEEN with character range. This returns no rows since Oracle is case specific and the current set of employees names begin with uppercase letters.

```
select emp_id, name_last AS "Employee", dept_id
from emp_employees
where name_last BETWEEN 'a' and 'zzzz'
order by name_last
;
```

no rows selected

Demo 20: Customers with a low credit rating

```
select customer_id
, customer_name_last, customer_name_first
, customer_credit_limit
from cust_customers
where customer credit limit between 0 and 1000;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
CREDIT_LIMIT		

401250 Morse	Samuel	750
402100 Morise	William	750
402110 Coltrane	John	750
402120 McCoy	Tyner	750
400801 Washington	Geo	750

Demo 21: But customers with no credit rating were not returned by the previous query.

```
select customer_id
, customer_name_last as "LastName", customer_name_first as "FirstName"
, customer_credit_limit as "CreditLimit"
from cust_customers
where customer_credit_limit is null
;
```

CUSTOMER_ID	LastName	FirstName	CreditLimit
402500	Jones	Elton John	
405000	Day	David	

The word "between" is one of those words that can be ambiguous in English, If I ask you how many animals appear between the Cat and the Dog, you will probably say 3.



But if I asked you how many integers there are between 12 and 16 { 12, 13, 14, 15, 16} some people will say 3 (13,14,15) , some will say 5 [12, 13, 14, 15, 16], and a few people will say 4 [12, 13, 14, 15) or (13, 14, 15, 16]. So if someone asks you to write a query that finds values between two points, it is a good idea to ask if they want to include the end points.

The SQL between operator is always inclusive of the end points. Your job is to write a query that does the job you were asked to do.

4.1. Gotchas with Between

Some tests to watch out for with the use of Between.

The Between operator will match no rows if the range is descending or if one of the range points is a null.

Demo 22: Range is decreasing

```
select emp_id, name_last AS "Employee", salary
from emp_employees
where salary BETWEEN 70000 AND 65000;
no rows selected
```

Demo 23: One end of the range is a null

```
select emp_id, name_last AS "Employee", salary
from emp_employees
where salary BETWEEN 30000 AND null;
no rows selected
```

Demo 24: One end of the range is a null

```
select emp_id, name_last AS "Employee", salary
from emp_employees
where salary BETWEEN null AND 51000;
no rows selected
```

5. Direct comparison operators

In this section we will look at another major row filtering technique, using the direct comparison operators.

These operators compare two expressions. The SQL comparison operators are;

= > >= < <= != or <>

The two expressions can be of the same type- such as comparing two string values or two integer values. The two expressions can be of types that can be cast to the same type- such as comparing an integer number to a float number. You need to avoid trying to compare two expressions of different types- such as comparing a date value to an integer. In some cases the dbms will attempt to do such comparisons but it is not a good idea.

These demos use the altgeld_mart tables.

5.1. Tests for exact matches

Demo 25: Display only rows with an exact match on Salary.

```
select emp_id
, name_last as "Employee"
, salary
from emp_employees
where salary = 20000;

```

EMP_ID	Employee	SALARY
150	Tuck	20000

Demo 26: Some queries do not return any rows. This does not mean the query is incorrect. We just do not have any matching rows.

```
select emp_id
, name_last as "Employee"
, salary
from emp_employees
where salary = 18888;

```

No rows selected

Demo 27: Display only location rows with a country-id of US.

```
select loc_city
, loc_street_address
from emp_locations
where loc_country_id ='US';

```

LOC_CITY	LOC_STREET_ADDRESS
Southlake	2014 Jabberwocky Rd
South San Francisco	2011 Interiors Blvd
San Francisco	50 Pacific Ave

Demo 28: Oracle is case specific on text comparisons.

```
select loc_city
, loc_street_address
from emp_locations
where loc_city ='SAN FRANCISCO';

```

No rows selected

5.2. Tests for non-matches

Demo 29: Use the not equals operator to exclude rows. You can use != or <>

```
select loc_city
, loc_street_address
from emp_locations
where loc_country_id !='US'
;
```

LOC_CITY	LOC_STREET_ADDRESS
Toronto	147 Spadina Ave
Munich	Schwanthalerstr. 7031
Mexico City	Mariano Escobedo 9991

5.3. Tests for inequalities

Demo 30: Finding jobs with a max salary less than \$60,000. Do not include formatting characters- such as the \$ or the comma in the literal.

```
select job_id, max_salary
, job_title
from emp_Jobs
where max_salary <60000
;
```

JOB_ID	MAX_SALARY	JOB_TITLE
8	30000	Sales Rep

Demo 31: Finding jobs with a max salary greater than or equal to 60000.

```
select job_id, max_salary
, job_title
from emp_Jobs
where max_salary >= 60000;
```

JOB_ID	MAX_SALARY	JOB_TITLE
1	100000	President
2	75000	Marketing
4	60000	Sales Manager
16	120000	Programmer

4 rows selected.

6. Tests that require conversions

These are queries that you could try to run that might not work at all in some dbms; that might work with invalid conversions; or that might turn out OK. In any case you should not run these types of queries- care about your data!

Implicit Type casting: Suppose you wrote the Where clause in the first demo as Where salary = '20000'

First of all, you should not do that. The salary attribute is defined as a numeric column and the literal '20000' is a string, not a number. Comparing a numeric attribute to a string is very poor style and makes you look like you

do not know how to write code. Most dbms will look at that expression and implicitly cast the string '20000' to a number to do the comparison. But you would need to know all of the cast rules for whatever dbms you are working with and you might occasionally be surprised at the cast that is done. An "implicit" cast is one that is done for you without the system informing you of the cast. So the solution is that you should write the literals correctly and avoid implicit casts when possible.

Demo 32: Finding employees with a salary of 20000. You should test the numeric salary attribute against a number- not against a string. **You should *not* do this type of test.**

```
select emp_id
, name_last as "Employee"
, salary
from emp_employees
where salary = '20000';

```

EMP_ID	Employee	SALARY
150	Tuck	20000

Table of Contents

1. Column Alias.....	1
2. Table Alias/Correlation Name/ Tuple Variable/Range Variable	1

There are two places in a Select statement where you might create something referred to as an alias. One is a column alias that you could use in the Select clause. The other is a table alias.

1. Column Alias

One use for column aliases is to make the output easier to understand.

This query does not use any column aliases; With Oracle if you have an expression that does not have a column alias, then the expression is used instead.

```
select order_id, prod_id, quoted_price, quantity_ordered
, quoted_price * quantity_ordered
from oe_orderDetails;
```

ORD_ID	PROD_ID	QUOTED_PRICE	QUANTITY_ORDERED	QUOTED_PRICE*QUANTITY_ORDERED
400	5002	23	5	115
400	5004	15	5	75

Now we have an alias for several of the columns; the column alias is used as the column header; it can be used in the Order By clause- but not in the Where clause.

```
select order_id, prod_id
, quoted_price AS PRICE
, quantity_ordered AS QUANTITY
, quoted_price * quantity_ordered AS ExtCost
from oe_orderDetails;
```

ORD_ID	PROD_ID	PRICE	QUANTITY	EXTCOST
400	5002	23	5	115
400	5004	15	5	75

Note that Oracle will let you use the same column alias for more than one column in the result. That is generally not advisable since the column alias should identify the column.

2. Table Alias/Correlation Name/ Tuple Variable/Range Variable

We can also designate an alternate name for a table. This is most commonly called a table alias; Oracle often uses the term correlation names; you may also see this called a tuple variable or a range variable. I'll generally use the term table alias since it is the most commonly used.

The table alias is defined in the From clause of the SQL statement and is limited in scope to that statement. The table alias is not saved on the server. Once you establish a table alias in a query, you need to use that alias, not the table name, in the other clauses of that query. You are not allowed to use the same table alias for two different tables in the query.

The use of table aliases is very common in SQL and you need to be aware of its use. However, poorly defined table aliases can make a query harder to read and understand.

The table alias is commonly a single letter but a single letter alias is often difficult to understand and remember. You should not have to keep referring to the From clause to interpret the Select clause. Table aliases can be longer than one character to make them more meaningful.

Some types of queries require the use of table aliases since the same table is included in the query more than once. In other queries, it is optional.

Each of the following queries will run. Each of these queries uses a single table and does not require the use of qualification of the columns in the Select or Where clause.

```
select order_id  
, prod_id  
, quoted_price  
from oe_orderDetails  
where quoted_price = 150.00;
```

This query qualifies each column name and uses the full table name; this is rather long and harder to read.

```
select oe_orderDetails.order_id  
, oe_orderDetails.prod_id  
, oe_orderDetails.quoted_price  
from oe_orderDetails  
where oe_orderDetails.quoted_price = 150.00;
```

This query qualifies each column name and uses a table alias; this is easier to read.

```
select od.order_id  
, od.prod_id  
, od.quoted_price  
from oe_orderDetails od  
where od.quoted_price = 150.00;
```

In this statement we use the token "od" for two different purposes. In the From clause, it identifies the table and stands for the set of rows that make up that table at the present time. In the Select and the Where clauses, "od" refers to a row at a time.

When we say

```
where od.quoted_price = 150.00
```

we are not testing that the entire table set of rows has a value for quoted_price of 150.00, or that at least one of the rows has that value; we are testing for each row- one row at a time- if that row has the value 150.00 for the quoted_price column in that row.

Likewise in the Select clause, the expression od.order_id stands for the order_id column of each row- one row at a time.

(This understanding of the token 'od' is the basis for calling this a tuple variable or a range variable.)

Table of Contents

1.	Why would you want to bother with this?	1
2.	Download and install SQL Developer	3
3.	Creating a connection	3
3.1.	Connecting to your CCSF Oracle Account	3
3.2.	Disconnect.....	4
4.	The Oracle SQL Developer interface	4
4.1.	Connect	4
4.2.	See a table as a description or data	4
5.	Running a query	5
6.	More on creating code/sql.....	5
7.	Creating a script	6
8.	SQL*Plus commands that do not work in SQL Worksheet	6
9.	Preferences	6
10.	Line and Column numbers	6
11.	dbForge for Oracle	6

Although you could do all of your work for this class using SQL*Plus as the client interface, you should gain some experience with a graphical interface client. SQL Developer is a free Java based graphical interface that Oracle provides (in Windows and Mac and Linux versions). You download it from the Oracle Tech website. You can use this client to connect to your hills CCSF Oracle account or to other Oracle installations. This document discusses:

- How to download and setup SQL Developer
- Creating a connection to your CCSF Oracle account
- What the interface looks like and how to use it

One problem that the use of SQL Developer might cause is that you get used to using the autocomplete and other features and have trouble writing queries on the midterm and final exam. Remember these exams are Paper and Pencil exams- and a piece of paper does not do autocomplete!

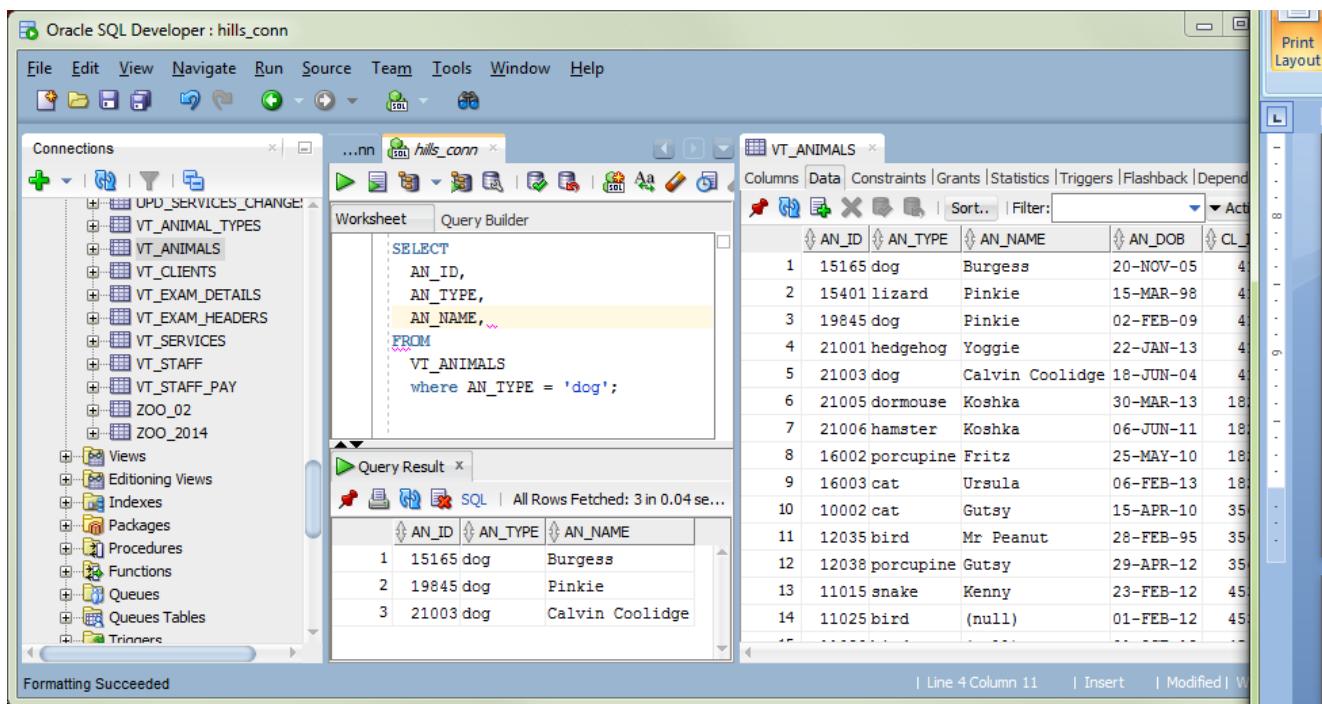
A critical thing to remember about this- and any other software tool- is that it is just a way to create and execute sql statements. You do not have to learn everything about a software tool to be more productive. It takes me 2-3 weeks to get comfortable using a new tool- so don't expect to like this immediately. If you are more comfortable using SQL*Plus, that is also OK. You can use SQL Developer for the things it is good at (such as editing in the code window) and SQL*Plus for the things that it is good at (running assignment scripts).

Since SQL*Plus and SQL Developer are different clients, you may find some places where the output looks different in terms of formatting. For example, SQL*Plus right aligns numbers in the column and SQL Developer left aligns.

Something that causes some people a lot of trouble with assignments. SQL Developer allows blank lines within a query. SQL*Plus does not. So do not include any blank lines within a query or your assignment script will be full of errors.

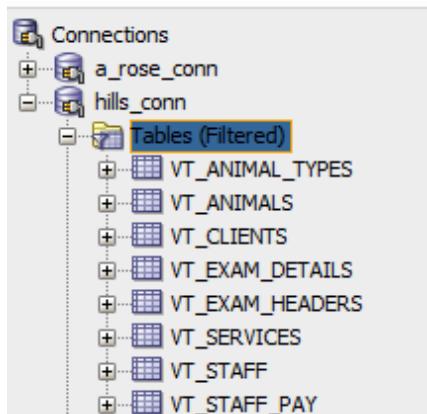
1. Why would you want to bother with this?

Because there is a very good chance that this will make you more efficient. You can edit the code window in place and rerun the command. You can highlight one command in the code window and run it by itself. You can see your table attributes without having to give another command. You can multiple windows open and switch between them. Did I mention that it is free?



The above image is a screen shot of SQL Developer (version 4.0.2). There is too much going on to explain everything at once- but what we have includes:

- A menu bar. I don't use the menus too often, but you probably want to look at the Tools menu at least.
- A tool button bar. You may want to use the Save button to save your sql statements. Or use the SQL worksheet button (near the right end) to open another window for your connection.
- You can open multiple tabs with different queries or table info displays in each.
The sql in the left window was run with the "Run Statement" button (the green arrow) and the output is displayed in a grid. You can use the "Run Script" button- the green arrow imposed on a text document- and the output from all of the queries in that window is displayed in the script output pane. You can highlight part of the sql and run just that part.
- The above screenshot shows a browser window. It shows your connections and a connection can be expanded to shows the tables and other objects in that schema; tables can be expanded to show the column names.
- You can also filter the table list using a wildcard pattern. I wanted to see only the tables for the vets database, and all of those table names start with vt_- right click the Tables node and use Apply Filter



2. Download and install SQL Developer

Download the zipped file from Oracle Tech.

http://www.oracle.com/technology/products/database/sql_developer/index.html

It is easiest to just take the version that includes Java although that may give you another Java installation.

Store the zipped file in a folder (not in a folder that holds any other Oracle files- this only applies to people with a local Oracle installation.). Unzip the file. It will unzip into a folder and you will have an executable file sqldeveloper.exe .

This is not a "regular" installation in the common sense. The files and folders are simply unzipped.

To run SQL Developer, simply double-click the file sqldeveloper.exe. You will probably want to set a short cut to this.

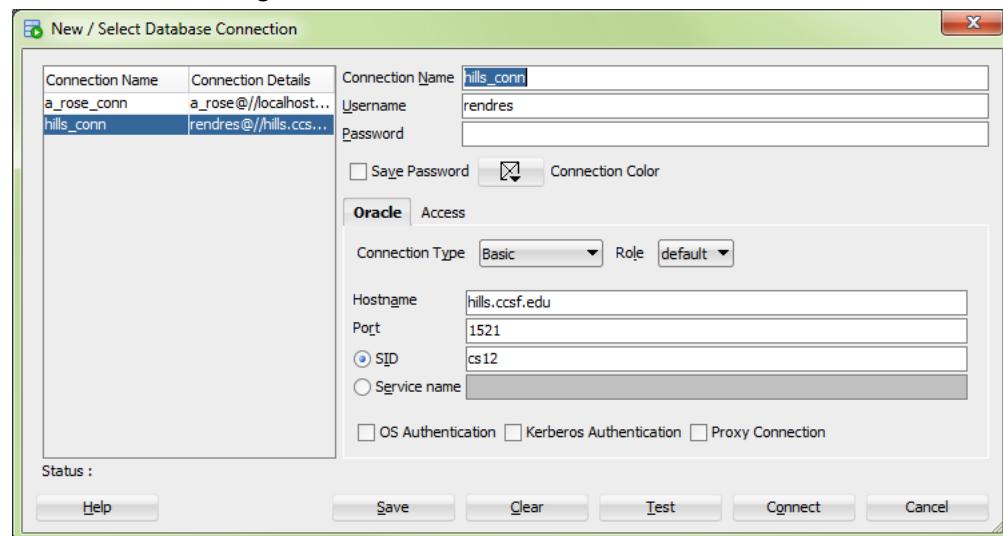
3. Creating a connection

3.1. Connecting to your CCSF Oracle Account

Right-click the Connections node and select New Connection. You will get a dialog box. I have two connections already established.



The connection dialog is shown here.



Enter a connection name in the text area for Connection Name. You can name this anything you want- hills_conn makes sense to me. Enter a valid user name and password in the next two text areas. You can decide if you want the system to save your password in this client. This would be a bad idea if this is on a machine that other people can use. Note that this is your Oracle password- not your hills password. You need to put your password in the first time you connect. Complete the bottom of the form. Leave the Role as Default and the Connection Type as Basic.

The host name is hills.ccsf.edu ; the default port number is 1521 and the SID is cs12 (that is a lowercase cs followed by the number twelve.)

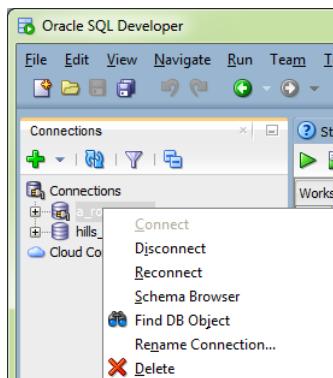
Click the test button and you should get a Status: Success! message in the lower left corner of the dialog.

If you want, you can now erase the password and the user will have to supply the password when they login.

(If you have a local installation of Oracle xe, you might use Connection type as Local/Bequeath)

3.2. Disconnect

Disconnect by right-click your connection and choosing Disconnect.

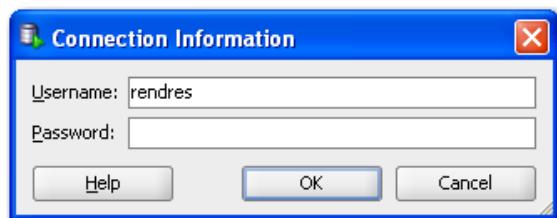


4. The Oracle SQL Developer interface

This interface is crowded but fairly simple if you have experience in using a graphical IDE.

4.1. Connect

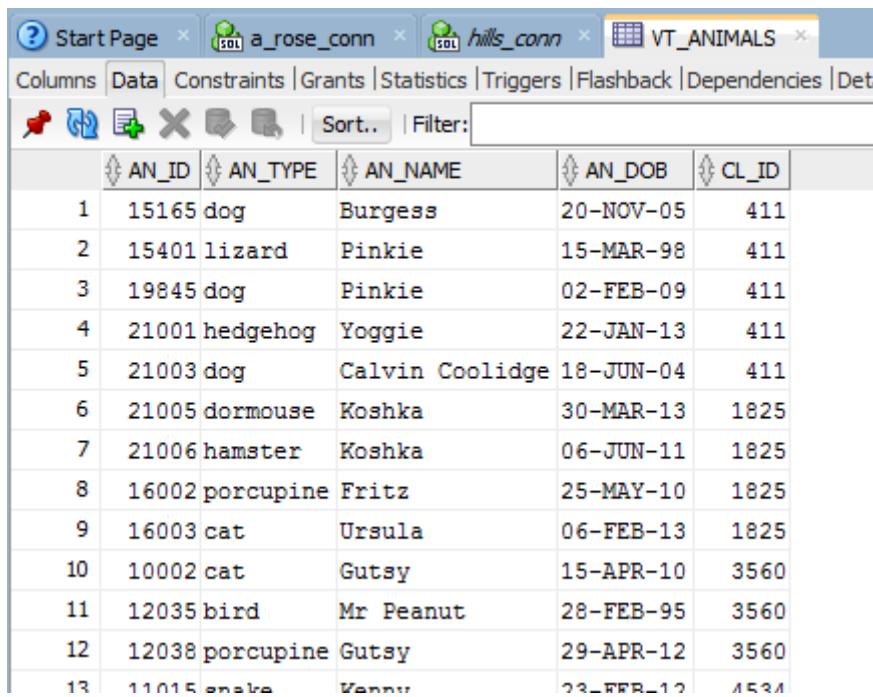
We have a Connection window which shows established connections to schema. If we right click a schema node, and select connect we can be prompted for a username and password so that we can login. We can then navigate to the node for tables and see our table names and then columns



4.2. See a table as a description or data

If you double-click a table name, you can see its description using the Columns tab and the data in the Data tab.

COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1 AN_ID	NUMBER (6,0)	No	(null)	1	(null)
2 AN_TYPE	VARCHAR2 (25 BYTE)	No	(null)	2	(null)
3 AN_NAME	VARCHAR2 (25 BYTE)	Yes	(null)	3	(null)
4 AN_DOB	DATE	No	(null)	4	(null)
5 CL_ID	NUMBER (6,0)	No	(null)	5	(null)



The screenshot shows the Oracle SQL Developer interface with the VT_ANIMALS table selected in the Data tab. The table has columns: AN_ID, AN_TYPE, AN_NAME, AN_DOB, and CL_ID. The data includes 13 rows of animal records.

	AN_ID	AN_TYPE	AN_NAME	AN_DOB	CL_ID
1	15165	dog	Burgess	20-NOV-05	411
2	15401	lizard	Pinkie	15-MAR-98	411
3	19845	dog	Pinkie	02-FEB-09	411
4	21001	hedgehog	Yoggie	22-JAN-13	411
5	21003	dog	Calvin Coolidge	18-JUN-04	411
6	21005	dormouse	Koshka	30-MAR-13	1825
7	21006	hamster	Koshka	06-JUN-11	1825
8	16002	porcupine	Fritz	25-MAY-10	1825
9	16003	cat	Ursula	06-FEB-13	1825
10	10002	cat	Gutsy	15-APR-10	3560
11	12035	bird	Mr Peanut	28-FEB-95	3560
12	12038	porcupine	Gutsy	29-APR-12	3560
13	11015	snake	Venom	23-FEB-12	4534

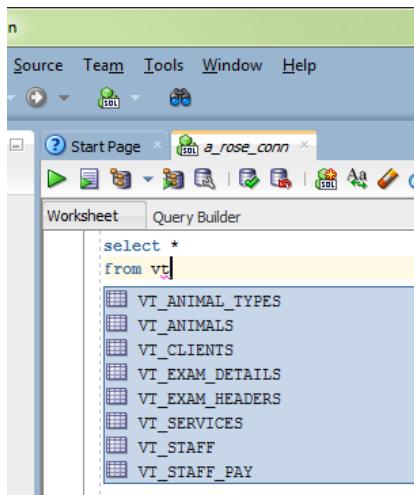
5. Running a query

You want to get to a Worksheet window. You can use the Tools menu or the dropdown menu for the SQL tool button.

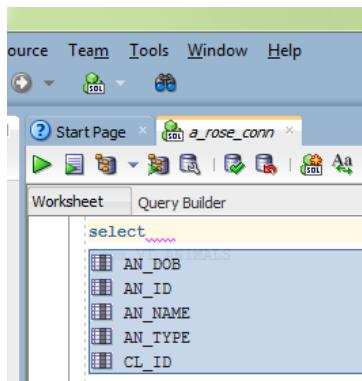
6. More on creating code/sql

This shows an autocomplete feature (Code Insight) You may also want to look at the following web page
<http://www.thatjeffsmith.com/archive/2014/05/video-completion-insight-in-oracle-sql-developer-v4-0-2/>

In the Worksheet window, I type "select * from vt" and wait a moment and the names of my tables that start with vt appear in a pop up box and I can select the table I want.



Then I go back up and put my cursor after the word select and type a blank; I get a pick list of the attribute names, I can select one, type a comma and I get the pick list again.



You can toggle individual lines or selected groups of lines as comments with Control /

7. Creating a script

What you may want to do is build and test the script file in SQL Developer and then do the final run using SQL*Plus. You ftp your script to your hills account as needed and ftp your spool file to your local computer.

8. SQL*Plus commands that do not work in SQL Worksheet

Some of the SQL*Plus commands you might want to use do not work in SQL Worksheet. The most annoying commands missing are the column command for formatting output. You may get a message about skipped commands.

9. Preferences

From the Tools menu you can get to Preferences. There are a lot of choices and you do not need to set them all. The Help for this is not particularly helpful.

You might want to change the display colors or font size. I find that making comments stand out is helpful. You can save your style settings.

10. Line and Column numbers

If you get an error message when you try to run a query, the message usually refers to a line and column. The line numbers can be displayed in the query window. The status bar at the bottom of the SQL Developer window displays the line and Column number of the position of the cursor in the query (code) window. As you move the cursor to a different position the status bar updates. This can help you find the problem with Error at Command Line:15 Column 43

11. dbForge for Oracle

dbForge has a free version of its Oracle client and also paid versions. It includes an Object browser, editing abilities, tabbed windows, code completion. Its interface is similar to that of Visual Studio.

<http://www.devart.com/dbforge/oracle/studio/>

The Einstellung effect refers to a person's predisposition to solve a given problem in a specific manner even though "better" or more appropriate methods of solving the problem exist. The Einstellung effect is the negative effect of previous experience when solving new problems. (http://en.wikipedia.org/wiki/Einstellung_effect)

How is that related to this class? Sometimes we have people in the class who have studied traditional programming techniques and know how to solve certain types of logic problems. For example, suppose we want to find animals that are either cats or dogs. A traditional programming approach would be a test that looked like: `an_type = 'cat' or an_type = 'dog'` and that kind of logic also works in SQL. But suppose we want animals that were cats, dogs, birds, turtles, ferrets, guinea pigs, hamsters or hedgehogs. With traditional programming logic that would be

```
an_type = 'cat' or an_type = 'dog' or an_type = 'bird' or an_type = 'ferret'  
or an_type = 'guinea pig' or an_type = 'hamster' or an_type = 'turtle' or  
an_type = 'hedgehog'
```

With SQL we can use an In list

```
an_type IN ('cat', 'dog', 'bird', 'ferret', 'guinea pig', 'hamster', 'turtle',  
'hedgehog')
```

The IN list is easier to write, easier to read, easier to maintain and it avoids a potential problem with the precedence of the logical operators. (Which one would you prefer to write on an exam?)

So why do people write that long set of Or tests? Often because that is what they have already learned and it sticks in their brain. One of the purposes of this class is to shake up your brain a bit and get you to try new techniques. So some assignments require you to use certain techniques and avoid other techniques. Once you have some experience with the In list, you may find that you actually do prefer it to a long set of ORs. As the assignments go on you will have more flexibility in the exact techniques you use- but always pay attention in assignments to required techniques.

Table of Contents

1. Testing your query	1
1.1. Gotchas for testing your query	1
1.1. Parent and child rows.....	2
2. Resetting the tables	2

I showed you the syntax of the insert statement in unit 01. What I want to discuss here is something of the logic for inserting rows into a table. Your first question might be why would you want to do that since I provide the inserts for all of the demo and assignment tables.

1. Testing your query

One time that you might want to do inserts is to test your query. Suppose I ask you to write a query that finds all of the dogs in the animals table that have the name 'Pudding'. We have not discussed all of the following row filters, but you should be able to see that this query does that task.

```
select *
from vt_animals
where an_type = 'dog' and an_name = 'Pudding';
```

But if you run this query, the result is empty; we do not currently have any dogs named 'Pudding'. But that does not give you any reassurance that the query is correct. There are a lot of queries that you could write that return an empty result.

So you might want to add a row to the animals table that adds a dog named 'Pudding'. Following the model in the inserts script, you might try the following:

```
insert into vt_animals ( an_id, cl_id, an_name, an_type, an_dob)
values(15165, 42, 'Pudding', 'dog', date '2007-07-07');
```

The syntax is valid, but that insert won't run. If you review the document on the vets database you can detect two errors in the insert.

- 1) The column an_id is the primary key for this table and we already have an animal row using the an_id value 15165. So we would need to use a value for an_id that does not match any of our current rows.
- 2) The column cl_id is the client id and it is the foreign key to the clients table, so that value has to match a client we have in the clients table. We do not have a client with an id of 42.

We could do the following insert:

```
insert into vt_animals ( an_id, cl_id, an_name, an_type, an_dob)
values(15, 5689, 'Pudding', 'dog', date '2007-07-07' );
```

Now if we run the first query again- to find dogs named 'Pudding', we get our new row in the result.

1.1. Gotchas for testing your query

One thing that can get confusing is that you cannot prove your query is correct by examining the result. For example, I could write the following query- which returns the same result but which is not a valid query to find dogs named Pudding.

```
select *
from vt_animals
where an_dob = date '2007-07-07';
```

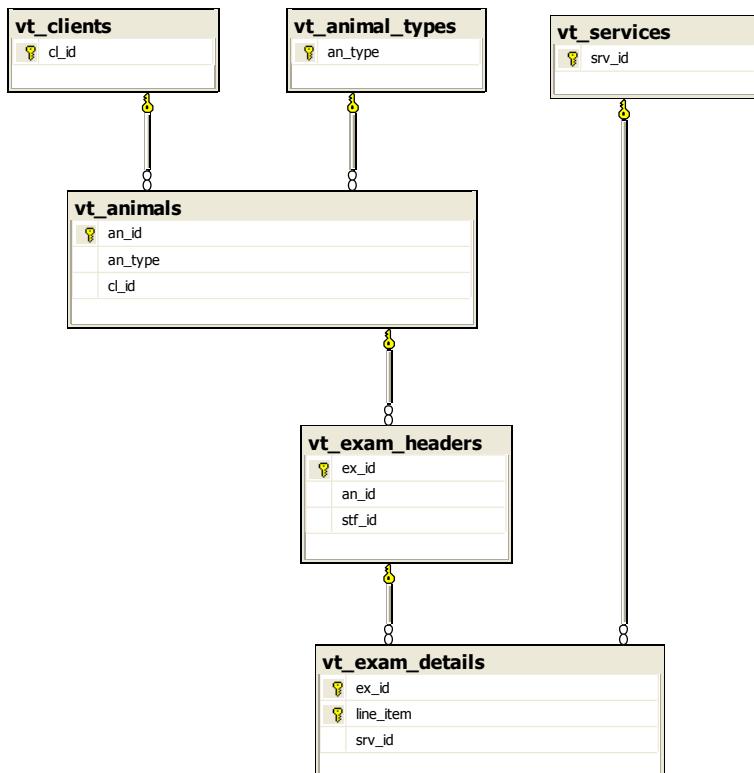
You can demonstrate that your query is wrong if it returns an incorrect result- if your query returns cats then your query is incorrect. But you cannot prove your query is correct by examining the result.

But it can help to add test rows.

1.1. Parent and child rows

The following is a diagram of some of the tables in the vets databases that shows the tables and their PK and FK columns. Note that the tables for clients, animal types and services have no FK columns. You can add columns to these tables without worrying about FK- without worrying about the parent table rows.

But if you add a row to the animals table, you need to worry about the an_type (it needs to match a row in the animal_types table) and about the cl_id (which needs to match a row in the clients table).



Suppose you want to insert a row in the `vt_exam_details` table; in that case you need to have FK matches for the other 5 tables shown.

The `srv_id` needs to match a `srv_id` in the `vt_services` table; the `ex_id` needs to match a row in the `vt_exam_headers` table.

The `an_id` value in the `vt_exam_headers` table needs to match an `an_id` in the `vt_animals` table.

The `cl_id` value in the `vt_animals` table needs to match an `cl_id` in the `vt_clients` table. The `an_type` value in the `animals` table needs to match an `an_type` in the `vt_animal_types` table.

You could build on the existing rows to add additional rows to the tables, starting with the insert for the parent before the insert for the child.

2. Resetting the tables

When you run the assignment script to turn in, you need to use the original inserts that I provided- this should not include any test data that you inserted. You did the test inserts to test your queries- but not for grading.

If you did a simple one row add- such as in the start of the discussion, you could do a simple delete based on the pk.

```
delete from vt_animals where an_id = 15;
```

But if you have added several rows, it can be quicker to simply rerun the inserts script before you run the assignment script for grading.

Table of Contents

1. Predicates and nulls	1
1.1. Predicates and propositions (Optional)	2
2. Logical processing order of the Select statement.....	2

1. Predicates and nulls

The expressions we have been using in the Where clause are called predicates- they are expressions that have a truth value- that value can be True, False, or Unknown. Let's consider the z_name value in the zoo_2016 table.

These are the rows I currently have in that table.

```
select z_id, z_type, z_name
from zoo_2016;
```

Z_ID	Z_TYPE	Z_NAME
23	Giraffe	Sam
25	Armadillo	Abigail
56	Lion	Leon
57	Lion	Lenora
85	Giraffe	Sally Robinson
43	Zebra	Huey
44	Zebra	Dewey
45	Zebra	Louie
47	Horse	
52	Giraffe	Dewey

Suppose we write the following query:

```
select z_id, z_type, z_name
from zoo_2016
where z_name in ('Sam', 'Dewey', 'Trixie')
;
```

Z_ID	Z_TYPE	Z_NAME
23	Giraffe	Sam
44	Zebra	Dewey
52	Giraffe	Dewey

This query is executed by looking at each row in the zoo table, one row at a time, evaluating that expression.

The query is **logically** executed as using the From clause to get data from the zoo_2016 table into a temporary storage area

Then evaluating the Where clause- looking at the first row

23 Giraffe Sam

and evaluating the expression in the Where clause; for this row that expression evaluates as True and the row is passed into the result set.

Then the query is logically executed as looking at the next row

25 Armadillo Abigail

and evaluating the expression in the Where clause; for this row that expression evaluates as False and the row is not passed into the result set.

Then the query is logically executed as looking at the next row

56 Lion Leon

and evaluating the expression in the Where clause; for this row that expression evaluates as False and the row is not passed into the result set.

The predicate is evaluated independently for each row as the query executes. The value of the predicate , `z_name` in ('Sam', 'Dewey', 'Trixie'), is dependent on the value of the column `z_name` for this row. In the case of the row for `z_id` 47, the name column is null and the truth value of the expression is unknown.

We are not sure what the human meaning is of the null in the `z_name` column for this row- it could be that is a horse with no name or it could be that the horse has a name but the person entering the data did not know what the name is; it could even be that the horse is a very private animal and did not want his name entered in the database! But as far as SQL and queries are concerned, the query is evaluated as if we do not know the name and therefore we do not know if the name is Sam or Dewey or Trixie - we just do not know- and so the value of the predicate is Unknown. The rule for a Where clause is that if the predicate evaluates as True then the row is returned into the result; if the predicate is evaluated as False or a Unknown the row is not returned into the result.

Suppose we change the Where clause to Where `z_name` IsNull. With that predicate, the value of the predicate is True for the row with `z_id` = 47 and False for the other rows.

1.1. Predicates and propositions (Optional)

The expression `z_name` in ('Sam', 'Dewey', 'Trixie') is called a predicate. It contains a parameter (`z_name`). We cannot tell if this predicate is True or not until we know a specific value for `z_name`. Once the dbms evaluates the expression for the row with `z_id` 23, the expression becomes 'Sam' in ('Sam', 'Dewey', 'Trixie'). That is a proposition and has a truth value. The value of that proposition is True so that row is in the result. When the dbms evaluates the expression for the row with `z_id` 25, the expression becomes 'Abigail' in ('Sam', 'Dewey', 'Trixie') and that is not true so that row is not in the result.

Examples:

Predicate: `x > 10` We cannot tell if this expression has the value True or False since we do not have a value for `x`

Proposition: `15 > 10`. That has the value True.

Proposition: `2 > 10`. That has the value False.

2. Logical processing order of the Select statement

When you create a select statement, you write the statements with the following model.

```
Select    column_expressions
From     table_expression
Where    filter_expressions
Order By sort-keys;
```

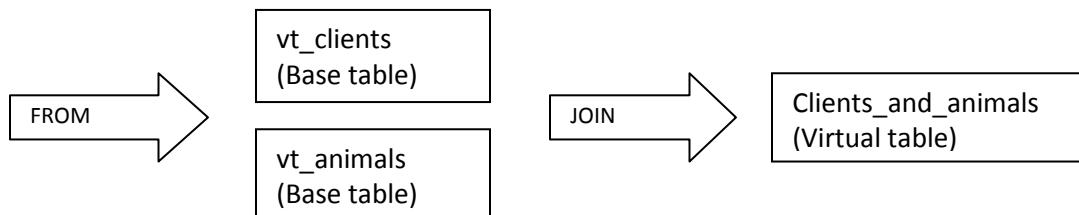
When the query is presented to the dbms, the parser and optimizer determine the actual steps used in the execution and you do not directly control that. But you should consider the statement as being processed in the following order.

1. The FROM clause is evaluated first
2. The WHERE clause
3. The SELECT clause
4. The ORDER BY clause is done last

Suppose we are running this query; the From clause joins two tables.

```
select cl_name_last || ', ' || cl_name_first as ClientName
, an_name as AnimalName
from vt_clients CL
join vt_animals AN on CL.cl_id = AN.cl_id
where an_type in ('cat', 'dog')
order by ClientName, AnimalName;
```

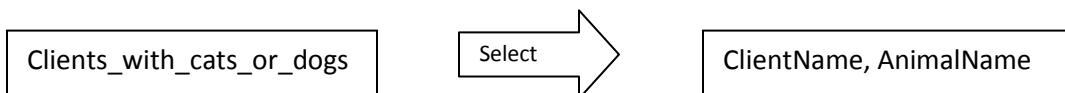
The **From** clause assembles a collection of rows from the table or tables to be used. In this unit we start to talk about more complex From clauses- but the purpose of the From clause is to get data from the tables and build the first virtual table. The optimizer ends up determining the order in which these rows are ordered in the virtual table.



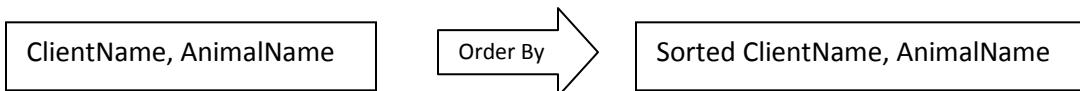
The **Where** clause filters that first virtual table for the rows that meet the Where clause filters. Assume we have a Where clause that filters for an_type in ('cat', 'dog'). The Where clause now produces a second virtual table that includes only the clients with cats and/or dogs. We have no control over the order of the rows in this virtual table.



The **Select** clause now takes that virtual table and returns only the columns and expressions in the Select clause. Perhaps we want to display only the client name and the animal name. We can also provide column aliases at this step. Since the column alias is defined only in the Select clause, Oracle does not let you use column aliases in the Where clause. Oracle also does not let you use the column alias for one of the columns to calculate another column in the select.



The **Order By** clause takes that last virtual table and sorts the rows.



Since the column aliases are now defined, we can use them in the Order By clause.

```

select cl_name_last || ' ', ' || cl_name_first as ClientName
, an_name as AnimalName
from vt_clients CL
join vt_animals AN on CL.cl_id = AN.cl_id
where an_type in ('cat', 'dog')
order by ClientName, AnimalName;
    
```

Table of Contents

1. Compound criteria	1
1.1. The AND logical operator	1
1.2. The OR logical operator.....	3
1.3. The NOT logical operator	5
2. Hierarchy of evaluation of the logical operators	6
3. DeMorgan's laws.....	7
4. Three-way logic and truth tables	8

1. Compound criteria

For more interesting queries, we can use compound criteria. These are criteria that contain multiple conditions joined with the logical operators AND, OR, and NOT.

1.1. The AND logical operator

With this operator, the compound test has a true value if both conditions are true.

Demo 01: We want to see employees who have a salary greater than 15000

```
select emp_id, name_last as "Employee", salary, dept_id
from emp_employees
where salary > 15000;
```

EMP_ID	Employee	SALARY	DEPT_ID
100	King	100000	10
101	Koch	98005	30
102	D'Haa	60300	215
145	Russ	59000	80
146	Partne	88954	215
108	Green	62000	30
205	Higgs	75000	30
162	Holme	98000	35
200	Whale	65000	35
...			

Demo 02: We want to see employees in dept 30 who have a salary greater than 15000. A row has to pass both tests to be included in the result set.

```
select emp_id, name_last as "Employee", salary, dept_id
from emp_employees
where salary > 60000
AND dept_id =30;
```

EMP_ID	Employee	SALARY	DEPT_ID
101	Koch	98005	30
108	Green	62000	30
205	Higgs	75000	30
203	Mays	64450	30
109	Fiet	65000	30
110	Chen	60300	30
206	Geitz	88954	30
204	King	99090	30

When we AND in another filter we will generally reduce the number of rows returned by the query.

Demo 03: We want to see jobs that do not seem to be in Sales with a minimum salary more than 40000. We cannot be certain that these are all of the non-sales jobs- just that they are jobs which do not have Sales in the job title. The Like filter is discussed in another document in this unit.

```
select job_id, min_salary, max_salary
from emp_jobs
where job_title NOT LIKE '%Sales%'
AND min_salary > 40000
;
```

JOB_ID	MIN_SALARY	MAX_SALARY
1	100000	100000
16	60000	120000
32	60000	
64	60000	
128	60000	

Demo 04: This shows employees with a salary between 20000 and 60000 . The Between operator tests True for the end points.

```
select emp_id, name_last as "Employee", salary
from emp_employees
where salary between 20000 and 60000
order by salary, emp_id;
```

EMP_ID	Employee	SALARY
150	Tuck	20000
155	Hiller	29000
207	Russ	30000
145	Russ	59000

Demo 05: If you need to exclude the end point, then use expression $> x$ and expression $< y$ for a strictly greater than test.

```
select emp_id, name_last as "Employee", salary
from emp_employees
where salary > 20000
AND salary < 60000
order by salary;
```

EMP_ID	Employee	SALARY
155	Hiller	29000
207	Russ	30000
145	Russ	59000

Demo 06: Avoid writing tests that logically can never have a True value. What value for salary could pass both of these tests?

```
select emp_id, name_last as "Employee", salary
from emp_employees
where salary < 12000
AND salary > 25000
order by salary;
```

no rows selected

Demo 07: You are not limited to combining two tests.

```
select emp_id, name_last as "Employee"
, dept_id, salary, job_id
from emp_employees
where dept_id = 35
AND salary > 50000
AND job_id in (8, 16)
;
```

EMP_ID	Employee	DEPT_ID	SALARY	JOB_ID
162	Holme	35	98000	16
200	Whale	35	65000	16

1.2. The OR logical operator

With this operator, the compound test has a true value if either one or both conditions are true.

Demo 08: Find employees who work in either dept 20 or 30. It would be better to use an IN operator for this test. Notice that you have to repeat the full test for each OR clause.

```
select emp_id, name_last as "Employee", dept_id
from emp_employees
where dept_id = 30
OR      dept_id = 20
order by "Employee"
;
```

EMP_ID	Employee	DEPT_ID
110	Chen	30
109	Fiet	30
206	Geitz	30
108	Green	30
201	Harts	20
205	Higgs	30
204	King	30
101	Koch	30
203	Mays	30

Demo 09: Here we want employees who earn more than 70000

```
select emp_id
, hire_date, salary, job_id
from emp_employees
where salary > 70000
order by emp_id;
```

EMP_ID	HIRE_DATE	SALARY	JOB_ID
100	17-JUN-89	100000	1
101	17-JUN-08	98005	16
146	29-FEB-12	88954	64
161	15-JUN-11	120000	16
162	17-MAR-11	98000	16
204	15-JUN-13	99090	32
205	01-JUN-08	75000	16
206	15-JUN-13	88954	32

8 rows selected.

Demo 10: Here we want employees who earn more than 70000 OR are in dept 30.

```
select emp_id
, dept_id, salary, job_id
from emp_employees
where dept_id = 30
OR      salary > 70000
order by emp_id
;
```

EMP_ID	DEPT_ID	SALARY	JOB_ID
100	10	100000	1
101	30	98005	16
108	30	62000	16
109	30	65000	32
110	30	60300	32
146	215	88954	64
161	215	120000	16
162	35	98000	16
203	30	64450	16
204	30	99090	32
205	30	75000	16
206	30	88954	32

12 rows selected.

Demo 11: Now we add another possibility - that the employee's job id is 8 or 16

```
select emp_id
, hire_date, salary, job_id
from emp_employees
where dept_id = 30
OR      salary > 70000
OR      job_id in (8, 16)
order by emp_id
;
```

EMP_ID	HIRE_DATE	SALARY	JOB_ID
100	17-JUN-89	100000	1
101	17-JUN-08	98005	16
108	14-APR-95	62000	16
109	29-FEB-12	65000	32
110	31-DEC-12	60300	32
146	29-FEB-12	88954	64
150	28-OCT-01	20000	8
155	05-MAR-04	29000	8
161	15-JUN-11	120000	16
162	17-MAR-11	98000	16
200	17-JUN-11	65000	16
203	30-JUN-10	64450	16
204	15-JUN-13	99090	32
205	01-JUN-08	75000	16
206	15-JUN-13	88954	32
207	17-JUN-11	30000	8

16 rows selected.

With each additional OR clause we add, we have the potential of having more rows match.

Demo 12: We have query for max_salary >= 20000

Here we are also including the nulls with an IS NULL test

```
select job_id, job_title, min_salary, max_salary
from emp_jobs
where max_salary >= 20000
OR max_salary is null;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	President	100000	100000
2	Marketing	5000	75000
4	Sales Manager	15000	60000
8	Sales Rep	10000	30000
16	Programmer	60000	120000
32	Code Debugger	60000	
64	DBA	60000	
128	RD	60000	

8 rows selected.

1.3. The NOT logical operator

The NOT operator works on a single test and reverses the value of that test. The NOT test is often used in combination with AND or OR tests.

Demo 13: We want employees who are **not** in department 20 or 30.

```
select emp_id, name_last as "Employee", dept_id
from emp_employees
where NOT dept_id IN ( 30, 20)
order by "Employee"
;
```

EMP_ID	Employee	DEPT_ID
102	D'Haa	215
161	Dewal	215
160	Dorna	215
104	Ernst	210
155	Hiller	80
162	Holme	35
103	Hunol	210
100	King	10
146	Partne	215
207	Russ	35
145	Russ	80
150	Tuck	80
200	Whale	35

13 rows selected.

The above test could also be written as Where dept_id NOT IN (30, 20) and I think that is easier to read. Note that NOT IN is closer to the way the task is described. I would also encourage you to use Where salary not between 10000 and 20000 instead of Where NOT salary between 10000 and 20000.

Using the not operator before the tests means that your mind has to keep track of the NOT while it reads the rest of the test.

Take extra care when using two NOT words in the same test- often people get the logic of double negatives wrong.

2. Hierarchy of evaluation of the logical operators

If you write a criterion that includes more than one logical operator, you need to be concerned about the hierarchy of evaluation. The order of operations is: first the NOT operators are evaluated then the ANDs and then the ORs. Parentheses are used to change the order of operations.

Suppose we want to see products that are either pet supplies or sporting goods that cost less than 100. This is an ambiguous statement. Assume this essentially means we want the cheaper sporting good and the cheaper pet supplies items.

Demo 14: This query following the wording of the task description but does not do the job. We have two Pet items that cost more than \$100.

```
select prod_id, prod_list_price, catg_id
from prd_products
where catg_id = 'PET' OR catg_id = 'SPG'
AND prod_list_price < 100;
```

PROD_ID	PROD_LIST_PRICE	CATG_I
1020	12.95	SPG
1030	29.95	SPG
1140	14.99	PET
1141	99.99	PET
1142	2.5	PET
1143	2.5	PET
1150	4.99	PET
1151	14.99	PET
1152	55	PET
4567	549.99	PET
4568	549.99	PET
4576	29.95	PET
4577	29.95	PET

13 rows selected

Demo 15: If we reverse the testing of the two categories, we get sporting goods items that cost more than \$100. That is not right.

```
select prod_id, prod_list_price, catg_id
from prd_products
where catg_id = 'SPG' OR catg_id = 'PET'
AND prod_list_price < 100;
```

PROD_ID	PROD_LIST_PRICE	CATG_I
1010	150	SPG
1020	12.95	SPG
1030	29.95	SPG
1140	14.99	PET
1141	99.99	PET
1142	2.5	PET
1143	2.5	PET
1150	4.99	PET
1151	14.99	PET
1152	55	PET
1040	349.95	SPG
1050	269.95	SPG
1060	255.95	SPG
4576	29.95	PET
4577	29.95	PET

15 rows selected.

What is happening here is that we have an AND operator and an OR operator. The rules of precedence is that the AND operator is evaluated first. So the second of these where clauses

```
where catg_id = 'SPG' or catg_id = 'PET' and prod_list_price < 100;
```

is evaluated as shown here and all of the sporting goods items are returned and Pet supplies that cost more than \$100 are returned.

```
where catg_id = 'SPG' or (catg_id = 'PET' and prod_list_price < 100);
```

We can use parentheses to change the order of evaluation

Demo 16: Adding the parentheses gives us the correct result.

```
select prod_id, prod_list_price, catg_id
from prd_products
where (catg_id = 'SPG' OR catg_id = 'PET')
AND prod_list_price < 100;
```

PROD_ID	PROD_LIST_PRICE	CATG_I
1020	12.95	SPG
1030	29.95	SPG
1140	14.99	PET
1141	99.99	PET
1142	2.5	PET
1143	2.5	PET
1150	4.99	PET
1151	14.99	PET
1152	55	PET
4576	29.95	PET
4577	29.95	PET

11 rows selected.

Demo 17: It is better to use the IN operator, avoiding the AND/OR Issue.

```
select prod_id, prod_list_price, catg_id
from prd_products
where catg_id IN ( 'SPG', 'PET')
AND prod_list_price < 100;
```

3. DeMorgan's laws

Often, there is more than one way to write a complex logical expression. The following equivalencies are known as DeMorgan's Laws.

Where expP and expQ represent logical expressions

NOT (expP AND expQ) is equivalent to

NOT expP OR NOT expQ

NOT (expP OR expQ) is equivalent to

NOT expP AND NOT expQ

Demo 18: These are equivalent queries. Prod_list_price and catg_id are not null in the products table.

```
select prod_id
, prod_desc
, prod_list_price, catg_id
from prd_products
where NOT( prod_list_price < 300 OR catg_id = 'APL');
```

```

select prod_id
, prod_desc
, prod_list_price, catg_id
from prd_products
where NOT( prod_list_price < 300) AND NOT( catg_id = 'APL');

select prod_id
, prod_desc
, prod_list_price, catg_id
from prd_products
where prod_list_price >= 300 AND catg_id != 'APL';

```

4. Three-way logic and truth tables

Generally we think of logical expressions having two possible values — True and False. Because database systems allow the use of Null, we have to be concerned with three logical values —True, False, and Unknown. Suppose we have a row in the jobs table with no value for the attribute max_salary , and we evaluate the logical expression: max_salary > 25000 the value of the expression is Unknown for that row. If you are executing a query with a Where clause, if the value of the test is Unknown, the row is not returned.

Remember, NULL is a data value, UNKNOWN is a logical value.

These are the truth tables for the operators NOT, AND, and OR.

The evaluation of the True and False cases are straight forward. With the NOT operator, if I do not know the value of an expression is True or False then I do not know if the negation of that expression is True or False.

NOT	
True	False
Unknown	Unknown
False	True

For the AND operator to Return True both of the operators must have a True value. So if one of the operands is True and the other is unknown, then I cannot know if the ANDed expression is true- so the value is unknown. But if one of the operands is False, then the ANDed expression cannot be true and we know its value is False.

AND	True	Unknown	False
True	True	Unknown	False
Unknown	Unknown	Unknown	False
False	False	False	False

For the OR operator to Return True at least one of the operators must have a True value. So if one of the operands is True and the other is unknown, then the ORed expression is TRUE. If one of the operands is False and the other is unknown then I cannot know the value of the Ored expression and its value is Unknown.

OR	True	Unknown	False
True	True	True	True
Unknown	True	Unknown	Unknown
False	True	Unknown	False

Demo 19: Cust id 402500 has a null for the credit limit column

```
select customer_id, customer_name_last
from cust_customers
where customer_id = 402500;

```

CUSTOMER_ID	CUSTOMER_NAME_LAST
402500	Jones

1 row selected.

Demo 20: Cust id 402500 passes the first of these tests; the second test for that row has a value of unknown and therefore Cust id 402500 is not returned by this query since the tests are ANDed

```
select *
from cust_customers
where customer_id < 403050 AND customer_credit_limit < 1000;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_LAST	CUSTOMER_CREDIT_LIMIT
400801	Washington	Geo	750
401250	Morse	Samuel	750
402100	Morise	William	750
402110	Coltrane	John	750
402120	McCoy	Tyner	750

5 rows selected.

Demo 21: Cust id 402500 passes the first of these tests; the second test for that row has a value of unknown and Cust id 402500 is returned by this query since the tests are ORed and Cust id passed at least one of the tests

```
select *
from cust_customers
where customer_id < 403050 OR customer_credit_limit < 1000;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_LAST	CUSTOMER_CREDIT_LIMIT
401250	Morse	Samuel	750
401890	Northrep	William	1750
402100	Morise	William	750
402110	Coltrane	John	750
402120	McCoy	Tyner	750
402500	Jones	Elton John	
403000	Williams	Sally	6000
403010	Otis	Elisha	6000
400801	Washington	Geo	750
400300	McGold	Arnold	6000

10 rows selected.

Table of Contents

1. SQL wildcards	1
2. Numbers and Like- don't do this!.....	3
3. Dates and Like- don't do this!	3
4. To escape a wildcard:.....	3

1. SQL wildcards

Wildcard characters are used when you want to do a **partial string match**. Suppose you are looking for any product that has the word "mixer" in the product description. We might want to find 'portable mixer', '6-speed mixer'. We can use the Like operator and a wildcard pattern to find all products which contain the pattern 'mixer'- this will also find 'cement mixers' but that is the way that wildcard matching works. We cannot really do a **word** match, but we can do a pattern match.

Wildcards are used for matching strings. It is acceptable to match digits in text values- such as in an ISBN code. Oracle will do wildcard matching with numeric values but this is not a good idea- **wildcards are designed for string matching and numbers are not strings**. It is common to do wildcard matches with date values- but that relies on date formats. There are safer ways and generally more efficient ways to test dates and numeric values.

The wildcard patterns are interpreted as wildcards only if you use the LIKE operator.

You can also test using the operator NOT LIKE to find rows that do not match your pattern.

Do not use the LIKE operator if you are not using wildcards. The Like operator is more expensive- if you are testing if the customer city is Chicago- that is an equality test, not a wildcard test.

The wildcard patterns used are:

- % for zero or more characters, matching any character
- _ for a single character, matching any character

For these examples we are using a table named z_wildcards with the following rows:

CUSTID	CUSTPHONE	CUSTADDRESS
1	510-239-8989	101 Bush Street
2	510-45-78785	One Bush Street
3	415-809-8989	124 High
4	415-124-2398	15 High Road
5	415-239-8523	1554 Rural Highway 12

Demo 01: This is the SQL to create and populate the table.

```
Create table z_wildcards (cust_id number(2), cust_phone varchar2(12),
cust_address varchar2(30));
insert into z_wildcards values (1, '510-239-8989', '101 Bush Street');
insert into z_wildcards values (2, '510-45-78785', 'One Bush Street');
insert into z_wildcards values (3, '415-809-8989', '124 High');
insert into z_wildcards values (4, '415-124-2398', '15 High Road');
insert into z_wildcards values (5, '415-239-8523', '1554 Rural Highway 12');
```

Demo 02: We want to locate customers with an address on Bush.

```
Select cust_id, cust_address
From z_wildcards
Where cust_address LIKE '%Bush%';
```

CUST_ID	CUST_ADDRESS
1	101 Bush Street
2	One Bush Street

Demo 03: We want to locate customers whose phone number is in the 415 area code.

```
Select cust_id, cust_phone
From z_wildcards
Where cust_phone LIKE '415%';
```

CUST_ID	CUST_PHONE
3	415-809-8989
4	415-124-2398
5	415-239-8523

Demo 04: We want to find customers in the 239 exchange (the middle 3 digits of their phone number). This does not work properly.

```
Select cust_id, cust_phone
From z_wildcards
Where cust_phone LIKE '%239%';
;
```

CUST_ID	CUST_PHONE
1	510-239-8989
4	415-124-2398
5	415-239-8523

Demo 05: We can look for the pattern '-239-'.

```
Select cust_id, cust_phone
From z_wildcards
Where cust_phone LIKE '%--239--';
```

CUST_ID	CUST_PHONE
1	510-239-8989
5	415-239-8523

Demo 06: Text strings can be harder to match. We want to locate customers with an address on High – and we don't care if it is a street, road, etc. If we try this approach, we also get the Cust 5.

```
Select cust_id, cust_address
From z_wildcards
Where cust_address LIKE '%High%';
;
```

CUST_ID	CUST_ADDRESS
3	124 High
4	15 High Road
5	1554 Rural Highway 12

Demo 07: If we try adding a space after the word High, we miss Cust 3.

```
Select cust_id, cust_address
From z_wildcards
Where cust_address LIKE '%High %';
;
```

CUST_ID	CUST_ADDRESS
4	15 High Road

Do not use the LIKE operator if you are doing an exact match. It makes no sense to test for a specific phone number by using the following query. This should be an equality test.

```
Select cust_id, cust_phone
From z_wildcards
Where cust_phone LIKE '510-239-8989';
```

Wildcards are only interpreted if you use the LIKE predicate.

Use the Like operator for string comparisons only. There are safer ways to test numbers and dates. Note there is no problem with testing data that is stored as a string of digits (such as a ssn) with the Like operator.

We will discuss regular expression for testing soon; these provide more precise testing of patterns.

2. Numbers and Like- don't do this!

Use the Like operator for string comparisons only. There are safer ways to test numbers and dates. The following shows some of the issues in testing numbers with wildcards in Oracle.

Demo 08: This will find rows where the list price for a service contains the digit 5. I do not know why anyone would want to do that.

```
Select *
From vt_services
Where srv_list_price like '%5%';
```

Demo 09: This looks like it is trying to find prices that are not even dollar prices- prices that have digits after the decimal point. But because of the way that Oracle casts data types a value such as \$45 is not necessarily cast to 45.00. Remember that Like expects to work with strings and numbers are not strings.

```
Select *
From vt_services
Where srv_list_price not like '%.00%';
```

3. Dates and Like- don't do this!

Using Like with date values can also problems. Suppose we want to filter the exam headers tables for certain date components. I will discuss dates and the like operator in another document in this unit.

4. To escape a wildcard:

Since the wildcard is part of the SQL statement and is sent to the server, you cannot escape the wildcard character via SQL*Plus- you have to do this within the SQL statement.

Suppose we had data values that included the % symbol and we wanted to search for those values. The Where clause would start as

```
WHERE col_1 LIKE '%%%
```

But this is an obvious problem since we need to indicate that the middle % is really just a percent symbol and not a wildcard. You need to define an escape character for this query- I am going to use the # character. Now my Where clause looks like this:

```
WHERE col_1 LIKE '%#%' Escape '#'
```

The first % is a wildcard, the second % is the literal character and the third % is again a wild card.

If you are looking for a value which contains two % characters, you could use

```
WHERE col_1 LIKE '%#%#%' Escape '#'
```

We will discuss regular expressions soon which provides for more precise matching.

Table of Contents

1. Dates	1
1.1. Date values are not stored as strings.....	1
1.1. Date versus DateTime	2
2. Testing with a Date value.....	2
3. Comparing string to string.....	2
3.1. Using Between with dates.....	2
3.2. Dates and Like	3

1. Dates

Date values are essential to most systems. But date values can be confusing.

1.1. Date values are not stored as strings.

We enter dates using strings and they look like strings when we display them; but dates are dates - not strings.

Oracle has a default format for entering and displaying strings. dd-MON-yy.

Demo 01: set up the following table with a date column and insert dates using various styles.

```
create table z_tst_dates_0 (
    id integer primary key
, col_date date not null
);
insert into z_tst_dates_0 values(1, '12-JUN-15') ;
insert into z_tst_dates_0 values(2, '12-jun-2015') ;
insert into z_tst_dates_0 values(3, '12-JUN-1915') ;
insert into z_tst_dates_0 values(4, date '1915-06-12') ;
insert into z_tst_dates_0 values(5, date '2015-06-12') ;
```

--This is the default date format for many Oracle systems. Note that rows 3 and 4 appear to be the same value as the other rows.

```
select * from z_tst_dates_0;
```

ID	COL_DATE
1	12-JUN-15
2	12-JUN-15
3	12-JUN-15
4	12-JUN-15
5	12-JUN-15

-- But if I use a format that includes the full4 digit year, I can see the difference.

```
select id, to_char(col_date, 'yyyy-mm-dd') from z_tst_dates_0;
```

ID	TO_CHAR(CO
1	2015-06-12
2	2015-06-12
3	1915-06-12
4	1915-06-12
5	2015-06-12

1.1. Date versus DateTime

Oracle does not have a separate date only type. The name of the type is Date- but it always includes both a date component and a time component. The time component is not part of the default display format.

The following expression will include the time component of a date value with a precision of Hour and Minute
`To_char(ex_date, 'YYYY-MM-DD HH24:mi')` We will discuss this function and more formats in another unit.

2. Testing with a Date value

If you are positive that all of the date values for a column were stored with the time component set to midnight then date testing is easier. In the vt_animals table the an_dob all have a time component of midnight. But in the vt_exam_headers table, the ex_date values have a time component.

Demo 02: These are the rows in the vt_exam_headers table for the month of April 2015.

EX_ID	EXAMDATE
2228	2015-04-04 12:30
2205	2015-04-08 10:30
2289	2015-04-11 13:00
2290	2015-04-11 17:00

3. Comparing string to string

Demo 03: If I test for exams on April 4, 2015 using date '2015-04-08' or '04-APR-15', I get no matches.

```
select ex_id , ex_date
from vt_exam_headers
where ex_date = date '2015-04-08';
no rows selected
```

```
select ex_id , ex_date
from vt_exam_headers
where ex_date = '04-APR-15';
no rows selected
```

Demo 04: What I need to do is cast the ex_date to a **string** which has the pattern YYYY-MM-DD and then compare that string expression to the proper string literal.

```
select ex_id , ex_date
from vt_exam_headers
where to_char(ex_date, 'YYYY-MM-DD') = '2015-04-08'
;
EX_ID EXAMDATE
-----
2205 08-APR-15
```

3.1. Using Between with dates

Suppose I want to display all of the exams in the month of Jan 2016.

Demo 05: I could try a Between test but the following will miss the exam on 2016-01-31 9:00 am. If you do not include a time component, then the date value gets a default time component of midnight. The first query shows we do have 8 rows for Jan 2016.

```
select ex_id , ex_date, to_char(ex_date, 'YYYY-MM-DD HH:Mi')
from vt_exam_headers
order by ex_date desc;
-- selected rows
```

EX_ID	EX_DATE	TO_CHAR(EX_DATE, 'YYYY-MM-DDHH:MI')
3288	31-JAN-16	2016-01-31 09:00
3494	22-JAN-16	2016-01-22 09:00
3325	15-JAN-16	2016-01-15 10:45
3104	09-JAN-16	2016-01-09 04:30
4103	08-JAN-16	2016-01-08 03:30
4102	08-JAN-16	2016-01-08 01:00
4101	02-JAN-16	2016-01-02 01:00
3420	01-JAN-16	2016-01-01 04:30

```
select ex_id , ex_date
from vt_exam_headers
where ex_date Between date '2016-01-01' and date '2016-01-31';
```

EX_ID	EX_DATE
4101	02-JAN-16
4102	08-JAN-16
4103	08-JAN-16
3104	09-JAN-16
3325	15-JAN-16
3420	01-JAN-16
3494	22-JAN-16

I could try a Between test with the upper range value being '2015-02-01' but if we did have a ex_date of 2015-02-01 midnight, that row would be returned.

Demo 06: A better approach is a compound comparison test; note the comparison operators used.

```
select ex_id , ex_date
from vt_exam_headers
where ex_date >= date '2016-01-01' and ex_date < date '2016-02-01';
```

3.2. Dates and Like

Using Like with date values can also problems. Suppose we want to filter the exam headers tables for certain date components.

Demo 07: We might try the following to find exam dates in Jan 2016.

```
select ex_id , ex_date
From vt_exam_headers
Where ex_date like '2016-01%';
```

But that does not return any rows- even though we have exams in Jan 2016.

We could use the default Oracle format and let the system do the conversion. This works.

```
select ex_id , ex_date
From vt_exam_headers
Where ex_date like '%-%JAN-16';
```

This also works since we cast the ex_date to a string that matches our wild card pattern.

```
select ex_id , ex_date
```

```
From vt_exam_headers  
Where to_char(ex_date, 'YYYY-MM-DD') like '2016-01%';
```

But you might as well do a better string pattern and use an equal tests. Wildcard tests are generally more expensive than equality tests.

```
select ex_id , ex_date  
From vt_exam_headers  
Where to_char(ex_date, 'YYYY-MM') = '2016-01';
```

Demo 08: This would find exams done in January of any year. Remember this will be case sensitive

```
select ex_id , ex_date  
from vt_exam_headers  
where ex_date like '%JAN%';
```

But it is better to do a more exact pattern and avoid like.

```
select ex_id , ex_date  
from vt_exam_headers  
where to_char(ex_date, 'MON') = 'JAN';
```

In addition to the problems of matching the default date format for wildcard matching, we do not have a default time format and it is possible that the dba could change the default date format to a different format - such as YYYY-MM-DD- and all code that uses the Like operator with date values will have to be inspected and possibly changed.

Table of Contents

1. Two table join.....	1
2. Inner join	2
3. The vets database relationships.....	2
4. Deciding on tables to use in a query.	3

In unit 2, we looked at the vets database and its collection of tables. Hopefully by now you have noticed that these tables work together to store the data. The first time you work with a relational database it might not make sense that we have so many tables. We have a table for animal data, another for client data and two tables that together hold the exam data; we also have tables for the staff and services. We will talk more about the theory behind having all of these tables later in the semester, but for now I want to focus on how we associate the data in these tables.

1. Two table join

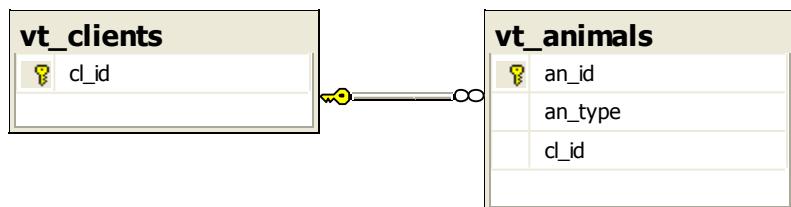
Let's start with two tables- the clients and the animals. One of the rules the vet has for his clinic is that every animal that he treats has to have an associated client- the vet might suggest a dental treatment for the cat Mittens, but the client makes decisions about whether or not Mittens gets her teeth cleaned and it is the client who ends up paying the vet bills for Mittens. There also is a rule that there is only one client stored for each animal. So the dbms needs to have a way to go from the information for Mittens to the client responsible for Mittens. We often talk about joining tables, but it sometimes makes more sense to think about navigating from one table to another to collect all of the data we need.

In database terms, we need to establish a relationship between the clients table and the animals table. Commonly we do this when we set up the tables by defining primary and foreign keys.

This is **part** of the definition of these two tables- I am including mainly the key columns in each table.

```
create table vt_clients(
    cl_id          int      not null primary key
);

create table vt_animals(
    an_id          int      not null primary key
    , an_type       varchar2(25) not null
    , cl_id          int      not null references vt_clients
);
```



In this relationship, the clients table is called the parent table and each client row is identified by the cl_id. The cl_id attribute is the primary key for the clients table.

The animals table is called the child table; each animal row is identified by the an_id. The an_id attribute is the primary key for the animals table. And each animal row also has a value of a cl_id.

The clause `references vt_clients` states that the value in the cl_id column for an animal row has to match a value for cl_id in the clients table. Also this column is said to be Not Null so each animal has to have an associated client. The cl_id attribute in the animals table is the foreign key to the clients table.

Because we have set this rule about animals and clients when we defined the tables, the dbms will not let us enter a row for an animal unless it has a value for cl_id that matches an existing client. The dbms also will not let us delete a client row if that client has associated animal rows. (If we could delete a client and leave the animals rows- they would be called orphaned animal rows- and we don't want that. Who would pay the vet bills?)

You would, quite logically, think that since we set up that relationship and the dbms knows about it that you could just then write SQL that knows the relationships without being told. But that is not the case. When you write a query that wants to see information about an animal and information about the client for that animal, your SQL statement has to state the relationship between the tables- which is called a join in SQL.

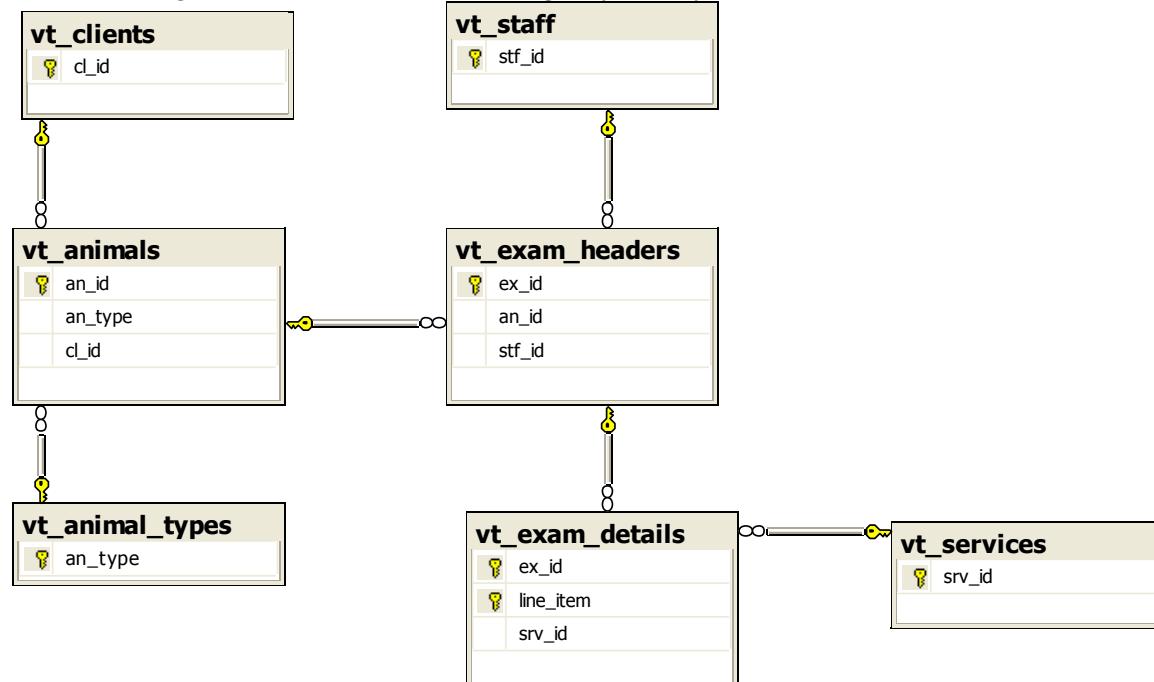
Historically, years ago, a database was commonly set up without defining the relationships in the table definitions and so SQL was developed needing you to define those joins. We can also write joins that use columns other than the PK and FK fields.

2. Inner join

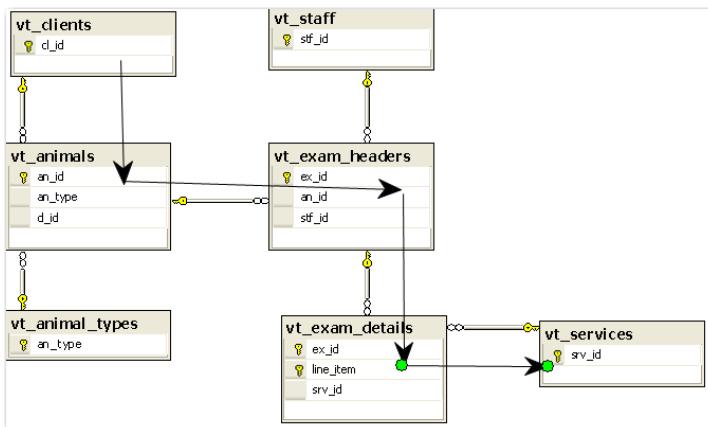
We have several types of joins possible between tables. The first one we work with is called an inner join. An inner join joins two tables using a column in each table that have values in common. A row is included in the result if each of the two tables have matching rows. If we do an inner join between clients and animals then we get results back only for clients who have animals. Soon we will also look at outer joins which would let us get results back both for clients who have animals and for clients who don't have animals.

3. The vets database relationships

This is the diagram for the vets tables showing only the keys for each table.



If you think of the relationship lines on the diagram as pathways between the tables (as possible joins) then you can navigate from the clients table down to the services table and see, for each client, which services his pets has received.



Showing just the table name, the query would look like this

```

select . . .
from vt_clients
join vt_animals . . .
join vt_exam_headers . . .
join vt_exam_details . . .
join vt_services . . .
  
```

4. Deciding on tables to use in a query.

When you are writing a query it helps to look at the diagram and find a path between all the tables you need and then write the From clause **starting at one end of that pathway** and proceed one table at a time.

Sometimes people have trouble deciding which tables to include in a query. This is a technique you might try.

A: Suppose I said to write a query showing the last name of each staff person and the name of the services they performed on an animal and the date they did the service. First make a list of the attributes you need to display or filter for. Here we do not need a filter and we need to display

last name of staff

service description

exam date

Now list the tables these attributes appear in. This one is fairly easy since each of these attributes appear in only one table

Attribute	stf_name_last	srv_desc	ex_date
Table	vt_staff	vt_services	vt_exam_headers

Do we have a direct path between these tables?

```

From vt_staff
join vt_exam_headers
join vt_services
  
```

We can go from **vt_staff** to **vt_exam_headers**. We do not have a direct path from **vt_exam_headers** to **vt_services**, so we need to also include the **vt_exam_details** table

```

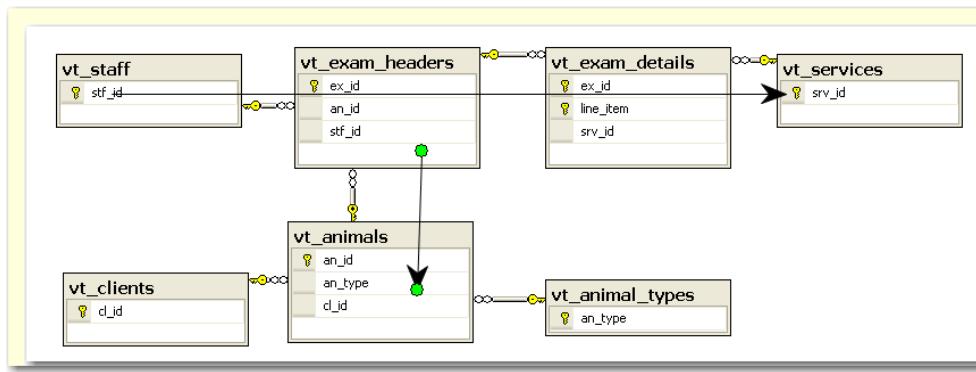
From vt_staff
join vt_exam_headers
join vt_exam_details
join vt_services
  
```

B: now suppose I changed the query and said that I only wanted to see the services if they were performed on a Bird. We need to include the an_type attribute in the Where clause. an_type appears in two tables: vt_animals and vt_animal_types

Attribute	stf_name_last	srv_desc	ex_date	an_type
Table	vt_staff	vt_services	vt_exam_headers	vt_animals
Table				vt_animal_types

In this case we can use the attribute in the vt_animals table since that will let us know if an animal row is a bird or not. There is no need to include the vt_animal_types table.

Now we have a divided path



But we still have a path- we can travel from the exam headers to the animals table.

```

From vt_staff
join vt_exam_headers
join vt_exam_details
join vt_services
join vt_animal
    
```

C: Suppose I asked you to show me the animal id for any animal ever seen by the staff person with the staff id 103. We need to see the an_id which is in two tables and we need to filter on the staff_id which is also in two tables.

Attribute	stf_id	an_id
Table	vt_staff	vt_animals
Table	vt_exam_headers	vt_exam_headers

At first you might think of using

```

From vt_staff
join vt_exam_headers
join vt_animal
    
```

Do we need all three tables in this query? No- the data that we need is all in the table vt_exam_headers and we do not need any join.

Every extra table you add to a join makes the query slower so you want the From clause to be a short as needed (but no shorter!)

Table of Contents

1. Qualified references.....	1
2. Using a Table Alias.....	1
3. SQL inner joins.....	2
4. ANSI inner join: Column Name Join (the USING clause)	3
5. ANSI inner join: Condition Join (the ON clause).....	6
6. NATURAL JOIN.....	9

One of the goals of a relational database is to reduce redundancy; we want to store descriptive data one time only. To do this we need to split data across several tables- so we have a Customer table and an Order table and an Order details table. As a consequence, we will often need to write queries that bring the data back together again from two or more tables. We will need to indicate in the query how these tables are related. We will first discuss table aliases which are commonly used when our query involves more than one table. Then we will look at the ANSI standard inner join syntax.

1. Qualified references

We know that each column in a table needs to have a unique name. But we can have the same identifier for columns in different tables. It is common, and good style, for the pk column in the parent table and the fk column in the child table to have the same name. For example, with the AltgeldMart tables, we have a column for the product id in both the products table and in the orderDetails table; in each of these tables the column name is prod_id. But we also store an employee identifier in the employees table and in the orderHeaders table; in the employees table it makes sense to call this attribute emp_id and it makes sense in the orderHeaders table to call this attribute sales_rep_id.

If we are joining two tables, we may have to qualify any reference to a column name which is the same in the two tables. (The exception is the joining column when we use the Using (col) syntax.) The format for a qualified name is `tblName.ColumnName`.

We can qualify all of the column names in the query. If this is a single table query, there is no need to do this. With a multi-table query, the query may be somewhat more efficient if we fully qualify the column names.

For most of the example in this discussion, I will not fully qualify all of the column names since the queries are easier to read without the qualification. The purpose of these demos is to show you techniques and syntax- and ease of reading is more important for this purpose than execution efficiency.

An **ambiguous reference** means that you have a column identifier in your query and the same identifier is used in more than one of the tables used in the query. Therefore the system does not know which column is being referenced. This is an error.

2. Using a Table Alias

The table aliases (correlation names) are alternate names for tables. The table alias is defined in the From clause of the SQL statement and is limited in scope to that statement. The table alias is not saved on the server. Once you establish a table alias in a query, you need to use that alias, not the table name, in the other clauses of that query.

Some types of queries require the use of table aliases since the same table is included in the query more than once. In other queries the use of table aliases is optional.

The use of table aliases is very common in SQL and you need to be aware of its use. However, poorly defined table aliases can make a query harder to read and understand.

The table alias is commonly a single letter but single letter names are often difficult to read and remember. You should not have to keep referring to the From clause to interpret the Select clause. Table aliases should be long enough to make them meaningful.

Demo 01: A single table select without an explicit table alias

```
select dept_id, dept_name
from emp_departments;
```

DEPT_ID	DEPT_NAME
10	Administration
20	Marketing
30	Development
35	Cloud Computing
210	IT Support
215	IT Support
80	Sales
90	Shipping
95	Logistics

9 rows selected

Demo 02: A single table select with a table alias; this will give us the same output as the first query. The table alias is now used to refer to the table.

```
select dp.dept_id, dp.dept_name
from emp_departments dp;
```

The use of the key word As is not allowed with a table alias.

We could rewrite the second query as

```
select dept_id,dept_name
from emp_departments dp;
```

But we cannot use the following; once we set up the alias we cannot use the table name as a qualifier.

```
select prd_warehouses.warehouse_id
, prd_warehouses.loc_id
from prd_warehouses wh;
```

The error message is ORA-00904: "EMPLOYEE"."DEPARTMENTS"."DEPT_NAME": invalid identifier

3. SQL inner joins

We can use several types of joins between tables. We start with inner joins. For a row to appear in the result returned by an inner join, there needs to be matching data in the joining columns in the two tables. There are several ways to implement the inner join.

These examples show the ANSI standard Condition Join syntax and the Column Name join syntax. Most dbms now support these syntax models.

The legacy comma join syntax is discussed later. **For assignments in this class you are required to use a syntax that does the join in the From clause.** In a job situation where you are reading and maintaining old code, you will need to understand the legacy syntax which expresses the joining condition as a criterion in the Where clause along with any filter criteria.

4. ANSI inner join: Column Name Join (the USING clause)

Let's start with an example of an inner join between the employee table and the department table. We want to see the name of the employee (which is in the employee table) and the name of the department which is in the department table. If we described this situation to someone we might say that we are **joining** the data in these two tables and that we are **using** the department id to associate each employee row with the correct department row. We also want to display the department id. If someone asks us which department id we want- the one from the employee table or the one from the department table, we would reply that we don't care because they have to be the same value.

The ANSI syntax shown here models that way of talking about the join.

Demo 03: Inner join employees and their dept

```
select emp_id
, name_last as "Employee"
, dept_name
from emp_employees
INNER JOIN emp_departments USING (dept_id);
```

EMP_ID	Employee	DEPT_NAME
100	King	Administration
201	Harts	Marketing
205	Higgs	Development
108	Green	Development
101	Koch	Development
206	Geitz	Development
204	King	Development
110	Chen	Development
109	Fiet	Development
...		

Discussion:

With this syntax, the joining column must have the same name in both tables.

We do not qualify the column names in common- since we are really using a generated column that Oracle produced for us. We do need to put parentheses around the common column name in the USING clause.

We need to qualify any other column names that would cause an ambiguous reference.

All of the information about the joining of the tables is placed in the From clause. The From clause is supposed to indicate the data source and this syntax defines the data source.

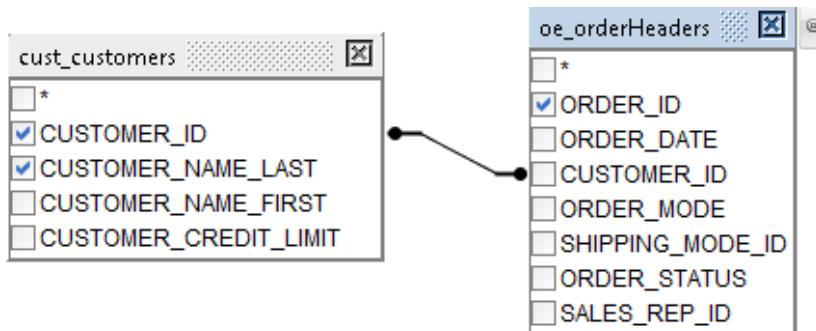
It might seem that the dbms could figure out which columns are the joining columns- particularly if we define the relationships when we create the table- but it doesn't.

Another thing to be aware of is if we have any department with no employees, the inner join will not display them. To get those rows we would need an outer join which we will discuss later.

Demo 04: Inner join with USING - show customers and their orders.

```
select cust_id
, oe_customers.cust_name_last as "Customer"
, oe_order_headers.ord_id
from oe_customers
INNER JOIN oe_order_headers USING (cust_id)
order by cust_id;
```

CUSTOMER_ID	Customer	ORDER_ID
400300	McGold	378
401250	Morse	506
401250	Morse	301
401250	Morse	113
401250	Morse	106
401250	Morse	552
401250	Morse	119
. . . rows omitted		



We can join more than two tables- we just add them to the From clause in a logical order and set up the joining column for each pair of tables as we go. In the next query we get one row for each product a customer has ordered.

Tables are joined two at a time. The customer table is joined to the orderHeaders table to form a virtual table, which is then joined to the orderDetails table to form another virtual table used as the table expression. If you have a straight line inner join, start listing the tables with a table at one end of the path and then list the tables in order. Some people list the tables erratically which can make the joins harder to understand and error prone.

The key word Join and Inner Join both define an inner join. I will skip the work Inner in the following queries to make these easier to read. There is another type of join called an outer join we will discuss later.

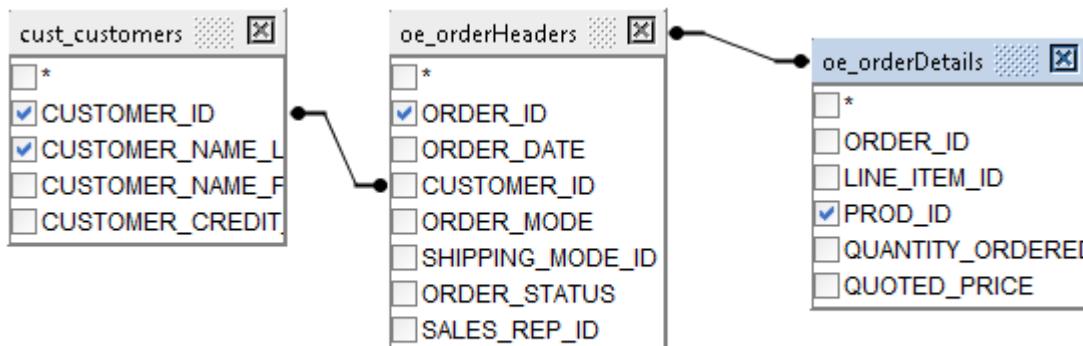
Class standards require that you start a new line for each Join keyword and keep the Using phrase on the same line as the table name. This style of SQL layout makes it easy to scan down the left edge of the query and see the tables involved.

Demo 05: three table join; inner join with column name join

```
select cust_id
, oe_customers.cust_name_last as "Customer"
, ord_id
, oe_order_details.prod_id
from oe_customers
JOIN oe_order_headers USING (cust_id)
JOIN oe_order_details USING (ord_id)
order by cust_id, ord_id;
```

CUSTOMER_ID	Customer	ORDER_ID	PROD_ID
400300	McGold	378	1125
400300	McGold	378	1120
401250	Morse	106	1060
401250	Morse	113	1080
401250	Morse	119	1070
401250	Morse	301	1100
401250	Morse	552	2984

401250 Morse	552	2014
401890 Northrep	112	1110
401890 Northrep	519	1110
401890 Northrep	519	1020
. . . rows omitted		



Demo 06: four table join; inner join with USING

```

select customer_id
, cust_customers.customer_name_last as "Customer"
, order_id
, prod_id
, prd_products.prod_name
from cust_customers
join oe_orderHeaders using ( customer_id )
join oe_orderDetails using ( order_id )
join prd_products using ( prod_id ) ;
  
```

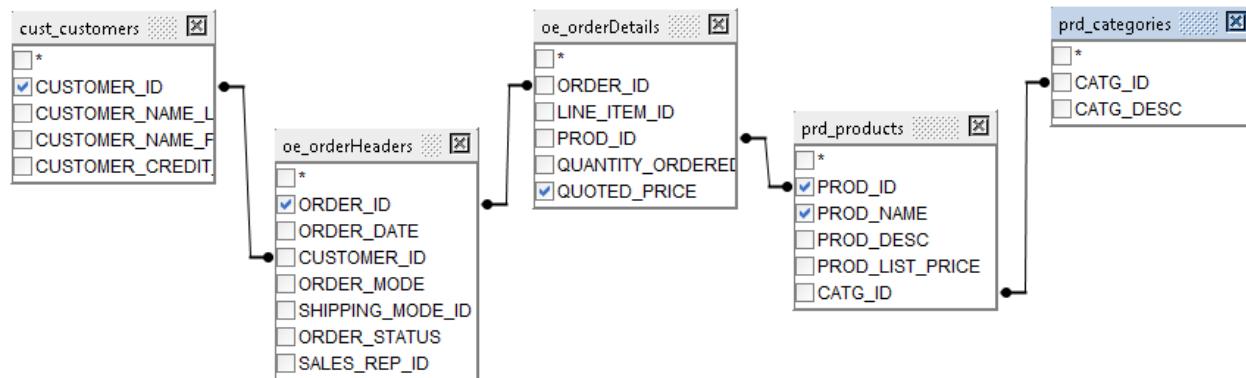
CUST_ID	Customer	ORD_ID	PROD_ID	PROD_NAME
403000	Williams	808	1000	Hand Mixer
903000	McGold	313	1000	Hand Mixer
404100	Button	303	1000	Hand Mixer
402100	Morise	115	1000	Hand Mixer
403000	Williams	808	1010	Weights
903000	McGold	551	1010	Weights
903000	McGold	550	1010	Weights
403000	Williams	528	1010	Weights
408770	Clay	405	1010	Weights
. . . rows omitted				

Demo 07: five table join including a row filter for appliances.

```

select customer_id
, order_id
, prod_id
, prd_products.prod_name
, oe_orderDetails.quoted_price
from cust_customers
join oe_orderHeaders using ( customer_id )
join oe_orderDetails using ( order_id )
join prd_products using ( prod_id )
join prd_categories using ( catg_id )
where catg_desc in ( 'APPLIANCES' ) ;
  
```

CUSTOMER_ID	ORDER_ID	PROD_ID	PROD_NAME	QUOTED_PRICE
903000	306	1125	Dryer	500
903000	306	1120	Washer	500
900300	307	1125	Dryer	450
900300	307	1120	Washer	450
400300	378	1125	Dryer	450
400300	378	1120	Washer	450
403000	412	1130	Mini Freezer	149.99
409150	415	1125	Dryer	500
404100	605	1130	Mini Freezer	112.95
403000	109	1130	Mini Freezer	149.99
404950	110	1130	Mini Freezer	149.99
402100	114	1130	Mini Freezer	125
402100	115	1120	Washer	475
403010	118	1125	Dryer	475
409030	130	1125	Dryer	500
409030	130	1120	Washer	500
404100	2503	1130	Mini Freezer	149.99
404100	2504	1130	Mini Freezer	149.99
404100	2805	1130	Mini Freezer	112.95
403000	2509	1130	Mini Freezer	149.99
403000	4511	1130	Mini Freezer	149.99
409150	3518	1125	Dryer	500
409150	2218	1125	Dryer	500
409150	223	1130	Mini Freezer	148.99
409150	718	1125	Dryer	500
409150	523	1130	Mini Freezer	149.99
403000	529	1130	Mini Freezer	149.99



5. ANSI inner join: Condition Join (the ON clause)

Sometimes we need to join two tables that have different names for the joining column. In this example we are joining the order table to the employee table. In the employee table employees are identified with an employee_id. The order table uses the term sales_rep_id to refer to the same values.

Now we use the key word ON instead of Using and we list the two columns with an equality operator. We would not really have to qualify these columns, since they have different names, but it is common, and helpful, to do so.

Demo 08: two table join- inner join with ON clause. In this case, the joining columns have different names and you cannot have a USING clause.

```

select oe_orderHeaders.order_id
, oe_orderHeaders.customer_id
, emp_employees.emp_id
, emp_employees.name_last as "SalesRep"
    
```

```
from oe_orderHeaders
join emp_employees on oe_orderHeaders.sales_rep_id = emp_employees.emp_id
order by oe_orderHeaders.order_id ;

```

ORD_ID	CUST_ID	EMP_ID	SalesRep
105	403000	150	Tuck
106	401250	150	Tuck
107	403050	150	Tuck
108	403000	155	Hiller
109	403000	155	Hiller
110	404950	155	Hiller
111	403000	150	Tuck
. . . rows omitted			

It is legal to use the On syntax when you are joining two tables which use the same column name for the joining column.

Demo 09: Inner join with ON clause - Show customers and their orders. Now you have to qualify the customer_id column because we are not using the column name join and cusomer_id appears in more than one of these tables.

```
select cust_customers.customer_id
, cust_customers.customer_name_last as "Customer"
, oe_orderHeaders.order_id
from cust_customers
join oe_orderHeaders on cust_customers.customer_id =
oe_orderHeaders.customer_id ;
```

CUST_ID	Customer	ORD_ID
400300	McGold	378
401250	Morse	506
401250	Morse	106
401250	Morse	113
401250	Morse	301
401250	Morse	119
. . . rows omitted		

You can combine both syntaxes in the same query.

Demo 10: Inner join with a USING and an ON clause- Show customers and their orders and their sales rep.

```
select customer_id
, cust_customers.customer_name_last as "Customer"
, oe_orderHeaders.order_id
, emp_employees.emp_id
, emp_employees.name_last as "SalesRep"
from cust_customers
join oe_orderHeaders using ( customer_id )
join emp_employees on oe_orderHeaders.sales_rep_id = emp_employees.emp_id ;
```

CUST_ID	Customer	ORD_ID	EMP_ID	SalesRep
401250	Morse	119	155	Hiller
401250	Morse	552	150	Tuck
401250	Morse	301	150	Tuck
401250	Morse	506	150	Tuck
401250	Morse	113	150	Tuck
401250	Morse	106	150	Tuck
401890	Northrep	519	155	Hiller
. . . rows omitted				

Demo 11: Inner join Using syntax 1 & 2- Show customers and their orders and their sales rep.

```

select customer_id
, cust_customers.customer_name_last as "Customer"
, oe_orderHeaders.order_id
, emp_employees.emp_id
, emp_employees.name_last as "SalesRep"
, emp_departments.dept_name
from cust_customers
join oe_orderHeaders using ( customer_id )
join emp_employees on oe_orderHeaders.sales_rep_id = emp_employees.emp_id
join emp_departments using ( dept_id )
order by customer_id
, oe_orderHeaders.order_id ;

```

CUST_ID	Customer	ORD_ID	EMP_ID	SalesRep	DEPT_NAME
400300	McGold	378	150	Tuck	Sales
401250	Morse	106	150	Tuck	Sales
401250	Morse	113	150	Tuck	Sales
401250	Morse	119	155	Hiller	Sales
401250	Morse	301	150	Tuck	Sales
401250	Morse	506	150	Tuck	Sales
401250	Morse	552	150	Tuck	Sales
401890	Northrep	112	145	Russ	Sales
401890	Northrep	519	155	Hiller	Sales
. . . rows omitted					

Demo 12: Rewriting the query with table aliases can make this easier to read.

```

select customer_id
, cs.customer_name_last as "Customer"
, oh.order_id
, em.emp_id
, em.name_last as "SalesRep"
, dp.dept_name
from cust_customers cs
join oe_orderHeaders oh using ( customer_id )
join emp_employees em on oh.sales_rep_id = em.emp_id
join emp_departments dp using ( dept_id )
order by customer_id
, oh.order_id ;

```

Demo 13: Five table join including a row filter for appliances using the condition join syntax. I find it easier to get all the joins written if I have a pattern. My pattern is to use 2 character table aliases. This reduces the problems with using O for the order headers table and then using D for the order details table.

When I use the condition join syntax I generally write the join as the prior table.col = this table.col.

```

select
  CS.customer_id
, OH.order_id
, OD.prod_id
, PR.prod_name
, OD.quoted_price
from cust_customers      CS
join oe_orderHeaders    OH  on CS.customer_id = OH.Customer_id
join oe_orderDetails    OD  on OH.order_id = OD.order_id
join prd_products       PR  on OD.prod_id = PR.prod_id
join prd_categories     CT  on PR.catg_id = CT.catg_id
where CT.catg_desc in('APPLIANCES');

```

There is a small difference in using a Select * from tbl1 join tbl2 depending on whether you use the column name join and the condition join. The condition join show two copies on the joining column and the column name join shows only one column in the result with the joining column.

6. NATURAL JOIN

This join can be used if the columns in common have the same name. This will join on any and all columns having the same identifier in the two tables. Therefore it creates maintenance problems if attributes are renamed or additional columns are added to the tables. Suppose we had two tables such as a customer table and a salesrep table which should be joined on a salesrepID column. But perhaps we also have attributes named City and State in each of these tables. If we did a natural join, the join would be on all of these columns. We will not use the Natural Join syntax in this class.

If you specify a natural join on tables that do not have any column names in common, then you get a cross join.

With an inner join between tables we define how the tables are to be related to each other. With a Cartesian product we join two tables but without any attempt to relate the tables to each other. Most of the time a Cartesian product is **not** what you want to use, but over the semester we will find places where the Cartesian product is helpful. The Cartesian product forms the logical basis for all of joins but you generally do not want a Cartesian product in your query.

1. The Cartesian product or cross join

Assume we have the following tables and rows. The sql is in the demos

Z_EM_Dept		Z_EM_Emp			Z_EM_EmpProj	
D_ID	D_Name	E_ID	E_Name	D_ID	P_ID	E_ID
100	Manufacturing	1	Jones	150	ORDB-10	3
150	Accounting	2	Martin	150	ORDB-10	5
200	Marketing	3	Gates	250	Q4-SALES	2
250	Research	4	Anders	100	Q4-SALES	4
		5	Bossy		ORDB-10	2
		6	Perkins		Q4-SALES	5

Suppose I want to display the name of every employee who is assigned to a department and the name of that department. You should be thinking that this is done with an inner join- and that would be correct.

Demo 01: Inner join.

```
select
    D.d_name, D.d_id
    , E.e_id, E.e_name
  from z_em_dept D
  join z_em_emp E  on D.d_id = E.d_id;
```

D_NAME	D_ID	D_ID	E_ID	E_NAME
Accounting	150	150	1	Jones
Accounting	150	150	2	Martin
Research	250	250	3	Gates
Manufacturing	100	100	4	Anders

The result contains the employee id twice- once from the department table and once from the employee table. Employee id 1 works in dept 150 which is the Accounting department.

But suppose you were in a hurry and you made an error. In the From clause you listed the two tables with a comma between them and skipped the On clause.

Demo 02: Getting a Cartesian product by default.

```
select z_em_dept.d_name, z_em_dept.d_id
    , z_em_emp.d_id, z_em_emp.e_id, z_em_emp.e_name
  from z_em_dept
    , z_em_emp;
```

D_NAME	D_ID	D_ID	E_ID	E_NAME
Manufacturing	100	150	1	Jones
Accounting	150	150	1	Jones

Marketing	200	150	1 Jones
Research	250	150	1 Jones
Manufacturing	100	150	2 Martin
Accounting	150	150	2 Martin
Marketing	200	150	2 Martin
Research	250	150	2 Martin
Manufacturing	100	250	3 Gates
Accounting	150	250	3 Gates
Marketing	200	250	3 Gates
Research	250	250	3 Gates
Manufacturing	100	100	4 Anders
Accounting	150	100	4 Anders
Marketing	200	100	4 Anders
Research	250	100	4 Anders
Manufacturing	100		5 Bossy
Accounting	150		5 Bossy
Marketing	200		5 Bossy
Research	250		5 Bossy
Manufacturing	100		6 Perkins
Accounting	150		6 Perkins
Marketing	200		6 Perkins
Research	250		6 Perkins

24 rows selected.

You get 24 rows in the result and it looks like employee 1 works in Manufacturing and in Accounting and in Marketing and in Research. But it looks like employee 2 also works in all of those departments- In fact every employee seems to work in every department including Bossy who is assigned to no department at all.

This is a legal query- it runs without error and produces a result; but the result is not very meaningful- unless you wanted to see a table of all possible assignments of employees to all departments.

What you have here is a Cartesian product of two tables; it contains each row in the first table associated with each of the rows in the second table. There is no join on matching attribute values. Therefore, we get each of the 4 department rows matched with each of the 6 employee rows for a total of 24 rows.

Demo 03: Specifying the cross join gives the same results but makes it obvious that we are intentionally doing a Cartesian product.

```
select z_em_dept.d_name, z_em_dept.d_id
, z_em_emp.d_id, z_em_emp.e_id, z_em_emp.e_name
from z_em_dept
CROSS JOIN z_em_emp;
```

Demo 04: If we added in the project table with no joins, then we get 144 rows returned.

```
select z_em_dept.d_name, z_em_dept.d_id
, z_em_emp.d_id, z_em_emp.e_id, z_em_emp.e_name
, p_id
from z_em_dept
CROSS JOIN z_em_emp
CROSS JOIN z_em_EmpProj;
```

-- a few sample rows				
Manufacturing	100	150	1 Jones	ORDB-10
Accounting	150	150	1 Jones	ORDB-10
Marketing	200	150	1 Jones	ORDB-10
Research	250	150	1 Jones	ORDB-10
Manufacturing	100	150	2 Martin	ORDB-10
Accounting	150	150	2 Martin	ORDB-10
Marketing	200	150	2 Martin	ORDB-10

Research	250	150	2 Martin	ORDB-10
Manufacturing	100	250	3 Gates	ORDB-10
Accounting	150	250	3 Gates	ORDB-10
Marketing	200	250	3 Gates	ORDB-10
Research	250	250	3 Gates	ORDB-10
Manufacturing	100	100	4 Anders	ORDB-10
Accounting	150	100	4 Anders	ORDB-10
Marketing	200	100	4 Anders	ORDB-10
Research	250	100	4 Anders	ORDB-10
Manufacturing	100		5 Bossy	ORDB-10
Accounting	150		5 Bossy	ORDB-10
Marketing	200		5 Bossy	ORDB-10
Research	250		5 Bossy	ORDB-10
. . . rows omitted				

144 rows would take a bit more than two pages to display- that is a lot of rows and these are tiny tables.

Suppose you did a cross join of the vets clients table (15 rows) with the animal table (32 rows) The result would have 480 rows. And every client would be associated with every animal; That is unacceptable.

Now consider adding the exam headers table (32 rows) to the From clause The result would now have 15,360 rows. Of those rows only 32 are meaningful (in the sense that they matching up a client to an animal to an exam).

The number of rows in the result of a Cartesian product is the product of the number of rows in each table.

Suppose our clinic had 300 clients with an average of 2 animals each and that there were an average of 2 exams per animal each year.

Our tables would have 300 rows for clients, 600 rows for animals and 1200 rows for exams. If we wanted to display exam header data for last year we would expect about 1200 rows in the result table. With a Cartesian product we would get 216,000,000 rows in the result

$$300 \times 600 \times 1200 = 216,000,000$$

Sometimes there is a value in intentionally doing a Cartesian product of two tables, but most of the time a query with a Cartesian product has been miscoded. If your query result has a lot more rows than you expected, check your joins carefully.

Error Alert

The following is not actually a Cartesian product but it is an error I see occasionally which also results in too many rows. Note the join in the ON phrase. The join test is D.d_id = D.d_id which is always true.

```
select
  D.d_name, D.d_id
, E.e_id, E.e_id, E.e_name
from z_em_dept D
join z_em_emp E on D.d_id = D.d_id;
```

Table of Contents

1. Bind variable demo	1
1.1. Define a variable, assign a value, and display	1
1.2. Using the variable in a row filter	2
2. What is happening and some syntax issues.....	3
2.1. Defining a bind variable and data types.....	3
2.2. Assigning a value to a bind variable	3
2.3. Unassigned variables.....	4
2.4. Using variables to define other variables.....	4
2.5. Displaying the bind variables	5
2.6. Using the bind variable, scope, lifespan.....	5
3. Why does this matter.....	5
4. Some advantages in using variables	6
5. How does this work (optional)	7

We can use another Oracle technique - declaring and using a variable. Most computer languages have a way to define variables. A variable is a named memory location that can store a value that we can use in our SQL queries. The name/identifier gives us a way to refer to the stored value. To use a variable:

- We need a way to define the variable giving it a name
- We need to consider the data type of the variable
- We need a way to assign a value to that variable and change the value
- We need to consider the scope of the variable- what parts of your code can refer to that variable
- We need to consider the lifetime of the variable- how long does it keep its value

We will limit this discussion to assigning a value to a user-defined variable, here called a bind variable, and then using that variable in an SQL statement.

The variables we are considering here are scalar variables- which means they hold a single data value.

1. Bind variable demo

We will start with a demo of a bind variable and then discuss what it is and why you might want to use these.

1.1. Define a variable, assign a value, and display

Demo 01: Run the following steps in SQL*Plus (SQL Developer does not do as well with bind variables)

```
variable lst_prc number
execute :lst_prc := 25
print :lst_prc
select :lst_prc from dual;
```

The following screen shot shows the results of running the above commands in SQL*Plus.

```
SQL> variable lst_prc number
SQL> execute :lst_prc := 25
PL/SQL procedure successfully completed.
SQL> print :lst_prc
LST_PRC
-----
25
SQL> select :lst_prc from dual;
:LST_PRC
-----
25
```

We are defining a numeric variable, giving it a value and then printing it with a print command and also displaying it with a select query.

1.2. Using the variable in a row filter

Now try the following SQL statement.

Demo 02: Use the bind variable in the Where clause

```
select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = :lst_prc
;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
1080	Cornpopper	25

The value that was assigned to the bind variable is used in the SQL statement.

Change the bind variable and rerun the sql statement in the sql buffer. We are using the same query with a different value being supplied to the row filter. The slash command in SQL*Plus reruns the sql statement that is in the sql buffer. Even though we issued a command to set a variable, the SQL buffer is not changed and you can rerun the SQL command with the slash.

Demo 03:

```
execute :lst_prc := 29.95
/
```

PROD_ID	PROD_NAME	prod_list_price
1030	Basketball	29.95
4576	Cosmo cat nip	29.95
4577	Cat leash	29.95

It might not be obvious but you are rerunning the same SQL statement that you ran before, with a different value being used in the filter. The work that Oracle has to do to prepare your SQL statement to be run does not have to be repeated when you use the bind variables.

Demo 04: We could set a variable for a tax rate and then use it in a calculation

```
variable sales_tax_rate number

execute :sales_tax_rate := 0.095;

select prod_id
, quantity_ordered as Quantity
, quoted_price as Price
, quantity_ordered * quoted_price as AmtDue
, quantity_ordered * quoted_price * :sales_tax_rate as SalesTaxDue
from oe_orderDetails
where rownum <=5
;
```

PROD_ID	QUANTITY	PRICE	AMTDUE	SALESTAXDUE
1020	5	12.95	64.75	6.15125
1110	1	49.99	49.99	4.74905
1020	3	12.95	38.85	3.69075
2747	3	12.95	38.85	3.69075
1080	2	22.5	45	4.275

2. What is happening and some syntax issues

A variable is a named location in memory that can hold a data value. The bind variable is declared at the SQL*Plus client level. The variable statement declared a name and a data type for the variable. This is not an SQL statement; it is an SQL*Plus command- note that it did not end with a semicolon.

The next statement used an execute command to assign a value to that bind variable. The message that was provided indicated that this ran a PL/SQL procedure. We have not talked about PL/SQL yet but it is a programming language provided by Oracle. Two things to note about the syntax:

- in this statement we had to prepend a colon to the variable name so that it is not confused with other identifiers that we might have.
- the assignment operator in PL/SQL is written with a colon and an equals character :=

Now that the variable has a value, we can print it. Print is a SQL*Plus level command; we do not need a semicolon at the end of this command (it is not an SQL statements) and we do not need the colon in front of the variable at the SQL*Plus level but we can include it.

Then we use that variable (with the colon) in an SQL statement.

So this variable is being passed back and forth between SQL*Plus and the Oracle engines which run SQL and PL/SQL. The variable could also get its value from an application program that connects with the database.

2.1. Defining a bind variable and data types

To declare a variable you need to supply a name and a data type. The command to define a variable is variable.

The data types can be one of these (there are others we won't discuss); date is not an allowed type here.

```
number
char
char(99)
varchar2(99)
```

You cannot provide a precision for number; you must provide a length for varchar2; you may supply a length for char- it defaults to one character.

By itself, the command variable will show you the names and data types of your bind variables.

2.2. Assigning a value to a bind variable

If you need to assign a value to a single bind variable use the execute command as shown above. This has the advantage of not changing your SQL buffer contents.

If you have several bind variables to assign, you can use the following syntax.

Demo 05: Using an anonymous block to assign a value to the variable.

```
variable lst_prc number
variable catg  varchar2(10)

begin
  :lst_prc := 15.987;
  :catg := 'APL';
end;
/
```

The syntax here is that we start with the keyword begin. Each assignment is done on a new line which ends with a semicolon. Then we finish with the word end followed by a semicolon and a slash to run this section of code. However this does replace the SQL buffer with this PL/SQL block of code.

```
select :lst_prc, :catg from dual;
```

```
:LST_PRC :CATG
```

```
-----
```

```
15.987 APL
```

Demo 06:

```
variable target  varchar2(20)
exec :target := 'Shingler Hammer';

select prod_id, prod_list_price, prod_name
from prd_products
where prod_name = :target;
PROD_ID PROD_LIST_PRICE PROD_NAME
----- -----
      5005          45 Shingler Hammer
```

2.3. Unassigned variables

Demo 07: Using a variable that has been declared but not assigned a value gives us a null

```
variable quantity number
print :quantity
QUANTITY
-----
select :quantity from dual;
:QUANTITY
-----
```

2.4. Using variables to define other variables

Demo 08: You can use one variable to define another.

```
variable varb number;
variable varb_2 number;

exec :varb := 45 * 3;
exec :varb_2 := :varb +100;

select :varb, :varb_2 from dual;
:VARB    :VARB_2
----- -----
     135      235
```

Demo 09: This builds up a variable to use with a Like operator.

```
variable target  varchar2(20);
variable target2  varchar2(20);
exec :target := 'Hammer';
exec :target2 := '%' || :target || '%';

select prod_id, prod_list_price, prod_name
from prd_products
where prod_name Like :target2;
PROD_ID PROD_LIST_PRICE PROD_NAME
----- -----
      5002          23 Ball-Peen Hammer
      5004          15 Dead Blow Hammer
      5005          45 Shingler Hammer
```

2.5. Displaying the bind variables

You can use the print command to show a single bind variable or use the print command by itself to show the values of all of your bind variables

```
variable
variable sales_tax_rate
datatype NUMBER

variable lst_prc
datatype NUMBER

variable quantity
datatype NUMBER

variable catg
datatype VARCHAR2(10)
```

```
print
SALES_TAX_RATE
-----
.095
```

```
LST_PRC
-----
15.987
```

```
QUANTITY
-----
```

```
CATG
-----
APL
```

2.6. Using the bind variable, scope, lifespan

When you use the bind variable in an sql statement, remember to refer to it with a leading colon. You can use the bind variable in Where clauses, Select clauses, calculated columns, Insert statements, etc.

You cannot use bind variables for things such as table names or columns names- only for the literals.

Although you won't see any difference in speed of execution with our small tables and single user demo accounts, the difference in a production system can be significant when you use variables.

Here we are assigning values to the variables; commonly the values for these variables would come from the application level programs.

Lifespan: The bind variables are part of your session; they disappear when you close your session. If there is a need to keep the values of these variables, you should store them in a table created for that purpose.

Scope: Other people who are running other sessions cannot see your bind variables.

3. Why does this matter

Suppose you have an application running where users were running that sql statement multiple times with different literals for the price.

```

select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = 25;

select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = 500;

select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = 150;

```

The way Oracle works is that each of these SQL statements would be seen as a brand new statement and Oracle would do a lot of work for each of these to create an execution plan to run this efficiently. This takes time to do. Also each of these SQL statements in its parsed form would be stored in memory (in the shared pool) in case someone needed to use it again; if the statement were issued again then Oracle has less work to do in terms of setting the statement up to run. But since we have different numbers in each statement, Oracle sees these as different statements.

But we can abstract this a bit- these are all the same SQL statement- they just have different numbers for the filter test. We can let Oracle build the execution plan for the SQL statement with the bind variable and set that up in memory and when the statement is executed, the bind variable is replaced with its current value. This is more efficient.

4. Some advantages in using variables

One advantage in the use of variables in your queries is when you need to test your queries for different values for the filter tests.

1) Suppose you have a long complex query that needs to test the product category value several times. If you store the value for the product category in a variable, then you can change that variable one time and have all references to it in the query change. If you wrote the literal value of the category in the query, then you have to find and change each of those values.

2) Suppose you had a query that needs to filter on the list price of an item and also filter on the quantity of the item. And your first test was

```
Where prod_list_price > 50 and quantity > 50;
```

If you then need to change the filter for quantity to 60 (and remember you have this test several times in your query) you have to be certain to change only the quantity test and not the price test.

Setting up two variables would make maintenance easier and less error prone.

```
Where prod_list_price > :priceLimit and quantity > :quantityLimit
```

3) Another example you will have in assignments is a filter that tests a date value compared to the current date. We have not discussed date expressions to any detail yet, but suppose you have this test. We want to find orders with the same month number as the current month. So if we run this query in August, it will find orders placed in August, ignoring the month and year.

```
Where extract ( month from ord_date ) = extract ( month from sysdate )
```

How do you test that query for validity if you need to be able to run in at a later time (remember the actual test may be much more complex).

Suppose you set up a variable.

```
Variable dtm varchar2(20);
exec :dtm := sysdate;
```

And use that in your query

```
Where extract ( month from ord_date) = extract ( month from to_date(:dtm))
```

Now you can change the value of the variable to other date values and run the same query using a different "current date".

You might not have noticed that I used to_date(:dtm) in that Where clause. I could also use cast(:dtm as date). The reason for this is that :dtm was defined as a varchar2 type. The exec took the date value returned by sysdate, in its default format as a string and assigned that string to :dtm. In order to get the month from that string we have to cast it back to a date value. (I know- it is annoying to have to do that but if you don't ,you get an error message and that is even more annoying.)

4) Using variables can help you think more methodically about your queries. With the previous example you could write:

```
Variable dtm varchar2(20);
exec :dtm := sysdate;
Variable curr_month number
exec :curr_month := extract ( month from to_date(:dtm));
```

and then use the following filter which is closer to the wording of the task.

```
Where month(ord_date) = :curr_month
```

You can also display the value of :curr_month to check that part of the calculation.

5. How does this work (optional)

Parsing an SQL statement makes it ready to be executed. One of the concerns with executing a query against a database is efficiency and the parser tries to find the most efficient technique for executing that query. The first time a query is parsed is called a hard parse and it produces an execution plan. Parsing takes time and computer resources. So the execution plan for the parsed query is stored in memory. If you run the identical query again, the execution plan can be reused if it is still in memory.

But in order to use the execution plan again, the second query has to be identical to the first (even white space counts). Suppose you have a query that counts how many cats we saw at the vet clinic last month. Then you want to run a query that counts how many dogs we saw at the vet clinic last month. That is not the same query and a new execution plan is created.

Now suppose you have a query that counts how many animals we saw last month of type XXX where you will fill in the value for XXX every time you run the query. There can be one execution plan created and reused. This is not going to save any time if the query plan is used only once- but it will if you are running several queries of this type.

The way to handle this is with bind variable- in the discussion above XXX represented the bind variable. Your code has to set up the bind variable, give it a value, and then use it. The bind variable is a place holder for values to be passed during execution.

If you think about the most efficient way for this query to be expected this might depend on the data we have. Suppose that we see mostly cats and dogs and pretty much in the same numbers. But there are a few examinations of hedgehogs. When you are searching for cats or dogs, the use of an index might not be the most efficient way; but you are searching for a hedgehog, then an index will help. So what is the execution plan that is set up? The execution plan is set up with the first query and what Oracle does is sneak a peek at the value of the bind variable the first time it is used and uses that. So if your data has the same relative distribution things will work out ok; if you have these wide variations in distribution this might not be a good way to handle things. Some people avoid bind variables in these cases, forcing the parser to create a new execution plan.

With Oracle 11g there is a new approach. With Oracle if there is an execution plan with a bind variable, the Oracle parser marks it and keeps track of the bind variable values being used to decide if it should consider using a different execution plan. This is done automatically by Oracle 11g.

Table of Contents

1. What is an Intrinsic Function.....	1
2. Demonstrating Functions.....	1
2.1. Nesting Functions.....	2
3. Developing an expression using functions.....	2

1. What is an Intrinsic Function

One of the strengths of a dbms is the number and variety of single-row supplied functions. Functions do tasks such as rounding numbers to the desired accuracy, locating a string pattern within a target string, and dealing with nulls. A function returns a value and we can use that value in a Select clause, in a Where clause- and later we will use them in action statements.

One of the things you have to consider is that these functions are Oracle supplied functions; these are often called intrinsic functions since they are built into the dbms. Other dbms will have similar functions but they might not use the same function name and the same arguments. When you start using supplied functions, your code is less portable.

The value returned by a function can be passed to another function; this is called nesting functions and we will see examples of this as we work through various functions.

To use a function you need to know the name of the function and the type of arguments it needs. You also need to know the data type of the value returned by the function. You also need to consider (or test) what happens with a function if you pass it a null as one of the arguments, or pass it a value that is not appropriate for the function.

A function does not change the value of the arguments that were passed to it. For example, if you use the round function to round a numeric column in a Select query, the function does not change the data stored in the table; it simply returns the rounded version of the numbers.

These functions are referred to as **single-row functions**. They take one or more arguments, typically from one row in a table, and return a value. When the function is used in a Select statement, it returns one value for each row in the table. We will later discuss multi-row functions which take a group of rows and return a single value for the group.

We will not cover every function and I do not expect you to remember them all and all of their variations. But you should be able to quickly find and use functions from these documents and your book in doing assignments and during exams.

We will categorize the functions in the areas of

- functions that work with numbers
- functions that work with strings
- functions that work with regular expressions
- functions that convert data from one format to another
- functions that do some sort of logic
- functions that work with temporal values

Some functions are hard to classify and may be discussed in more than one category.

2. Demonstrating Functions

Functions are best explained with examples. For many of the functions, I will show you examples of the use of the function with literal values- such as the Upper function which accepts a string and returns the string with all letters converted to upper case letters. You would not normally pass a literal string to this function but it is easier to show you examples using this technique.

You can test the function examples by running them against the Dual table. This demonstrates the Upper function. The SQL*Plus client will display the expression as the column header if there is no column alias.

```
select UPPER( '      This is MY sTrInG') from Dual;
-----  
UPPER('THISISMYSTRING')
-----  
THIS IS MY STRING
```

A more typical use is to change the display with a select that gets data from a table. Note the column aliases.

```
select an_id, upper(an_type), upper(an_type) as an_type
from vt_animals
where rownum < 6;
```

AN_ID	UPPER(AN_TYPE)	AN_TYPE
15165	DOG	DOG
15401	LIZARD	LIZARD
19845	DOG	DOG
21001	HEDGEHOG	HEDGEHOG
21003	DOG	DOG

2.1. Nesting Functions

This is an example of a nested function. This is an expression that uses two functions; one function returns a value that is used as an argument to the second function. The value that is returned by the upper function is passed to the Ltrim function which removes any spaces at the Left edge of the string.

```
select LTRIM(UPPER( '      This is MY sTrInG')) from Dual;
-----  
LTRIM(UPPER('THIS
-----  
THIS IS MY STRING
```

When you work with functions, as for the assignments, you need to work methodically. This example will make more sense after you work through the other documents in this unit. So make a note to come back to this.

3. Developing an expression using functions

Suppose, for some reason, we want to take the product descriptions in the Altgeld mart products table, throw away all of the spaces in the description- including the spaces between words, and then get the first 10 characters in the description in uppercase letters. You could use this expression in a select statement that you run against that products table. Before you start to think about the expressions, think about possible description that you should test and what the results should be:

- A description that is longer than 10 characters, after removing blanks:
'Bird cage- simple; wire frame two feeder trays'
Should return: 'BIRDCAGE-S'
- A description that is shorter than 10 characters, after removing blanks:
'Red Mixer'
Should return: 'REDMIXER'
- A description that has no blanks:
'Discovery'
Should return: 'DISCOVERY'
- A description that is all blanks:
' '
Should return: " (this is an empty- zero-length- length string)

- A description that is null:

```
null
```

```
Should return: null
```

Remember that Oracle treats zero length strings and nulls as the same.

When you first start to think about creating these expressions, you probably don't think about all of these possibilities- but they could happen. So how would you test your expression? Instead of running it against the products table, set up a variable; give it the value to be tested and run your expression against the variable. Then you can change the value in the variable and rerun the expression.

Suppose we did this with the nested function example:

```
select LTRIM(UPPER('This is MY sTrInG')) from Dual;
```

We could do:

```
variable v varchar2(50);
exec :v := 'This is MY sTrInG';
select LTRIM(UPPER(:v)) from Dual;
LTRIM(UPPER(:V))
-----
THIS IS MY STRING
```

```
exec :v := 'What about digits 2 3 4 and punctuation !#%$';
select LTRIM(UPPER(:v)) from Dual;
LTRIM(UPPER(:V))
-----
WHAT ABOUT DIGITS 2 3 4 AND PUNCTUATION !#%$
```

Back to our function expression: This is going to use three functions that are discussed in this unit's notes. The functions are

- UPPER which returns the upper cased string
- REPLACE which changes a character to another character
- SUBSTR which returns a set number of characters from the front edge of the string

I am going to start with a variable set up as a varchar2(50) and assign a literal with a number of blanks in it.

```
variable v varchar2(50);
exec :v := 'Bird cage- simple; wire frame two feeder trays';
```

Now we can start working on the function expression. Do this one step at a time- it is quicker and easier that way.

First get rid of the blanks using replace; I am replacing each space (' ') with nothing ("").

```
variable v varchar2(50);
exec :v := 'Bird cage- simple; wire frame two feeder trays';
select replace(:v, ' ', '') from dual;
REPLACE(:V,'','')
-----
Birdcage-simple;wireframetwofeedertraysQueries:thebestbookever
```

Now we can change this to upper case letters. Use the replace function expression as an argument to the upper function.

```
variable v varchar2(50);
exec :v := 'Bird cage- simple; wire frame two feeder trays';
```

```
select replace(:v, ' ', '') from dual;
select upper(replace(:v, ' ', '')) from dual;
-----  
BIRDCAGE-SIMPLE;WIREFRAME TWO FEEDER TRAYS
```

Now we want the first 10 characters; that can be done with the Substr function.

```
variable v varchar2(50);
exec :v := 'Bird cage- simple; wire frame two feeder trays';
select substr(upper(replace(:v, ' ', '')), 1,10) from dual;
-----  
SUBSTR(UPP
-----  
BIRDCAGE-S
```

So far this looks good. But test the other values we considered at the start of this discussion.

```
exec :v := null;
```

If you are happy with the functions then you can use that expression in a query against the products table, replacing the :v variable with the column name.

```
select substr(upper(replace(prod_desc, ' ', '')), 1,10)
from prd_products;
```

This approach lets you control the data you need to test; lets you work a little at a time which helps with the parentheses problem and helps you think methodically.

Table of Contents

1. Math functions: Abs, Power, Sign, Sqrt.....	1
2. Rounding functions	2
3. Random values- pseudo-random values.....	4
3.1. DBMS_RANDOM	4

I am starting with the numeric functions. the function we use the most is the Round function, following by Floor and Ceiling. We also use the Rand to generate random numbers.

These demos are all run with literals- such as shown here. Instead of including these selects, I have used the expression as the column header for the result. I have adjusted column widths to fit the page.

SQRT (64)

8

1. Math functions: Abs, Power, Sign, Sqrt

I am discussing Abs, Sign, Power, Sqrt, and Mod here because they are pretty simple. Oracle also includes functions to do trig calculations; we won't be using those.

Demo 01: **ABS** returns the absolute value of the argument

ABS (12)	ABS (-12)	ABS (0)	ABS (125.85)	ABS (-74.15)
12	12	0	125.85	74.15

Demo 02: **SIGN** returns only 0, 1, or -1;

if the argument is 0, sign returns 0;
 if the argument is positive, sign returns +1;
 if the argument is negative, sign returns -1.

SIGN (0)	SIGN (12)	SIGN (-12)	SIGN (12.32)
0	1	-1	1

Demo 03: **POWER** (a, b) is used to calculate a raised to the b power

POWER (3, 2)	POWER (10, 3)	POWER (10, -3)	POWER (4.5, 3.2)
9	1000	.001	123.106234

Demo 04: **SQRT** returns the square root of its argument; the argument must be a non-negative number or you get an error

SQRT (64)	SQRT (68.56)	SQRT (ABS (-679.34))	SQRT (0)
8	8.28009662	26.0641516	0

Demo 05: **MOD** (a, b) returns the remainder after a is divided by b. Mod makes the most sense when used with integers.

If mod (a, 1) = 0 then a is an integer

If mod (a, 2) = 0 then a is even;

If mod (a, 2) = 1 or -1 then a is odd

Note the result when the second argument is 0.

If you have a time duration expressed as minutes, you can determine the number of full hours as `trunc(theTime/60)` and minutes as `mod(theTime, 60)`

<code>MOD(18, 12)</code>	<code>MOD(18, 6)</code>	<code>MOD(6, 18)</code>	<code>MOD(18.6, 12)</code>	<code>MOD(18, 6.4)</code>	<code>MOD(18.6, 6.4)</code>
6	0	6	6.6	5.2	5.8
<code>MOD(10, 3)</code>	<code>MOD(-10, 3)</code>	<code>MOD(10, -3)</code>	<code>MOD(-10, -3)</code>		
1	-1	1	-1		
<code>MOD(128, 1)</code>	<code>MOD(128.002, 1)</code>	<code>MOD(128.00, 1)</code>	<code>MOD(35, 2)</code>	<code>MOD(368, 2)</code>	
0	.002	0	1	0	
<code>MOD(0, 3)</code>	<code>MOD(10, 0)</code>	<code>MOD(0, 0)</code>			
0	10	0			

2. Rounding functions

The next set of numeric functions returns a value close to the argument.

Demo 06: **ROUND** returns the number rounded at the specified precision. The precision defaults to 0.
You can have a negative precision. Round rounds up at .5

<code>ROUND(45.678, 0)</code>	<code>ROUND(45.2, 0)</code>	<code>ROUND(46.5, 0)</code>	<code>ROUND(45.678, 2)</code>
46	45	47	45.68
<code>ROUND(45.55, 1)</code>	<code>ROUND(-46.5, 0)</code>	<code>ROUND(345.67, -2)</code>	<code>ROUND(45, -2)</code>
45.6	-47	300	0

Demo 07: **TRUNC** returns the number truncated at the specified precision

<code>TRUNC(45.678, 0)</code>	<code>TRUNC(45.678, 2)</code>	<code>TRUNC(45.55, 0)</code>	<code>TRUNC(-45.55, 0)</code>
45	45.67	45	-45
<code>TRUNC(345.67, -2)</code>	<code>TRUNC(399.99, -2)</code>	<code>TRUNC(399.99, -5)</code>	
300	300	0	

Demo 08: **CEIL** returns the smallest integer greater than or equal to the argument

<code>CEIL(10)</code>	<code>CEIL(10.2)</code>	<code>CEIL(-10.5)</code>	<code>CEIL(10.8)</code>
10	11	-10	11

Demo 09: **FLOOR** returns the largest integer less than or equal to the argument

<code>FLOOR(10)</code>	<code>FLOOR(10.5)</code>	<code>FLOOR(-10.5)</code>	<code>FLOOR(10.8)</code>
10	10	-11	10

Comparison of Round, Ceiling, Floor, and Truncate. Suppose you had calculated a numeric value and you needed to display it with 2 digits after the decimal. How would you want to display the value 25.0279? This often comes up in issues such as calculating sales tax- do we round up or round down? The decision about which one is correct is a business decision- not a programming decision. But we can use the following small table to see our choices in terms of SQL

Demo 10: Comparison of Round, Ceil ,Floor and Trunc

```
Create table z_tst_numerics ( id integer, val_1 float);
insert into z_tst_numerics values (1, 25.0034);
insert into z_tst_numerics values (2, 25.0079);
insert into z_tst_numerics values (3, 25.0279) ;
insert into z_tst_numerics values (4, 25.4239) ;
insert into z_tst_numerics values (5, -25.0279) ;
insert into z_tst_numerics values (6, -25.4239) ;
```

Demo 11: Comparison query

```
Select id
, val_1
, ROUND(val_1,2) as "round"
, TRUNC(val_1,2) as "trunc"
, CEIL(val_1 * 100.00)/100.00 as "ceil"
, FLOOR(val_1* 100.00)/100.00 as "floor"
From z_tst_numerics
;
```

ID	VAL_1	round	trunc	ceil	floor
1	25.0034	25	25	25.01	25
2	25.0079	25.01	25	25.01	25
3	25.0279	25.03	25.02	25.03	25.02
4	25.4239	25.42	25.42	25.43	25.42
5	-25.0279	-25.03	-25.02	-25.02	-25.03
6	-25.4239	-25.42	-25.42	-25.42	-25.43

Demo 12: Using a negative precision of -1 with round gives you values rounded off to the nearest 10 dollars.

```
Select
quoted_price as price
, round(quoted_price, -1) as Price_10
, round(quoted_price, -2) as Price_100
From oe_order_details
;
Selected rows
```

PRICE	PRICE_10	PRICE_100
15	20	0
14.5	10	0
4.25	0	0
75.99	80	100
25.5	30	0
150	150	200
149.99	150	100
12.95	10	0
300	300	300
225	230	200

3. Random values- pseudo-random values

Strictly speaking the technique to produce random values in Oracle is not a function, but it is an Oracle supplied technique and it makes sense to discuss it here.

We use the term random numbers to refer to a series of numbers that are produced by some process where the next number to be generated cannot be predicted. We also assume with random numbers that if we generate a set of 10,000 random integers between 1 and 10, then each of the values 1 through 10 should occur approximately the same number of times. Of course, we are not quite saying what "approximately" the same number of times really means.

There are discussions about what "truly" random means and for some applications there are random numbers that are based on using atmosphere noise or the degradation of atomic particles.

For many programming purposes, pseudo-random numbers are good enough. Pseudo-random numbers are generated by an algorithm which is provided with a seed value to get the series of values started. If you give the generator the same seed it will produce the same series of numbers. This is very good for test purposes, so that you can run your program repeatedly against the same series of test data. The pseudo-random number generators also work without a seed, in which case the seed is generally based on the computer time- that means each time you run the generator you get a different set of numbers. This is also good for testing programs.

Many pseudo-random number generators allow you to also specify the range of values you want as output- for example integers between 1 and 100, or floating point numbers between -10 and +10.

One use for random numbers is to create test values to insert into a table. Generating 100 or 100,000 rows of random test data can be handled with random functions.

3.1. DBMS_RANDOM

DBMS_RANDOM is a supplied Oracle package for generating pseudo random numbers or strings. For the rest of this discussion, I'll use the term "random numbers" to refer to pseudo-random numbers generated by the Oracle DBMS_RANDOM package.

Demo 13: Let's start with an example. Generating random values - run this several times

```
Select dbms_random.value  
From dual;
```

These are some typical values returned by the query- the query returns a single value.

```
0.87564450157828969250026606645729508674
```

```
0.03293085802772244969039334971831440769
```

```
0.5715936725133017963338688533991109245
```

```
0.35305488184218107629434141212297758639
```

```
0.17470228115667371240150793918577648334
```

```
0.23249386861943613971605269193433067208
```

You can specify a Low and a High parameter for a range of values. The values will range from Low (and can equal Low) up to High (but not equal to High)

Demo 14: Generating random values between 10 and 25 and converting to an integer

```
Select Floor(dbms_random.value(10, 26))
From dual;
```

I ran that query 10 times and got the following values.

```
21
12
21
24
17
12
14
19
14
22
```

There are 16 values between 10 and 25 (including the endpoints). If I use the expression in the previous demo to generate 16000 values, I would expect to get each values approximate 1000 times. Using a loop technique I generated those numbers and counted them. The results were are shown here. Note that the number 26 did not occur at all. Also note that the values 10 and 25 occurred about the same number of times.

number	10	11	12	13	14	15	16	17
count	1028	1046	10006	10004	990	953	1023	1035

number	18	19	20	21	22	23	24	25
count	993	966	978	1070	997	980	957	974

You may see people using the round function - such as round(dbms_random.value(10,25)). Using that expression and generating 16000 values, I get a count of 544 for value 10 and 522 for value 25 and the other values were all between 1032 and 1109. This certainly does not give the numbers at the end points the same chance as the others.

Demo 15: Random values from -10 to +10

```
Select floor(dbms_random.value(-10, +11 ))
From dual;
```

Demo 16: Random values from 0.05 to 0.08

```
Select round(dbms_random.value(50, 81)/1000 , 2)
From dual;
-- several runs
```

```
0.07
0.05
0.08
0.08
```

Demo 17: I could assign the random value to a variable

```
variable num number;
exec :num := dbms_random.value;
print :num;
-----
```

```
NUM
-----
.715401877
```

Table of Contents

1. Capitalization.....	1
2. Padding and Trimming strings.....	2
3. Parts of strings and matches within a string	4
4. Changing the string contents	6
5. Misc string functions	7

Oracle has many string functions. The ones we will discuss are:

Length	InitCap	Trim	Substr	Translate.
Upper	RPad	LTrim	Instr	Chr
Lower	LPad	RTrim	Replace	Ascii

Many of these functions have optional arguments to let you return a more precise value. The examples will start with the simpler versions and then add the additional arguments. Particularly with string functions we need to consider what happens if we "go off the end" of a string. The examples will show some of these situations.

Many of the expressions below are concatenated with an 'X' character to let you see any trailing blanks returned.

1. Capitalization

UPPER, LOWER, INITCAP return the string in a specific case pattern. This is often used in a Where clause:

WHERE UPPER(last_name) = 'KING' when you want to do a case-independent match

The InitCap pattern displays the string in a mixed case format that does not always refer to the way that a human would case a name. If the case pattern of people's name is critical to your business, then you should store the name in the pattern that a person uses for their name.

Demo 01:

```
select Upper( 'This is MY sTrInG'), Upper( '50 Phelan Ave SF 94112')
from dual;
-----
```

UPPER('THISISMYST	UPPER('50PHELANAVESF94
-----	-----
THIS IS MY STRING	50 PHELAN AVE SF 94112

```
select Lower( 'This is MY sTrInG') , Initcap ( 'This is MY sTrInG')
from dual;
-----
```

LOWER('THISISMYST	INITCAP('THISISMY
-----	-----
this is my string	This Is My String

```
select Initcap( 'MACDONALD McDonald o''reilly') , Initcap('e. e. Cummings')
from dual;
-----
```

INITCAP('MACDONALDMCDONALDO''R	INITCAP('E.E.C
-----	-----
Macdonald Mcdonald O'Reilly	E. E. Cummings

Demo 02: This is a query with a wildcard test and the Upper function.

```
select prod_id, prod_name
from prd_products
where upper(prod_desc) like '%BIRD%';
```

PROD_ID	PROD_NAME
1140	Bird cage- simple
1141	Bird cage- deluxe
1142	Bird seed

2. Padding and Trimming strings

RPAD, LPAD returns a string of a specified length

RPAD (strExp1, len, strExp2) returns the value of the strExp1 padded with the characters in strExp2 to length len. This could result in a truncated string. StrExp2 defaults to the space character. StrExpr2 could be a column name or other string expression.

Demo 03:

```
select
  RPAD ( 'MyString', 20) || 'X'
, RPAD ( 'MyString', 4) || 'X'
, RPAD ( 'MyString', 20, '-*') || 'X'
, LPAD ( 'MyString', 20, ':')
from dual;
-----
```

RPAD('MYSTRING',20)	RPAD(RPAD ('MYSTRING',20,'-	LPAD ('MYSTRING',20,'
MyString	X	MyStX	MyString-----X
			:::::::::::::MyString

```
select Lpad(25.95, 10, 0)      from dual;
-----
```

LPAD(25.95	
	0000025.95

In this case Oracle silently casts the number to a string to do the LPad since LPad expects a string.

Demo 04:

```
select rpad( lpad ( name_last, 20 + length (name_last)/2, '.'), 40,'.')
from emp_employees where rownum < 5;
-----
```

RPAD(LPAD(NAME_LAST,20+LENGTH(NAME_LAST)	
.....King.....	
.....Harts.....	
.....Koch.....	
.....Green.....	

RTRIM, LTRIM removes leading/trailing characters

RTRIM (strExp1, strExp2) returns the value of strExp1 with the specified characters removed from the right-hand end of the string. strExp2 defaults to the blank character.

The second argument can be a list of characters to trim from the end of the string

Demo 05:

```
select 'X' || RTRIM ( '    This is my string    ') || 'X' from dual;
-----
```

'X' RTRIM('THISISMYST	
X This is my stringX	

This removes \$ and * from the Right. Left or both sides of the string.

```
select
    RTRIM ( '***$$*$This ** is $50.00***', '*$')
, LTRIM ( '***$$*$This ** is $50.00***', '*$')
, RTRIM(LTRIM ( '***$$*$This ** is $50.00***', '*$'), '*$')
from dual;
```

RTRIM('***\$\$*\$THIS**IS\$5-----	LTRIM('***\$\$*\$THIS**I-----	RTRIM(LTRIM('***\$-----
\$*\$This ** is \$50.00	This ** is \$50.00*	This ** is \$50.00

TRIM lets you trim from both ends with one function call. Trim takes a single character only. The plain Trim removes leading and trailing blanks.

Demo 06:

```
select TRIM( ' This is my string ') || 'X' from dual;
TRIM('THISISMYSTRI-----
```

This is my stringX

```
Select
    TRIM( LEADING ' ' FROM ' This is my string ') || 'X'
, TRIM( TRAILING ' ' FROM ' This is my string ') || 'X'
, TRIM( BOTH ' ' FROM ' This is my string ') || 'X'
From dual;
```

TRIM(LEADING ''FROM'THI-----	TRIM(TRAILING ''FROM'T-----	TRIM(BOTH ''FROM'TH-----
This is my string	X	This is my stringX

```
select TRIM( BOTH '**' FROM '***$$*$This ** is $50.00***') from dual;
TRIM(BOTH '**'FROM'***$-----
```

\$**\$This ** is \$50.00

You will often want to use Upper and Trim when testing data values. Data values often have case issues and may contain trailing or leading blanks. There also are issues in dealing with Char and Varchar data with trailing blanks.

Demo 07:

```
Create table ZStrings (col_id number, col_varchar varchar2(10), col_char
Char(10));
Insert into ZStrings values (1, 'CAT', 'CAT');
Insert into ZStrings values (2, ' CAT', ' CAT');
Insert into ZStrings values (3, 'CAT ', 'CAT ');
Insert into ZStrings values (4, ' CAT ', ' CAT '');
```

Demo 08:

```
select col_id,
'x' || col_varchar || 'x' AS Varchardata,
'x' || col_char || 'x' AS chardata
from ZStrings
; 
```

COL_ID	VARCHARDATA	CHARDATA	
1	xCATx	xCAT	x
2	x CATx	x CAT	x ← has leading blank
3	xCAT x	xCAT	x ← has trailing blank
4	x CAT x	x CAT	x ← has leading & trailing blank

If we run this with a filter of `WHERE colChar = 'CAT'`, then rows 1 and 3 are returned. The literal is end padded with blanks to 10 characters.

Demo 09:

```
select col_id, col_char
from zStrings
where col char = 'CAT';
```

COL_ID	COL_CHAR
1	CAT
3	CAT

If we run this with a filter of `WHERE colVarChar = 'CAT'`, then only the first row is returned. The literal is not padded with the varchar test.

Demo 10:

```
select col_id, col_varchar
from zStrings
where col varchar = 'CAT';
```

COL_ID	COL_VARCHAR
1	CAT

Use `WHERE Trim(colVarChar) = 'CAT'` to avoid this issue if the leading and trailing blanks are not to be considered important.

Demo 11:

```
select col_id, col_varchar
from zstrings
where Trim(col VarChar) = 'CAT';
```

COL_ID	COL_VARCHAR
1	CAT
2	CAT
3	CAT
4	CAT

3. Parts of strings and matches within a string

SUBSTR –returns part of a string. `SUBSTR(strExp1, pos_start, len)` returns part of strExp1, starting from position pos_start and continuing for len characters. If len is omitted, the trailing end of the string is returned. If pos_start is 0, it is treated as 1. If len is negative, a null is returned.

Demo 12:

SUBSTR('ABCDEFGHIJK', 5) returns EFGHIJK
From position 5 to the end of the string

SUBSTR('ABCDEFGHIJK', 5, 3) returns EFG
From position 5 for a length of 3 characters

SUBSTR('ABCDEFGHIJK', 5, 60) returns EFGHIJK
It is not an error to ask for too many characters

SUBSTR('ABCDEFGHIJK', 25) returns no value is returned since we are past the end of the string

SUBSTR('ABCDEFGHIJK', -5) returns GHIJK
-- a negative value for start, means you are counting from the end of the string.

SUBSTR('ABCDEFGHIJK', -5, 2) returns GH

INSTR-locates one string pattern inside another string. INSTR (strExp1, strExp2, pos_start, NumOccurrences) returns the position in strExp1 where strExp2 starts. The third argument allows the search to start at position pos_start. The fourth argument searches for the NumOccurrences occurrence. Strings are numbered from position 1

Demo 13:

INSTR('ABCDEABCDE', 'CD') returns 3

The first match of CD is in position 3

INSTR('ABCDEABCDE', 'cd') returns 0

This is a case specific test and there is no match

INSTR('ABCDEABCDE', 'CD', 5) returns 8

The search starts at position 5; the return value is the count from the start of the string.

INSTR('ABCDEABCDE', 'CD', 25) returns 0

It is not an error to start the search after the end of the string.

INSTR('ABCDEABCDE', 'CD', 1, 2) returns 8

This is looking for the second occurrence of CD

INSTR('ABCDEABCDE', 'CD', 1, 3) returns 0

This is looking for the third occurrence of CD

Demo 14: Returns all products that have the string "mixer" in their description.

```
select prod_id
, prod_name
, prod_desc
from prd_products
where INSTR( UPPER(prod_desc ), 'MIXER') > 0;
```

Demo 15: Returns all products that do not have the string "mixer" in their description.

```
select prod_id
, prod_name
, prod_desc
from prd_products
where INSTR( UPPER(prod_desc) , 'MIXER') =0;
```

4. Changing the string contents

REPLACE (strExp1, StrExp2, strExp3) replaces, or deletes, occurrences of one string within another.

Demo 16:

```
select REPLACE( 'ABCDABCDABCD',      'B',      'cat')
,       REPLACE( 'ABCDABCDABCD',      'BCD',     '-')
from dual;
REPLACE('ABCDABCDA      REPLAC
-----      -----
AcatCDACatCDAcatCD      A-A-A-
```

```
select REPLACE( 'ABCDABCDABCD',      'DCB',    'xy')
,       REPLACE( 'ABCDABCDABCD',      'C' )
from dual;
REPLACE('ABC      REPLACE(
-----      -----
ABCDABCDABCD      ABDABDABD
```

TRANSLATE replaces one set of characters with another set of characters; if any of the arguments are null, the return value is null. We have three arguments, the expression to be translated, the from_string and the to_string. This is done on a character basis. (Replace works on a string basis)

Demo 17: In this function call 1 is translated to A, 2 to B and 3 to C. 4 and 5 have no translation characters in the from_string so they are not translated.

```
select TRANSLATE( '123452313',      '123',      'ABC') from dual;
TRANSLATE('123452313','123','ABC')
-----
ABC45BCAC
```

Now 3 is translated to C; 4 and 5 have no characters in the to_string argument and they are removed from the final result.

```
select TRANSLATE( '4152393768',      '345',      'C') from dual;
TRANSLATE('4152393768','345','C')
-----
12C9C768
```

Demo 18: Removing characters with translate

Suppose I have phone numbers that may have been entered with punctuation marks and I want to strip out the punctuation- specifically any space, parentheses, hyphens, dots, underscores. The result in column 1 returns a null because the ZLS in Oracle is considered a null. The solution is to tack on a character and translate it to itself (col2- all of the W were translated to W and the other characters in the from_string had no match in the to_string and were removed.).

```
variable ph varchar2(20);
```

```
exec :ph :='(415) 239-3768';

select
translate(:ph, ' ()-.#_', '') as col1,
translate(:ph, 'W ()-.#_', 'W') as col2
from dual;

```

COL1	COL2
-----	-----
	4152393768

Demo 19: Switching delimiters in a string- assumes that all of the commas are delimiters

```
select translate ('cat,dog,bird,horse,house fly', ',', '|') as col1
from dual;

```

COL1

cat dog bird hors house fly

Switching delimiters and removing all blanks ; the blanks within a value in the list are also removed.

```
select translate ('cat ,dog , bird , horse, house fly ', ', ', '|') as col1
from dual;

```

COL1

cat dog bird hors housefly

5. Misc string functions

Length: Length(strExp) returns the number of characters in the expression. If the argument is a varchar2 column, then Length returns the actual number of characters in the data value, not the defined maximum length of the column. If the argument is a null string, Length returns null (not 0).

Demo 20: **ASCII** returns the ASCII number corresponding to the first character in the argument string. The last column aliases defaulted to uppercase but the function ran with the lower case argument.

```
select ASCII('BUSH'), ASCII('B'), ASCII('baby') from dual;
```

ASCII('BUSH')	ASCII('B')	ASCII('BABY')
-----	-----	-----
66	66	98

Demo 21: **CHR** returns the character associated with an ASCII number

```
select CHR( 73), CHR( 74) ,CHR( 888) from dual;
```

C	C	C
-	-	-
I	J	X

Table of Contents

1. Regular Expressions.....	1
2. Metacharacters	2
3. Demos	2
3.1. Regexp_Like	2
3.2. Regexp_Instr.....	6
3.3. RegExp_Replace	8
3.4. RegExp_Count	8

1. Regular Expressions

Regular expressions allow you to do more complex pattern matching than wildcards. Wildcards are limited to matching any single character and any series of characters. With regular expressions we can build a pattern that specifies specific characters to search for, ranges of characters and repeating patterns of characters.

A regular expression is a string that expresses a pattern for text matching. The regular expression string can contain literal characters and metacharacters. For example the regular expression 'cat' matches the string 'cat' and the regular expression 'p..t' matches a string that contains a 'p' followed by any two characters followed by a 't'.

One of the most important things to understand about regular expressions is that they can work very well for matching strings that have some sort of consistent pattern, but they are not good for matching unstructured text. You may have to decide if you want to use a pattern that produces a lot of matches including undesired matches (false positives), or a pattern that produces fewer matches but misses some of the matches you want to find. You can find examples of regular expression patterns on various web sites but you need to test these- for example does the pattern for a phone number assume that you will use parentheses and a dash (415) 239-3768 or does it allow the format 415.239.3768; does it handle extensions?, country codes? Does the url pattern you found assume that the top level domain is two or three characters long? Does it assume English characters only?

If you have worked with regular expressions, you know that they can get quite complex- we will stick with some simple patterns. If you are planning to use complex regular expression, consult the Oracle manuals for more details and test carefully.

In order to use the regular expressions, you need functions to interpret the regular expression. Oracle includes the following functions for working with regular expressions.

- **REGEXP_LIKE** Returns Boolean
- **REGEXP_INSTR** Returns number
- **REGEXP_SUBSTR** Returns part of the string
- **REGEXP_REPLACE** Returns string with replacements

Each of these functions is built similar to the regular string functions/operators (Like, Instr, Substr, Replace).

They each take a source string, a regular expression pattern to match, and optional match options. The match options are

- i case insensitive
- c case sensitive
- n period matches new line
- m more than one line in source is ok

Regexp_Instr, Regexp_Substr, and Regexp_Replace take an optional start position for the matching and an optional occurrences parameter to indicate which occurrence should be returned.

Regexp_Replace has a parameter for the replacement string

Regexp_Instr has a parameter for a return option

- `rtrn_option = 0` → return first character of the occurrence
- `rtrn_option = 1` → return first character following the occurrence

That was probably rather confusing and intimidating. If this is your first experience with regular expressions just work through the section on Metacharacters and the demos for RegExp_Like.

2. Metacharacters

The Like operator recognized the metacharacters % and _. Regular expressions have a much richer set of metacharacters. You can find a list of them in the Price book

What the regex expressions provide is the ability to search for ranges/lists of characters. Instead of looking for any single character as with a wildcard, you could look for any of the listed characters [aeiou]. You could also specify that you want to match from 3 to 6 of those characters in a row [aeiou]{3-6}

You could match a pattern of as many 0s and 1s as occurs by using [01]* and if you really want at least one of these, use [01]+

There are also predefined patterns such as [:lower:] which stands for any lower case letter (bracket expressions, character classes)

In this first set of demos we use the metacharacters

<code>^</code>	start of the string
<code>\$</code>	end of the string
<code>.</code>	any single character
<code>{n}</code>	repetition; want exactly n of the previous character <code>g{3}</code> matches ggg
<code>{n, m}</code>	repetition; want between n and m of the previous character <code>g{3,5}</code> matches ggg, gggg, ggggg
<code>*</code>	repetition; any number, including 0 of the preceding character
<code>+</code>	repetition; any number, but at least one, of the preceding character
<code>?</code>	repetition; zero or one of the preceding character
<code>[abc...]</code>	list - matches any one of the included characters
<code>[a-p]</code>	range - matches any one of the characters in the indicated range

3. Demos

3.1. RegExp_Like

RegExp_Like is used in a Where clause and returns True if that pattern occurs within the search string.

For these demos we will use a table set up for this purpose. The table has an ID column and a name column; we will be testing against the name column. The SQL to create the table is in the demo.

These are the rows in the table.

ID	NAME
1	Fluffy
2	goofy
3	ursula
4	greg
5	pout
6	Sam 415
7	pretty bird
8	pat
9	peat
10	Patricia

```
11 Impromptu  
12 Pete  
13 pat the cat  
14 C3PO  
15 Mary Proud  
16 ptt  
17 pita
```

As you work with these, try to do the same task using string functions and wildcards,

Demo 01: The **^ metacharacter** matches the start of the string.

```
select * from z_tst_reg  
where regexp_Like (name, '^g')  
;  


| ID | NAME  |
|----|-------|
| 2  | goofy |
| 4  | greg  |


```

Note that if we do not include the character for matching the start of the string, we can match anywhere inside the string. For example, `Regexp_Like (name, 'p')` matches Impromptu as well as matching pita

Demo 02: The **\$ metacharacter** matches the end of the string.

```
select * from z_tst_reg  
where regexp_Like (name, 'g$')  
;  


| ID | NAME |
|----|------|
| 4  | greg |


```

Demo 03: The **. (dot) metacharacter** matches any single character. This pattern matches any string that starts with a p and ends with a t and has exactly one character between.

```
select * from z_tst_reg  
where regexp_Like (name, '^p.t$')  
;  


| ID | NAME |
|----|------|
| 8  | pat  |
| 16 | ptt  |


```

Demo 04: This pattern matches any string that starts with a p and ends with a t and has exactly two characters between.

```
select * from z_tst_reg  
where regexp_Like (name, '^p..t$')  
;  


| ID | NAME |
|----|------|
| 5  | pout |
| 9  | peat |


```

Demo 05: It can help to also display rows that do not match the pattern

```
select * from z_tst_reg
where Not regexp_Like(name, '^p..t$');
```

ID	NAME
1	Fluffy
2	goofy
3	ursula
4	greg
6	Sam 415
7	pretty bird
8	pat
10	Patricia
11	Impromptu
12	Pete
13	pat the cat
14	C3PO
15	Mary Proud
16	ptt
17	pita

Demo 06: We can use {n} to indicate that we want exactly n of the preceding character. Note that I do not have the \$ to tie this to the end of the string.

```
select * from z_tst_reg
where regexp_Like (name, '^p.{3}t')
;
```

ID	NAME
7	pretty bird
13	pat the cat

Demo 07: We can use {n, m} to indicate that we want between n and m of the preceding character.

```
select * from z_tst_reg
where regexp_Like (name, '^p.{4,7}a', 'i')
;
```

ID	NAME
10	Patricia

Demo 08: We can use * to indicate that we will match any number, including 0, of the preceding character.

```
select * from z_tst_reg
where regexp_Like (name, '^p.*t')
;
```

ID	NAME
5	pout
7	pretty bird
8	pat
9	peat
13	pat the cat
16	ptt
17	pita

Demo 09: Use + to indicate that you must match at least one. Use ? for matching either 0 or 1 of the preceding.

```
select * from z_tst_reg
where regexp_Like (name, 'pr?o');
```

ID	NAME
5	pout
11	Impromptu

Demo 10: Include the '**i**' option for case insensitivity.

```
select * from z_tst_reg
where regexp_Like (name, 'pr?o', 'i')
; 
```

ID	NAME
5	pout
11	Impromptu
14	C3PO
15	Mary Proud

Demo 11: The use of [] allows for a list or a range of characters to match.

```
select * from z_tst_reg
where regexp_Like (name, '[aeiouy]$')
; 
```

ID	NAME
1	Fluffy
2	goofy
3	ursula
10	Patricia
11	Impromptu
12	Pete
17	pita

Demo 12: The [] allows for a list or a range of characters to match. This matches an r followed by a single vowel followed by a character in the range a-m.

```
select * from z_tst_reg
where regexp_Like (name, 'r[aeiouy][a-m]')
; 
```

ID	NAME
4	greg
10	Patricia
11	Impromptu

Demo 13: Regular expression also includes character classes. This matches strings that include any whitespace character

```
select * from z_tst_reg
where regexp_Like (name, '[[:blank:]]')
; 
```

ID	NAME
6	Sam 415
7	pretty bird
13	pat the cat
15	Mary Proud

Demo 14: This matches strings that include an upper case letter followed by a lower case letter.

```
select * from z_tst_reg
where regexp_Like (name, '[[:upper:]][[:lower:]]')
;

```

ID	NAME
1	Fluffy
6	Sam 415
10	Patricia
11	Impromptu
12	Pete
15	Mary Proud

Demo 15: This matches any digit

```
select * from z_tst_reg
where regexp_Like (name, '[[:digit:]]')
;

```

ID	NAME
6	Sam 415
14	C3PO

3.2. **Regexp_Instr**

This is a model to use to experiment with regular expressions. The first two statements define a bind variable that will hold a string. The next statements set the values of those variables. You can change the literals to do other tests. Finally there is a select statement that uses the regexp_Instr function to tell you where the pattern occurs in the string.

The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the beginning or ending position of the matched substring, depending on the value of the return_option argument. If no match is found, the function returns 0.

Demo 16: Regexp_Instr

```
variable s varchar2(250)
variable p varchar2(25)

exec :s := 'This is my test string';
exec :p := 'is';

select regexp_Instr(:s, :p) as Location
from dual;

LOCATION
-----
3
```

REGEXP_INSTR and REGEXP_SUBSTR now (Oracle 11g) have an optional subexpr parameter that lets you target a particular substring of the regular expression being evaluated. You can say where to start the search and which occurrence to match.

Demo 17: Regexp_Instr with start and count parameters

```
select
  regexp_Instr(:s, :p)      as Loc_1
, regexp_Instr(:s, :p, 5, 1) as Loc_2
```

```
, regexp_Instr(:s, :p, 10, 1) as Loc_3
from dual;
```

LOC_1	LOC_2	LOC_3
3	6	0

(This is easier to use in SQL Developer where you can paste all of these lines into the code window and then edit the literals and rerun all of the code.)

You can also do simple demos with literals

Demo 18: Exact match with anchors; only '^CAT\$' is a match

```
select
  regexp_Instr ('CAT', '^CAT$')   as test_01,
  regexp_Instr ('CAT', '^CAAT$')  as test_02,
  regexp_Instr ('CAT', '^CAAAT$') as test_03
from dual;
```

TEST_01	TEST_02	TEST_03
1	0	0

Demo 19: The only one that matches is Sno- with 4 o's followed by a w

```
select
  regexp_Instr ('Snoooow', '^Sno{3}w$') as test_04,
  regexp_Instr ('Snoooow', '^Sno{4}w$') as test_05,
  regexp_Instr ('Snoooow', '^Sno{5}w$') as test_06
from dual;
```

TEST_04	TEST_05	TEST_06
0	1	0

Demo 20: The first argument contains 7 o's so this matches both {7,7} and {5,9}

```
select
  regexp_Instr ('Shoooooooo', '^Shmo{3,4}$') as test_7,
  regexp_Instr ('Shoooooooo', '^Shmo{7,7}$') as test_8,
  regexp_Instr ('Shoooooooo', '^Shmo{5,9}$') as test_9,
  regexp_Instr ('Shoooooooo', '^Shmo{9,12}$') as test_10
from dual;
```

TEST_7	TEST_8	TEST_9	TEST_10
0	1	1	0

Demo 21: This uses different patterns for the preceding character. The preceding character can be a single literal character or a [] list of possible characters to match

```
select
  regexp_Instr ('XabcbcacW', '^Xa{4,8}W$')   as test_11,
  regexp_Instr ('XabcbcacW', '^X[ab]{4,8}W$')  as test_12,
  regexp_Instr ('XabcbcacW', '^X[abc]{4,8}W$') as test_13,
  regexp_Instr ('XabcbcacW', '^X.{4,8}W$')    as test_14
from dual;
```

TEST_11	TEST_12	TEST_13	TEST_14
0	0	1	1

Demo 22: Using variable for the limits take a bit more work. You do not want the variable identifier :low inside the delimiters. The first expression does not handle the variables properly. You use || to build up the expression.

```
variable b_low number;
exec :b_low := 5;
variable b_high number;
exec :b_high := 9;

select
  :'Shoooooooo' , '^Shmo{:b_low,:b_high}$' ) as test_15,
  regexp_instr ('Shoooooooo' , '^Shmo'|| :b_low || ','|| :b_high || '$') as test_16
from dual;
```

TEST_15	TEST_16
0	1

3.3. RegExp_Replace

With the following literal, there are three blanks between San and Francisco. The first version of RegExp_Replace says to replace two blanks with one- that leaves us with two blanks. The second version says to replace any 2 or more blanks with a single blank.

```
select regexp_replace('San      Francisco', '  ', ' ')
from Dual;

select regexp_replace('San      Francisco', '[ ]{2,}', ' ')
from Dual;
```

3.4. RegExp_Count

RegExp_Count is a new (Oracle 11g) function that counts the number of occurrences of a specified regular expression pattern in a source string.

Demo 23: RegExp_Count

```
select name, regexp_count(name, '[aeiou]') as Vowel_count
, regexp_count(name, 'at') as at_count
from z_tst reg;
```

NAME	VOWEL_COUNT	AT_COUNT
Fluffy	1	0
goofy	2	0
ursula	3	0
greg	1	0
pout	2	0
Sam 415	1	0
pretty bird	2	0
pat	1	1
peat	2	1
Patricia	4	1
Impromptu	2	0
Pete	2	0
pat the cat	3	2
C3PO	0	0
Mary Proud	3	0
ptt	0	0
pita	2	0

Table of Contents

1. Cast.....	1
2. Formatting numbers with To_Char	1
3. Date formats	2
4. Formatting date values with To_Char.....	3

Oracle will do a lot of implicit data conversions for you, but there may be times when you need to do an explicit data type conversion. Some of the intrinsic functions are very "fussy" about the data type of their arguments; sorting is different if it is done alphabetically or numerically. Union queries also may need conversion functions to produce union compatible columns. The conversion functions are: CAST, TO_CHAR, and TO_DATE (To_Date is discussed with the temporal functions)

1. Cast

CAST is a general purpose conversion function. It can be used to convert between the built-in data types.

These cast functions work.

```
Cast ('345' as Number)      → 345
Cast ('345.678' as Number (8,2)) → 345.68
Cast ('345678' as Number (8,2)) → 345678
```

The following will fail. The first specifies too narrow a target; the second has a string argument that Oracle cannot convert.

```
Cast ('345678' as Number (5,2)) →
ORA-01438: value larger than specified precision allowed for this column

cast('VII' as number)
ORA-01722: invalid number
```

2. Formatting numbers with To_Char to get a string

To_Char is used to convert a numeric or date expression into a string; you can supply a format to be used for the conversion. This can be used if you want to have consistently formatted output. The result is enclosed within two uprights so that you can see where the spaces are in the result.

Examples of Using ' TO_CHAR (NumExp, 'FormatString') '''	Result
TO_CHAR (3.5, '9999')	4
TO_CHAR (3.5, '0999')	0004
TO_CHAR (3.5, '999.99')	3.50
TO_CHAR (3.5, '\$999.99')	\$3.50
TO CHAR (3.5, '\$099.99')	\$003.50
TO_CHAR (34567.5, '099.99')	#####
TO CHAR (34567.5, '999,999.99')	34,567.50
TO_CHAR (345, '999')	345

TO_CHAR (345, 'FM999')	345
TO CHAR (345, 'S999')	+345
TO_CHAR (345, '999S')	345+
TO CHAR (345, '9999V99')	34500
TO_CHAR (-345, '9999V99')	-34500
TO CHAR (345, 'RN')	CCCXLVI
TO CHAR (345, 'rn')	cccxlvi

Numeric Formats to use with TO_CHAR		Used to
Element	Example	
9	999.99	Display the specified number of digits; does not display a leading zero; there will be a leading space or minus sign
0	0999	Displays a leading zero if applicable
\$	\$999,999.00	Floating dollar symbol; moves to the left
L	L9999.00	The Local currency symbol
.	9.999	Decimal point; only one decimal point allowed
,	99,999,999	Comma ; you need to position this carefully; you can have multiple commas but they must appear to the left of any decimal point
PR at the end of the format	999.00PR	To display negative numbers inside angle brackets <99.00> is the negative number -99.00
EEEE	9.99EEEE	For scientific notation
FM	FM99.9	Suppresses the leading space
S	S999 999S	Displays a leading (or trailing) plus or minus sign.
MI	999MI	Display a minus sign or trailing blank in the last position.
V	999V99	Returns the number multiplied by 10 ⁿ , where n is the number of digits after the V symbol.

3. Date formats

You need to keep in mind that the format used to display a date is a string; it is a representation of a stored date value. Suppose we have a datetime value of July 4, 2012 midnight. We could display this value as "July 4, 2012", "2012-07-04", or , "2012/7/4" and these would each be a different string representation of the same value .

There is a default format for displaying and entering date values. The default format is commonly of the form '19-AUG-02' represented as a DD-MON-RR format. This means that dates entered with a two digit year have the first two digits of the year calculated based on the last two digits of the current year and the last two digits of

the specified year. It is safer to specify a 4-digit year format and enter dates with a 4-digit year. Since several tools can change default settings, you should not rely on a default date format.

Since we enter and display dates in a variety of formats, there are conversion functions to handle the differences.

Oracle DATE values always include a date and a time component. Suppose you create a table d_sysdate with a single date attribute and insert the value of sysdate.

```
create table z_sysdate (col1 date);
insert into z_sysdate values (sysdate);
```

Wait a minute and then run the following query.

```
select * from z_sysdate
where col1 = sysdate;
```

No rows will be returned since the time portions will not match.

If your date values are not time sensitive, then you could use date functions (such as TRUNC) to handle this issue when testing a date value against another date value.

4. Formatting date values with To_Char to get a string

The TO_CHAR function is used to specify a display format for a date value. This is a partial list of formats; consult your text or the manuals for a more complete list.

Examples run on Thursday June 03, 2004.

Examples	Result
TO_CHAR (SYSDATE, 'YYYY-MM-DD')	2004-06-03
TO_CHAR (SYSDATE, 'MON DD, YYYY')	JUN 03, 2004
TO_CHAR (SYSDATE, 'MM')	06
TO_CHAR (SYSDATE, 'YYYY')	2004
TO_CHAR (SYSDATE, 'D')	5 <i>{Thurs is the 5th day of the week}</i>
TO_CHAR (SYSDATE, 'Month DD, yyyy') The month name takes 9 spaces	June 03, 2004

"fm" is used to turn on and off leading zeros and blank padding

TO_CHAR (SYSDATE, 'fmMonth DD, yyyy') Now the month name gets only the space it needs	June 3, 2004
TO_CHAR (SYSDATE, 'fmMonth fmDD, yyyy') The month name is not padded, but the day number is.	June 03, 2004
TO_CHAR (SYSDATE, 'MON')	JUN
TO_CHAR (SYSDATE, 'DDD')	155
TO_CHAR (SYSDATE, 'DY')	THU
TO_CHAR (SYSDATE, 'day')	thursday
TO_CHAR (SYSDATE, ' DAY ')	THURSDAY
TO_CHAR (SYSDATE, 'Mon Ddth')	Jun 03rd
TO_CHAR (SYSDATE, 'Mon Ddsp')	Jun Three

TO_CHAR (SYSDATE, 'Mon DDspth')	Jun THIRD
TO_CHAR (SYSDATE, 'Mon Ddth Yyyysp')	Jun 03rd Two Thousand Four
TO_CHAR (SYSDATE, 'J')	2453160 <i>{Julian day- start Dec 31 4712BC}</i>

Time formats

If we display the value of sysdate, the default display format show only the day month and year. But sysdate does have a time component.

```
select sysdate from dual;
-----
```

SYSDATE

19-FEB-14

If we want to see the time, we need to include time formats.

```
select to_char(sysdate, 'HH24 MI SS') as "Time" from dual;
-----
```

Time

19 40 49

Wait a few moments and run the query again.

Time

19 41 28

You can combine the date and time formats.

```
select to_char(sysdate, 'YYYY-Mon-DD HH24:MI') as "Date and Time" from dual;
-----
```

Date and Time

2014-Feb-19 19:43

```
select to_char(sysdate, 'YYYY-Mon-DD HH24 "Hours and" MI "Minutes"')
as "Date and Time"
from dual;
```

Date and Time

2014-Feb-19 19 Hours and 45 Minutes

Table of Contents

1.	Testing Nulls	1
1.1.	Coalesce	2
1.2.	NVL	3
1.3.	NULLIF	3
1.4.	NVL2	7

1. Testing Nulls

We have nulls in our data tables. A null represents a situation where a value is not known or is not applicable. We have seen that nulls propagate in arithmetic, but not in string concatenation with Oracle. Some clients display nulls as blank space; others as the literal “(null)”. We may want to have more control over how nulls are handled and displayed in our SQL code. The function considered here are:

- COALESCE()
- NVL()
- NULLIF()
- NVL2()

A common situation is that we have nulls in a table and want to substitute a different value for the null in the result. We have a function Coalesce which lets us do that. One major concern is what is the appropriate value to use for a data value if the actual data value is not known. This is a **business rule decision** that is not the choice of the SQL coder. It is *not* always, or even often, appropriate to substitute a value of 0 if a numeric value is missing.

Suppose that we have an order for high def wide screen TVs and for some reason there is no price in the database for this item. If you substitute a value of 0- you have just given away a rather expensive product for free. It is more helpful to reject that data as being invalid and it might be even a better idea to not let an item be put into the products table for sale until we have a price.

On the other hand, if we are adding up test scores for student and a student did not take a test, then substituting a zero may make sense rather than returning a null for the test total score.

The point is that the person writing the code does not make the decision as to the proper way to handle a null. If the business knowledge expert does not specify how to handle this situation, ask!

Some of these demos use a small table z_tst_numbers, which has integer columns and which has a lot of nulls in it: I am inserting a <n> for the nulls in **this** display. Code for this table is in the demo.

A	B	C	D	E	F
1	10	10	50	90	45
2	15	5	<n>	10	0
3	<n>	<n>	50	50	-1
4	<n>	<n>	<n>	<n>	<n>
5	0	0	0	0	0
6	10	10	10	10	10
7	-10	10	0	-210	85
8	-10	-1	0	-210	85
9	200	-1	0	-1	85
10	200	200	0	-1	46

1.1. Coalesce

COALESCE accepts a series of values and returns the first non-null value. If all values are null, then the function returns null. Coalesce can cause an error if you use arguments of different data types and they cannot be cast to the same type. The coalesce function is an ansi standard function available in Oracle, T-SQL and MySQL. One thing you can do to improve the portability of your code is to use techniques and functions that work on multiple dbms. You will see the NVL function, which is an Oracle specific function, used in a lot of older code.

Demo 01: coalesce. The second expression looks at column B and if it is null, is evaluated as 9999. The third and fourth expression look at multiple columns; the arguments are in different order- the first non-null argument is returned.

```
select
  A
 , COALESCE(B, 9999) as B_Col
 , COALESCE(B,C,D,E) as Coalesce_BCDE
 , COALESCE(E,D,C,B) as Coalesce_EDCB
 from z_tst_numbers
;
```

A	B_COL	COALESCE_BCDE	COALESCE_EDCB
1	10	10	90
2	15	15	10
3	9999	50	50
4	9999		
5	0	0	0
6	10	10	10
7	-10	-10	-210
8	-10	-10	-210
9	200	200	-1
10	200	200	-1

Do not simply coalesce numeric columns to 0 or string columns to ZLS; this is generally incorrect. In the following query, you cannot tell the difference in the result between a value of 0 that was inserted into the column directly in the Insert statement, and a 0 that comes from coalesce. Just because you do not know what a value is does not mean it is zero. (I do not yet know your midterm exam score, but I do not think you want to me to assume it is 0.)

```
select
  A
 , COALESCE(B,0) as BCol
 , COALESCE(D,0) as DCol
 from z_tst_numbers
;
```

A	BCOL	DCOL
1	10	50
2	15	0
3	0	50
4	0	0
5	0	0
6	10	10
7	-10	0
8	-10	0
9	200	0
10	200	0

1.2. NVL

NVL(exp, sub_value) If the expression is null, then NVL returns sub_value; otherwise it returns the value of the expression.

The difference between Coalesce and NVL is that Coalesce accepts more than two parameters; Nvl accepts only two parameters; NVL is an Oracle proprietary function.

NVL (p_1, p_Null) is executed as

```
if p_1 is null then
    return p_null
else
    return p_1
end if
```

Demo 02: NVL. The expression for the column B_Null says that if column B is null use 87 instead. The expression for the column D_Null says that if column D is null use 0 instead. Notice that in this column you cannot tell the difference between the values of 0 that come from the original data in column D and the values of 0 that come from the NVL expression

```
select A
, B, NVL(B,87) as B_Null
, D, NVL(D,0) as D_Null
, Coalesce (D,0) as D_null_Coalesce
from z_tst_numbers;
```

A	B	B_NULL	D	D_NULL	D_NULL_COALESCE
1	10	10	50	50	50
2	15	15		0	0
3		87	50	50	50
4		87		0	0
5	0	0	0	0	0
6	10	10	10	10	10
7	-10	-10	0	0	0
8	-10	-10	0	0	0
9	200	200	0	0	0
10	200	200	0	0	0

Demo 03: This is how you write the query demo 01 using nvl and nested functions

```
select
  A
, NVL(B,9999) as B_Col
, NVL(B, NVL(C, NVL(D,E))) as NVL_BCDE
, NVL(E, NVL(D, NVL(C,B))) as NVL_EDCB
from z_tst_numbers
;
```

1.3. NULLIF

NULLIF (value1, value2) If value1 = value2, then NULLIF returns null, otherwise it returns value1.

NULLIF (p_1, p_2) is executed as

```
if p_1 = p_2 then
    return null
else
    return p_1
end if
```

Demo 04: NULLIF nullif(B,C) returns a null if B and C have the same value

```
select A, B, C, nullif(B,C)
from z_tst_numbers;
```

A	B	C	NULLIF(B,C)
1	10	10	
2	15	5	15
3			
4			
5	0	0	
6	10	10	
7	-10	10	-10
8	-10	-1	-10
9	200	-1	200
10	200	200	

This can be used as a cleanup function. Suppose you inherit a table from someone who does not believe in allowing nulls and they used the flag value -1 to indicate that this value should have been a null. Since you want to use code that works with nulls, you can use the nullif function.

Demo 05: NULLIF to clean up flags used instead of nulls

```
select A, C, NULLIF(C, -1)
from z_tst_numbers;
```

A	C	NULLIF(C, -1)
1	10	10
2	5	5
3		
4		
5	0	0
6	10	10
7	10	10
8	-1	
9	-1	
10	200	200

In the following query, the third column, NULLIF(B, 200), returns all values of column B except for 200: The value 200 gets nulled out.

The fifth column, NULLIF(85, F), nulls out any original value of 85 in column F and returns the value 85 for the rest of the rows.

Demo 06: NULLIF to null out values

```
select A
, B, NULLIF(B, 200)
, F, NULLIF(85, F)
from z_tst_numbers;
```

A	B	NULLIF(B, 200)	F	NULLIF(85, F)
1	10	10	45	85
2	15	15	0	85
3			-1	85
4				85
5	0	0	0	85
6	10	10	10	85
7	-10	-10	85	
8	-10	-10	85	
9	200		85	
10	200		46	85

`NULLIF(0, X-X)` returns a null if X has a value. If X has a value, then $X-X$ is 0 and the two arguments have the same value.

If X is null, then the function returns a 0. If X is null, the $X - X$ is null and the two arguments do not have the same value.

Demo 07: NULLIF to flip nulls and non-nulls

```
select A, B
, NULLIF(0, B - B)
from z_tst_numbers
;
-----
```

A	B	NULLIF(0, B-B)
1	10	
2	15	
3		0
4		0
5	0	
6	10	
7	-10	
8	-10	
9	200	
10	200	

Demo 08: Using Null if to avoid divide by zero error

```
-- this is a regular division that has no surprises
variable num number;
exec :num := 5;
select 20/:num from dual
;
-----
```

20/:NUM	4
---------	---

```
-- if we try to divide by zero we get an error message
exec :num := 0;
select 20/:num from dual;
```

ERROR at line 1:
ORA-01476: divisor is equal to zero

We can use `nullif` to return anull divisor instead of dividing by zero. Of course you need to decide if a null makes sense in terms of what you are trying to do. We now have one expression that executes with a zero or a non-zero value.

```
exec :num := 5;
select 20 / (nullif(:num,0)) as Quotient
from dual;
```

QUOTIENT	

4	

```
exec :num := 0;
/
-----
```

QUOTIENT	

-- this is the null return	

Demo 09: Using nullif in an aggregate. This uses the average function - which we get to in another unit.

Column D has some nulls and some zero values. Suppose that people who were entering data used a value of zero when they should have used a null.

```
select D from z_tst_numbers;
```

D

50
50
0
10
0
0
0
0
0

10 rows selected.

```
select D from z_tst_numbers  
where D is not null;
```

D

50
50
0
10
0
0
0
0

8 rows selected.

The average function skips any null values; but we want it to also skip any zero values. We can use nullif to say if the value is zero use a null instead.

The first column calculates the average including the zero values. The second column effectively says to ignore the zero values in the average.

```
select avg(D), avg( nullif(D,0))  
from z_tst_numbers;
```

AVG(D)	AVG(NULLIF(D,0))
-----	-----
13.75	36.666667

NULLIF is one of those functions that you think you will never need-until some day when you find yourself writing code to handle these situation.

1.4. NVL2

The NVL2 function returns an If-then test. It looks at the first value and then returns one of two values depending on whether or not the first parameter is null. If you have done programming, this is the logic for nvl2.

NVL2(exp, sub1_value, sub2_value) If the expression is not null, then NVL2 returns sub1_value, otherwise it returns sub2_value.

NVL2 (p_1, p_NotNull, p_Null) is executed as

```
if p_1 is null then
    return p_null
else
    return p_NotNull
end if
```

Demo 10: NVL2

```
select A
, B, NVL2(B, 300, 600)
from z_tst_numbers;
```

A	B	NVL2 (B, 300, 600)
1	10	300
2	15	300
3		600
4		600
5	0	300
6	10	300
7	-10	300
8	-10	300
9	200	300
10	200	300

Table of Contents

1. Searched Case Expression	1
1.1. Return type consistency.....	3
1.2. Including other functions	3
2. Simple Case Expression	6

The Case expressions are used to perform selection logic. The case expression is part of standard SQL and corresponds closely to selection logic found in most programming languages. The case expression is not a function but it is a bit more complex than the simpler expressions we used in earlier units.

1. Searched Case Expression

The searched Case expression requires a logical expression to be evaluated at each WHEN clause. It is faster than evaluating the DECODE function and is closer to the logic found in many programming languages. (Decode is discussed in another document in this unit.)

The data type of the return expressions must all be of the same type family or all numeric; that type becomes the return type of the case expression; if return expressions are all numeric then the return type of the case expression is the highest precedence of the various return expression numeric types.

You can use a variety of tests- In lists, Between, wildcard tests and you can mix the tests in a single case expression. You can nest case expressions.

Demo 01: We want to give customers a 5% savings for each pet supply item, 5% for each sporting goods item and 10% for each appliance. As a first step we will determine the saving percent.

```
select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id = 'PET'      THEN 0.95
    WHEN catg_id = 'SPG'      THEN 0.95
    WHEN catg_id = 'APL'      THEN 0.90
    ELSE 1
END "Savings %"
from prd_products
order by catg_id;
```

--- selected rows shown

CATG_I	PROD_ID	PROD_LIST_PRICE	Savings %
APL	1126	850	.9
APL	1125	500	.9
GFD	5001	5	1
GFD	5000	12.5	1
HD	5002	23	1
HD	5005	45	1
HW	4575	49.95	1
HW	1090	149.99	1
HW	1000	125	1
HW	1070	25.5	1
PET	1140	14.99	.95
PET	4577	29.95	.95
PET	4576	29.95	.95
PET	4568	549.99	.95
SPG	1060	255.95	.95
SPG	1050	269.95	.95

Demo 02: We can use the calculated percent to determine the sales price

```
select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id ='PET'      THEN 0.95
    WHEN catg_id ='SPG'      THEN 0.95
    WHEN catg_id ='APL'      THEN 0.90
    ELSE 1
END * prod_list_price AS "Today's Price"
from prd_products
order by catg_id;
--- selected rows shown
```

CATG_I	PROD_ID	PROD_LIST_PRICE	Today's Price
APL	4569	349.95	314.955
APL	1120	549.99	494.991
APL	1130	149.99	134.991
GFD	5001	5	5
GFD	5000	12.5	12.5
HD	5002	23	23
HD	5005	45	45
HW	1070	25.5	25.5
PET	1140	14.99	14.2405
PET	4576	29.95	28.4525
PET	1141	99.99	94.9905
PET	1142	2.5	2.375
SPG	1060	255.95	243.1525

Demo 03: You should include an Else clause unless you are certain that all possible values are handled. Here I have removed the else clause and products which do not fall into one of the three categories tested, get a value of null from the case expression and therefore have a null value for the last column. This does not follow the business rule of demo 01.

```
select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id ='PET'      THEN 0.95
    WHEN catg_id ='SPG'      THEN 0.95
    WHEN catg_id ='APL'      THEN 0.90
    END * prod_list_price AS "Today's Price"
from prd_products
order by catg_id;
--- selected rows shown
```

CATG_I	PROD_ID	PROD_LIST_PRICE	Today's Price
APL	4569	349.95	314.955
APL	1120	549.99	494.991
APL	1130	149.99	134.991
GFD	5001	5	
GFD	5000	12.5	
HD	5002	23	
HW	1000	125	
HW	1070	25.5	
PET	1140	14.99	14.2405
PET	4576	29.95	28.4525
PET	1142	2.5	2.375
SPG	1060	255.95	243.1525

For more maintainable code you might wish to have a table of categories and current discounts and do a join. That way if the discount rates changed or if new categories were added, you would not have to rewrite the query.

1.1. Return type consistency

We cannot write a case expression such as the following queries.

Demo 04: This fails since the return type was expected to be a number and one of the return expression is a character 'no discount' as the return value.

```
select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id ='PET'      THEN 0.95
    WHEN catg_id ='SPG'      THEN 0.95
    WHEN catg_id ='APL'      THEN 0.90
    ELSE  'no discount'
END "Savings %"
from prd_products
order by catg_id
;
SQL Error: ORA-00932: inconsistent datatypes: expected NUMBER got CHAR
```

Demo 05: This fails since the return type was expected to be a char and other the return expressions are numbers

```
select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id ='PET'      THEN 'no discount'
    WHEN catg_id ='SPG'      THEN 0.95
    WHEN catg_id ='APL'      THEN 0.90
    ELSE  1
END "Savings %"
from prd_products
order by catg_id
;
SQL Error: ORA-00932: inconsistent datatypes: expected CHAR got NUMBER
```

Demo 06: This fails since we have mixed return expression types even though we do not have any catg_id value of 'XXX' and that expression would never be evaluated.

```
select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id ='PET'      THEN 0.95
    WHEN catg_id ='SPG'      THEN 0.95
    WHEN catg_id ='APL'      THEN 0.90
    WHEN catg_id ='XXX'      THEN 'Invalid'
    ELSE  1
END "Savings %"
from prd_products
order by catg_id
;
SQL Error: ORA-00932: inconsistent datatypes: expected NUMBER got CHAR
```

1.2. Including other functions

Demo 07: We can then include the round function to improve the results.

```
select catg_id, prod_id, prod_list_price
, Round(
CASE
```

```

        WHEN catg_id = 'PET'      THEN 0.95
        WHEN catg_id = 'SPG'      THEN 0.95
        WHEN catg_id = 'APL'      THEN 0.90
    ELSE 1
    END * prod_list_price, 2 ) AS "Today's Price"
from prd_products
order by catg_id;
--- selected rows shown

```

CATG_I	PROD_ID	PROD_LIST_PRICE	Today's Price
APL	4569	349.95	314.96
APL	1120	549.99	494.99
APL	1130	149.99	134.99
APL	1126	850	765

In the next example we want the discount to apply only to products with a list price of \$50 or higher. The first When clause with a true value determines the result

Demo 08: –The first When clause with a true value determines the result. Items with prices under \$50 are not considered for a discount. This one uses the To_Char function to format the column.

```

select catg_id, prod_id, prod_list_price
, To_Char(
CASE
    WHEN prod_list_price < 50 THEN 1
    WHEN catg_id = 'PET'      THEN 0.95
    WHEN catg_id = 'SPG'      THEN 0.95
    WHEN catg_id = 'APL'      THEN 0.90
    ELSE 1
    END * prod_list_price, '9999.00') AS "Today's Price"
from prd_products
order by catg_id
;
--- selected rows

```

CATG_I	PROD_ID	PROD_LIST_PRICE	Today's Price
APL	4569	349.95	314.96
APL	1120	549.99	494.99
APL	1130	149.99	134.99
APL	1126	850	765.00
APL	1125	500	450.00
GFD	5001	5	5.00
GFD	5000	12.5	12.50
HD	5002	23	23.00
HD	5005	45	45.00
HD	5004	15	15.00
HD	5008	12.5	12.50

The next case structure looks daunting in code but look at the output first. With appliances we merely report back that this is an appliance item. With pet supplies and sporting good we break these down into cost categories (high, low, medium). The break points for sporting goods and pet supplies are different. For all other categories we do not report anything.

The outer case structure is based on the category id- there is a block for PET, another block for SPG, a third block for APL and there is no Else block. Items which do not fit in one of these categories do not get a block and the case returns a null. When you develop this code, you should write and test the outer case structure first.

The inner case structure for PET and the inner case structure for SPG are based on the prod_list_price

Demo 09: A nested Case structure using prd_products

```

select catg_id, prod_id, prod_list_price
, CASE
    WHEN catg_id = 'PET' THEN
        CASE
            WHEN prod_list_price < 10 THEN 'LowCost pet item'
            ELSE 'HighCost pet item'
        END
    WHEN catg_id = 'SPG' THEN
        CASE
            WHEN prod_list_price < 25 THEN 'LowCost sports item'
            WHEN prod_list_price between 25 and 150 THEN 'MidCost sports item'
            ELSE 'HighCost sports item'
        END
    WHEN catg_id = 'APL' THEN 'appliance item'
END AS "Result"
from prd_products
order by prod_id;
--- selected rows shown

```

CATG_I	PROD_ID	PROD_LIST_PRICE	Result
HW	1000	125	
SPG	1010	150	MidCost sports item
SPG	1020	12.95	LowCost sports item
SPG	1030	29.95	MidCost sports item
SPG	1040	349.95	HighCost sports item
HW	1090	149.99	
HW	1100	49.99	
APL	1120	549.99	appliance item
APL	1130	149.99	appliance item
PET	1140	14.99	HighCost pet item
PET	1141	99.99	HighCost pet item
PET	1142	2.5	LowCost pet item
PET	1152	55	HighCost pet item
HW	1160	149.99	
PET	4567	549.99	HighCost pet item
PET	4568	549.99	HighCost pet item
APL	4569	349.95	appliance item
HW	4575	49.95	
PET	4577	29.95	HighCost pet item

If we want to display a message instead of the missing value, we can wrap a coalesce function around the entire case expression.: Coalesce(CASE . . . END, 'No information available') as "Result"

Demo 10: We have a look up table for the credit ratings. This is another approach. If the credit levels for the rating terms were to change frequently, the lookup table would be a better approach.

```

select customer_id, customer_credit_limit
, CASE
    WHEN customer_credit_limit >= 10001 THEN 'Superior'
    WHEN customer_credit_limit >= 5001 THEN 'Excellent'
    WHEN customer_credit_limit >= 2001 THEN 'High'
    WHEN customer_credit_limit >= 1001 THEN 'Good'
    ELSE 'Standard'
END AS Rating
from cust_customers;
--- selected rows shown

```

CUSTOMER_ID	CUSTOMER_CREDIT_LIMIT	RATING

001250	750	Standard
001890	1750	Good
002100	750	Standard
002110	750	Standard
002120	750	Standard
002500		Standard
003000	6000	Excellent
003500	6000	Excellent
003750	6000	Excellent
003760	6000	Excellent
004000	3500	High
004100	3500	High

2. Simple Case Expression

Oracle has another version of the Case expression called a Simple Case expression.

Demo 11: Simple case; only one attribute is being compared; the comparisons are all equality tests.

```
select catg_id, prod_id, prod_list_price
, CASE catg_id
    WHEN 'PET' THEN 0.95
    WHEN 'SPG' THEN 0.95
    WHEN 'APL' THEN 0.90
    ELSE 1
END * prod_list_price AS "Today's Price"
from prd_products;
```

--- selected rows shown

CATG_I	PROD_ID	PROD_LIST_PRICE	Today's Price
HW	1000	125	125
SPG	1010	150	142.5
SPG	1020	12.95	12.3025
HW	1090	149.99	149.99
HW	1100	49.99	49.99
PET	1140	14.99	14.2405
PET	1141	99.99	94.9905
PET	1142	2.5	2.375
SPG	1040	349.95	332.4525
SPG	1050	269.95	256.4525
SPG	1060	255.95	243.1525
HW	1160	149.99	149.99
PET	4567	549.99	522.4905
APL	4569	349.95	314.955
HW	4575	49.95	49.95
APL	1125	500	450
APL	1126	850	765
APL	1130	149.99	134.991
GFD	5000	12.5	12.5
GFD	5001	5	5
HD	5002	23	23
HD	5004	15	15

Demo 12: Organizing sales by season

```
select order_id, To_char(order_date, 'yyyy-mm-dd') AS OrderDate
, CASE TO_CHAR(order_date, 'q')
    WHEN '1' THEN 'winter'
```

```

WHEN '2'    THEN 'spring'
WHEN '3'    THEN 'summer'
WHEN '4'    THEN 'fall'
END AS "Season"
from oe_orderHeaders ;

```

--- selected rows shown

ORDER_ID	ORDERDATE	Season
540	2015-06-02	spring
390	2015-06-04	spring
395	2015-06-04	spring
609	2015-09-27	summer
610	2015-09-27	summer
611	2015-09-30	summer
105	2015-10-01	fall
106	2015-10-01	fall
402	2015-10-18	fall
405	2015-11-19	fall

Demo 13: Using a case to do a special sort. We want to sort the products by the categories but not alphabetically. The order we want to use is PET, SPG, APL, HW.

```

select catg_id, prod_id, prod_list_price
from prd_products
order by CASE catg_id
        WHEN 'PET'    THEN 1
        WHEN 'SPG'    THEN 2
        WHEN 'APL'    THEN 3
        WHEN 'HW'     THEN 4
        ELSE      9999
END,
catg_id, prod_id;

```

--- selected rows shown

CATG_I	PROD_ID	PROD_LIST_PRICE
PET	1140	14.99
PET	1141	99.99
PET	1142	2.5
PET	1150	4.99
PET	1151	14.99
PET	1152	55
SPG	1030	29.95
SPG	1060	255.95
APL	1120	549.99
APL	1125	500
APL	1126	850
HW	1090	149.99
HW	1100	49.99
HW	1110	49.99
HW	1160	149.99
HW	4575	49.95
GFD	5000	12.5
GFD	5001	5
HD	5005	45
HD	5008	12.5

Table of Contents

1. GREATEST and LEAST	1
1.1. Examples using the demo tables	2
2. DECODE	3
3. Things to watch out for with Decode	5
3.1. Missing else value	5
3.2. Return type problems	5
3.3. Old code styles	7

1. GREATEST and LEAST

GREATEST and **LEAST** return the largest and smallest value from the list of arguments. Notice what happens with nulls. If there is a null in the list then the functions treat this as an unknown value and therefore the function cannot “know” which value in the list is the largest.

Warning: in a future unit we discuss the functions Max and Min which sound very much like Greatest and Least. They are NOT the same. Greatest and Least are row functions- **they work on multiple columns in a single row of data**. You can use them with literals. Max and Min work down a column over a set of rows.

Demo 01: Greatest, Least.

```
select A
, GREATEST(B,C,D,E,F)
, LEAST(B, C, D, E, F)
from z_tst_numbers;
```

A	GREATEST(B,C,D,E,F)	LEAST(B,C,D,E,F)
1	90	10
2		
3		
4		
5	0	0
6	10	10
7	85	-210
8	85	-210
9	200	-1
10	200	-1

Data type Issues: A function returns a single value. **GREATEST** (list), **LEAST** (list) returns the largest, smallest in the list. If the list is of mixed data type, the data type of the first argument is used and other elements of the list are converted to that data type. You may wish to use a conversion function to force a specific data type for the first argument.

Demo 02: greatest, least with numbers

```
select
GREATEST(4, 45.78, 9, -333.98)
, LEAST(4, 45.78, 9, -333.98)
from dual;
```

GREATEST(4,45.78,9,-333.98)	LEAST(4,45.78,9,-333.98)
45.78	-333.98

Demo 03: greatest, least with strings; the first argument is a string, so all of the other arguments are cast to strings. Note that the least function returns the string '4', not the number 4.

```
select
  greatest( 'flea', 4, 45.78, 9, 'elephant', 9)
, least   ( 'flea', 4, 45.78, 9, 'elephant', 9)
from dual;
GREATER
-----
flea 4
```

Demo 04: Data types matter; as numbers 25 is smaller than 125; as string '125' sorts before '25' and is considered least.

```
select least(25, 125) as Numbers, least('25', '125') as Strings
from dual;
NUMBERS STR
-----
25 125
```

Demo 05: This has a first argument that is numeric; when the function gets to the argument 'elephant' that cannot be cast to a number and we get an error.

```
select greatest(4, 45.78, 9, 'elephant') from dual;
select greatest(4, 45.78, 9, 'elephant')
*
ERROR at line 1:
ORA-01722: invalid number
```

1.1. Examples using the demo tables

Demo 06: This query returns the largest of the two columns quoted_price and prod_list_price

```
select order_id
, prod_id
, quoted_price
, prod_list_price
, GREATEST (quoted_price, prod_list_price)  as HigherPrice
from oe_orderDetails
join prd_products using (prod_id)
where prod_id in (1000, 1010);
```

ORDER_ID	PROD_ID	QUOTED_PRICE	PROD_LIST_PRICE	HIGHERPRICE
528	1010	150	150	150
390	1010	175	150	175
395	1010	195	150	195
303	1000	125	125	125
313	1000	125	125	125
550	1010	175	150	175
551	1010	175	150	175
605	1010	125	150	150
608	1000	100	125	125
609	1010	175	150	175
105	1010	150	150	150
405	1010	150	150	150
115	1000	100	125	125
2120	1010	175	150	175
2121	1010	175	150	175

3808	1000	100	125	125
3808	1010	125	150	150

17 rows selected.

Demo 07: The Greatest function just returns a number. You probably want to know which is bigger; that uses a Case. (In our tables these price columns are not null, but that is not true for all tables. So you always have to think about those nulls.)

```
select order_id
, prod_id
, quoted_price as "quoted"
, prod_list_price as "List"
, GREATEST (quoted_price, prod_list_price) as "higher"
, case when quoted_price = prod_list_price then 'same price'
       when quoted_price > prod_list_price then 'quoted is higher'
       when quoted_price < prod_list_price then 'list is higher'
       else 'one or more is null'
     end "PriceComparison"
from oe_orderDetails
join prd_products using (prod_id)
where prod_id in (1000, 1010)
;
```

ORDER_ID	PROD_ID	quoted	List	higher	PriceComparison
528	1010	150	150	150	same price
390	1010	175	150	175	quoted is higher
395	1010	195	150	195	quoted is higher
303	1000	125	125	125	same price
313	1000	125	125	125	same price
550	1010	175	150	175	quoted is higher
551	1010	175	150	175	quoted is higher
605	1010	125	150	150	list is higher
608	1000	100	125	125	list is higher
609	1010	175	150	175	quoted is higher
105	1010	150	150	150	same price
405	1010	150	150	150	same price
115	1000	100	125	125	list is higher
2120	1010	175	150	175	quoted is higher
2121	1010	175	150	175	quoted is higher
3808	1000	100	125	125	list is higher
3808	1010	125	150	150	list is higher

2. DECODE

The Decode function is included here for completeness and because you may find it in old code and need to figure it out. It is generally a good idea to avoid this in new code. (I do not like it when more than half of this document is taken up by a technique that I do not even want you to use. But Decode exists and is not obvious. I do not have any queries in the assignments or exams where Decode is the appropriate choice.)

The Decode function is used to provide an If-Then-Else logic within a SQL statement. The first argument is an expression to be evaluated. Then you have pairs of arguments; the first of the pair is a possible value for the expression, the second of the pair is the return value. The last, optional, argument is a value to return if the expression is not matched by any of the pairs. Decode is limited to exact matches.

The Decode function can be difficult to debug if it is nested or if the pairings are not obvious. But it is a very common function in older Oracle code. Decode is an Oracle specific function. Use Case to produce more portable code.

Demo 08: We want to give customers a 5% savings for each pet supply item, 5% for each sporting goods item and 10% for each appliance. These savings are taken against the list price

```
select catg_id, prod_id, prod_list_price
, DECODE (catg_id,
      'PET', 0.95,
      'SPG', 0.95,
      'APL', 0.90,
      1.00 ) * prod_list_price AS "Today's Price"
from prd_products;
--- selected rows shown
```

CATG_I	PROD_ID	PROD_LIST_PRICE	Today's Price
HW	1000	125	125
SPG	1010	150	142.5
SPG	1020	12.95	12.3025
SPG	1030	29.95	28.4525
HW	1072	25.5	25.5
HW	1080	25	25
HW	1090	149.99	149.99
HW	1100	49.99	49.99
PET	1140	14.99	14.2405
PET	1141	99.99	94.9905
PET	1142	2.5	2.375

Demo 09: Use Decode to interpret small collections of coded values that seldom change.

```
select Order_Status
, Decode(order_Status,
0, 'New',
1, 'Produced',
2, 'Picked',
3, 'Shipped',
9, 'BackOrdered',
'Invalid Code') AS Order_Status
from oe_orderHeaders;
--- selected rows shown
```

ORD_STATUS	ORDER_STATUS
9	BackOrdered
9	BackOrdered
9	BackOrdered
2	Picked
2	Picked
9	BackOrdered
3	Shipped
3	Shipped
4	Invalid Code
7	Invalid Code
...	rows omitted

```
/* using case*/
select
Order_Status
, case order_Status
when 0 then 'New'
when 1 then 'Produced'
when 2 then 'Picked'
when 3 then 'Shipped'
```

```

when 9 then 'BackOrdered'
else 'Invalid Code'
end As Order_Status
From oe_orderHeaders
;

```

If this type of descriptive term is subject to change then it is a better idea to put these into a table and do a join to get the descriptive values (such as we do with the prd_categories table). Suppose you used the previous decode function in a series of queries that were then built into application programs. Five years later the company decides to use ord_status 4 and has a description for it- or decides that the label 'Produced' should be replaced with another term. Now someone would have to find all of those queries that were embedded in all of those application programs that use this decode function and change them and test them and update all of the applications.

3. Things to watch out for with Decode

3.1. Missing else value

Demo 10: This is the first decode demo but I have skipped the "else" value. That case Decode returns a null and it looks like a lot of products do not have a price value for the last column. You do not want to do this. The business rule was to give a discount for certain types of items; in a business situation that would not mean that the price of HW item. was unknown

You should be aware when a function, such as Decode, will produce a null as output but you should try to avoid those situations.

```

select catg_id, prod_id, prod_list_price
, DECODE (catg_id,
    'PET', 0.95,
    'SPG', 0.95,
    'APL', 0.90 ) * prod_list_price AS "Today's Price"
from prd_products;
--- selected rows shown

```

CATG_ID	PROD_ID	PROD_LIST_PRICE	Today's Price
HW	1000	125	
SPG	1010	150	142.5
SPG	1020	12.95	12.3025
SPG	1030	29.95	28.4525
HW	1072	25.5	
HW	1080	25	
HW	1090	149.99	
HW	1100	49.99	
PET	1140	14.99	14.2405
PET	1141	99.99	94.9905
PET	1142	2.5	2.375

3.2. Return type problems

The return value type is determined by the first return value in the Decode.

Demo 11: The following will run, the return type will be varchar2 and the numeric value 345 will be cast to the string '345'

```

select
    Order_Status
, DECODE(order_Status,
    0, 'New',

```

```

1, 'Produced',
2, 'Picked',
3, 'Shipped',
9, 345,
'Invalid Code') As OrderStatus
from oe_orderHeaders
Order By order_status
--- selected rows shown

```

ORD_STATUS	ORDER_STATUS
1	Produced
1	Produced
2	Picked
2	Picked
2	Picked
3	Shipped
3	Shipped
4	Invalid Code
4	Invalid Code
5	Invalid Code
5	Invalid Code
7	Invalid Code
7	Invalid Code
9	345
9	345

Demo 12: This will not run. The return type is determined by the first possible return value (in this query 150) which is numeric and the other values such as "Produced" cannot be cast to a number

```

select Order_Status
, Decode(order_Status,
    0, 150,
    1, 'Produced',
    2, 'Picked',
    3, 'Shipped',
    9, 345,
    'Invalid Code') AS OrderStatus
from oe_orderHeaders
order by order_Status
;

```

SQL Error: ORA-01722: invalid number

Where this can get very confusing is when you have a Decode that sometimes works and sometimes does not. Suppose we had the following decode expression where status is some numeric column in the table.

```

Decode(order_Status,
    0, 150,
    1, 250,
    2, 350,
    9, 'Invalid status',
    'Invalid Code')

```

If the table contains only the values 0, 1, 2 in the status column this will work. But if a new row is added with a value for status other than 0, 1, or 2 then the query will fail. You need to write queries that will not fail for any legitimate data in the table.

You can try this with a CTE that gets only rows with order_Status 0,1, and 2. and then use those rows in the main query with the above decode.

```
With tbl as (
    select *
    from oe_orderHeaders
    where order_status in ( 0,1,2 )
)
select distinct
    order_status
, Decode(order_status,
    0, 150,
    1, 250,
    2, 350,
    9, 'Invalid status',
    'Invalid Code')  as DecodeVal
from tbl;
```

ORD_STATUS	DECODEVAL
1	250
2	350

But if you change the cte to allow other values, then the query fails. Change the IN list in the CTE to where ord_status in (0,1,2, 3,4) and then the query produces an error message.

```
'Invalid Code')  as DecodeVal
*
ERROR at line 13:
ORA-01722: invalid number
```

The problem is that the decode takes its data type to return from the first pair- which in this case is a number(150). When the query gets to a 3 or 4, then it wants to return 'Invalid Code' which cannot be cast to a number.

3.3. Old code styles

You would not normally be writing Decode in queries you write today but you might have to maintain old SQL and you might come across very complex logic done with Decode. This is a very simple example, but it might not be obvious.

Demo 13: Nested decode using the Sign function

```
select
    catg_id
, prod_id
, prod_list_price
, decode(catg_id,
    'PET',
    decode(sign(prod_list_price - 10),
        -1, 'LowCost pet item',
        'HighCost pet item'
    ),
    'SPG',
    decode(sign(prod_list_price - 25),
        -1, 'LowCost sports item',
        decode(sign(prod_list_price - 150),
            -1, 'MidCost sports item',
            0, 'MidCost sports item',
            1, 'HighCost sports item'
        )
    ),
    'APL', 'appliance item'
```

```

    ) AS "Result"
  from prd_products
  where prod_list_price <=
    decode (catg_id,
      'PET', 100,
      'SPG', 300,
      'APL', 400,
      'MUS', 10,
      75)
  order by catg_id, prod_list_price
;

```

CATG_ID	PROD_ID	PROD_LIST_PRICE	Result
APL	1130	149.99	appliance item
APL	4569	349.95	appliance item
GFD	5001	5	
GFD	5000	12.5	
HD	5008	12.5	
HD	5004	15	
HD	5002	23	
HD	5005	45	
HW	1100	49.99	
HW	1110	49.99	
MUS	2487	9.45	
MUS	2412	9.87	
PET	1143	2.5	LowCost pet item
PET	1142	2.5	LowCost pet item
PET	1150	4.99	LowCost pet item
PET	1140	14.99	HighCost pet item
PET	1151	14.99	HighCost pet item
SPG	1020	12.95	LowCost sports item
SPG	1030	29.95	MidCost sports item
SPG	1010	150	MidCost sports item
SPG	1060	255.95	HighCost sports item
SPG	1050	269.95	HighCost sports item

If you see nested decodes in a query, you need to investigate it carefully. The use of the sign function in a decode is generally a technique for checking if a number is equal to, less than or greater than a specific number.

The use of case expressions here is easier to use and easier to maintain.

Table of Contents

1. Data types for temporal data.....	1
2. How Date values are stored	1
3. How Date values are written and displayed	1
4. TO_DATE and ANSI DATE literals.....	2
5. Getting the current date	3
6. TimeStamp data	3
7. Assumptions we make and why we shouldn't.....	4

1. Data types for temporal data

Oracle has a few data types for temporal data

- Date: which stores both a date and a time with a precision of second
- TimeStamp: which stores both a date and a time with a precision of fractional seconds
- Interval Year To Month - this includes a precision for years of 0-9 digits with a default of 2 digits.
A precision of 2 digits allows a maximum interval of 99 years and 11 months. The month component is integral.
- Interval Day To Second - this stores data as day, hours, minutes, seconds, and fractions of a second
The way that Oracle has implemented temporal intervals is not very intuitive and causes errors where it really shouldn't. I am not going to discuss time intervals in this class.

We don't really need the accuracy of TimeStamp for this class- so I only mention it briefly.

So we will focus on the Date data type.

2. How Date values are stored

The Oracle DATE data type stores values that include **a date and a time component** for each value. When Oracle stores a DATE value, it stores the value in a proprietary format as 7 components: Century, Year, Month, Day, Hour, Minute, Second. Oracle can handle dates from 4712 BC to 9999 CE and times from 0:00:00 to 23:59:59.

The earliest date for Oracle is January 1, 4712 BC.

3. How Date values are written and displayed

The people who develop a dbms and client software have to make some decisions about the default display for data. Oracle's decision for date values is to display the day using two digits, the month using three characters and the year using two digits. So if I do

```
select sysdate from dual;
```

I get back a display such as the following. The time component in that value for sysdate exists but it isn't displayed.

SYSDATE

13-FEB-16

If you look at the inserts I gave you for the vets set of tables, the animal dob in the animals table was given using just the year-month-day part

```
insert into vt_animals ( an_id, cl_id, an_name, an_type, an_dob)
values(15165, 411, 'Burgess', 'dog', date '2005-11-20' );
```

And the inserts used for the exam_date in the exam headers table had a time component.

```
insert into vt_exam_headers(ex_id, an_id, stf_id, ex_date)
values (3105, 17002, 29, TO_DATE('2014-10-10 9:15' , 'YYYY-MM-DD HH24:MI') );
```

And I had to do more work to specify that.

Both of the columns were simply defined as date type. The values for the an_dob have the time set to midnight. And I included a constraint that you could not use a different time component for the an_dob.

```
constraint an_dob_ck      check (trunc(an_dob) = an_dob)
```

Look at that constraint again after you read about the temporal functions.

If we display exam date values, we get the default display and the time component is not displayed.

```
select * from vt_exam_headers;
```

EX_ID	AN_ID	STF_ID	EX_DATE
2290	21320	38	11-APR-15
2300	21316	38	08-MAY-15
2352	10002	38	12-MAY-15
2389	21006	38	20-MAY-15
2400	12038	38	02-JUN-15
...			

In the document on Conversion function in this unit I show you how to display dates in a variety of formats. In this document I will show you how to change the default date format for a session. (If you decide to do that for an assignment, you need to include that command in your script file.

4. TO_DATE and ANSI DATE literals

With Oracle, you can specify a date value with an ANSI date literal, such as Date '2003-08-15'. You need to use the format 'YYYY-MM-DD'. ANSI date literals do not include a time component.

Demo 01: The following and the next 4 demos create a table and insert 14 rows of date values.

```
create table z_tst_cal ( ID NUMBER(2), MyDate DATE);
```

Demo 02: -- Using the To_Date function

```
insert into z_tst_cal values ( 1, TO_DATE('1/1/2003', 'MM/DD/YYYY') );
insert into z_tst_cal values ( 2, TO_DATE('2/28/2003', 'MM/DD/YYYY') );
insert into z_tst_cal values ( 3, TO_DATE('2003-05-11', 'YYYY-MM-DD') );
insert into z_tst_cal values ( 4, TO_DATE('2003-jun-05', 'YYYY-MON-DD') );
insert into z_tst_cal values ( 5, TO_DATE('20030908', 'YYYYMMDD') );
```

Demo 03: -- Specifying a time component

```
insert into z_tst_cal values ( 6, TO_DATE('2003-08-19 1200pm', 'YYYY-MM-DD HHMIAM') );
insert into z_tst_cal values ( 7, TO_DATE('2003-08-19 0600am', 'YYYY-MM-DD HHMIAM') );
insert into z_tst_cal values ( 8, TO_DATE('2003-08-19 1200am', 'YYYY-MM-DD HHMIAM') );
insert into z_tst_cal values ( 9, TO_DATE('2003-08-19 0600pm', 'YYYY-MM-DD HHMIAM') );
```

Demo 04: -- Specifying BC or AD

```
insert into z_tst_cal values (10, TO_DATE('40000101BC', 'YYYYMMDDBC') );
insert into z_tst_cal values (11, TO_DATE('99991231AD', 'YYYYMMDDBC') );
```

Demo 05: -- Using DATE

```
insert into z_tst_cal values (12, DATE '2003-06-25' );
insert into z_tst_cal values (13, DATE '2003-6-5' );
insert into z_tst_cal values (14, DATE '03-06-25' );
```

Demo 06: This shows the values in the table, using two different display formats.

```
select ID
, TO_CHAR (MYDATE, 'YYYY-MM-DD A.D. HHMIAM') AS COL2
, TO_CHAR (MYDATE, 'DD MON, YYYY') AS COL3
from z_tst_cal;
```

ID	COL2	COL3
1	2003-01-01 A.D. 1200AM	01 JAN, 2003
2	2003-02-28 A.D. 1200AM	28 FEB, 2003
3	2003-05-11 A.D. 1200AM	11 MAY, 2003
4	2003-06-05 A.D. 1200AM	05 JUN, 2003
5	2003-09-08 A.D. 1200AM	08 SEP, 2003
6	2003-08-19 A.D. 1200PM	19 AUG, 2003
7	2003-08-19 A.D. 0600AM	19 AUG, 2003
8	2003-08-19 A.D. 1200AM	19 AUG, 2003
9	2003-08-19 A.D. 0600PM	19 AUG, 2003
10	4000-01-01 B.C. 1200AM	01 JAN, 4000
11	9999-12-31 A.D. 1200AM	31 DEC, 9999
12	2003-06-25 A.D. 1200AM	25 JUN, 2003
13	2003-06-05 A.D. 1200AM	05 JUN, 2003
14	0003-06-25 A.D. 1200AM	25 JUN, 0003

5. Getting the current date

Sysdate returns the current date and time of the operating system on which the database resides. Current_date is the date and time of the user's session

Current_timestamp is another way to get the date and time.

Demo 07:

```
select sysdate from dual;
```

```
-----
```

```
13-FEB-16
```

```
select current_date from dual;
```

```
-----
```

```
13-FEB-16
```

```
select current_timestamp from dual;
```

-- using the SQL *Plus client

```
CURRENT_TIMESTAMP
```

```
-----
```

```
13-FEB-16 06.07.45.468000 PM -08:00
```

The display will be different in SQL Developer.

```
select to_char( sysdate, 'YYYY-MM-DD HHMIAM') from dual;
```

```
TO_CHAR(SYSDATE, '
```

```
-----
```

```
2016-02-13 0609PM
```

```
select to_char( current_date, 'YYYY-MM-DD HHMIAM') from dual;
```

```
TO_CHAR(CURRENT_D
```

```
-----
```

```
2016-02-13 0609PM
```

6. TimeStamp data

The timestamp data type will store date values to fractional parts of a second, with a precision of 0 to 9 digits. Sysdate returns the system date as a date type; to get a current time as a timestamp value use SysTimeStamp.

Example of the Timestamp data type.

Demo 08: If you run this, copy and paste the first two inserts as a pair. Then do the other inserts one at a time rather than paste them all at once. Maybe go get a cup of coffee between some of the inserts.

```
create table z_tst_timeStamp (id number, simple_date DATE, time_stamp_date TIMESTAMP);

insert into z_tst_timeStamp values (1, SYSDATE, systimestamp);
insert into z_tst_timeStamp values (2, SYSDATE, systimestamp);

insert into z_tst_timeStamp values (3, SYSDATE, systimestamp);
insert into z_tst_timeStamp values (4, SYSDATE, systimestamp);
commit;
```

Demo 09: sample table data

```
select ID
, to_char(simple_date, 'YYYY-MM-DD HHMIAM') as "Simple Date"
, time_stamp_date as "TimeStamp Date"
from z_tst_timeStamp;
```

ID	Simple Date	TimeStamp Date
1	2016-02-13 0611PM	13-FEB-16 06.11.06.104000 PM
2	2016-02-13 0611PM	13-FEB-16 06.11.06.125000 PM
3	2016-02-13 0611PM	13-FEB-16 06.11.46.826000 PM
4	2016-02-13 0612PM	13-FEB-16 06.12.30.071000 PM

In this case the first rows have the same value for the Simple Date column within a minute. The timestamp values are different even with a copy and paste of the two inserts. Timestamp is a more precise type.

7. Assumptions we make and why we shouldn't

This is semi-optional- which means I want you to read this and think about it- but it is not on the test or assignments.

Since it is assumed for this class that we are all working on the same Oracle system and that it has been set up with certain defaults, we are working under certain assumptions. And for this class, that is OK. But we should at least mention some issues in this area.

Demo 10: Suppose you want to get all the animals that were born in the month of March. Trust me- this is a terrible idea but you will see it in a lot of old sql.

```
select an_id, an_dob
from vt_animals
where an_dob like '%MAR%';
```

AN_ID	AN_DOB
15401	15-MAR-10
21005	06-MAR-11
21205	30-MAR-15

Demo 11: Or you want to find animals born after the 15th day of any month. This is also a terrible way to do this.

```
select an_id, an_dob
from vt_animals
where substr(an_dob, 1, 2) > 15;
```

```

AN_ID AN_DOB
-----
11015 23-FEB-12
12035 28-FEB-95
12038 29-APR-12
15165 20-NOV-05
16002 25-MAY-15
21001 22-MAY-09
21003 18-JUN-14
21205 30-MAR-15
21305 27-JUL-14
21306 27-JUL-14
21307 27-JUL-14

```

11 rows selected

These seem to work and seem pretty clever.

But then someone changes the default date format to match the standard- you can do this on a session basis or the dba can do this for the system.

Demo 12:

```

alter session set nls_date_format= 'YYYY-MM-DD';
Session altered

```

If you run those two queries again- it looks like no animals were born in March and a lot more animals were born after the 15th of the month.

```

SQL> select an_id, an_dob
  2  from vt_animals
  3 where an_dob like '%MAR%';

no rows selected

SQL> select an_id, an_dob
  2  from vt_animals
  3 where substr(an_dob, 1, 2) > 15;

AN_ID AN_DOB
-----
10002 2010-04-15
11015 2012-02-23
11025 2012-02-01
11028 2015-10-01
11029 2015-10-01

38 rows selected.

```

What those queries are doing is treating the date value as a string and then doing string manipulation. It is better to use functions which are designed to handle dates to do date manipulation. (See the next document for how to do these correctly.)

Date values are not strings; don't pretend they are strings

The result of a To_CHAR function is a string.

You can reset the nls_date_format back - or just log out and start a new session.

```
alter session set nls_date_format='DD-MON-YY';
```

Table of Contents

1.	Extracting part of a date value	1
2.	Date arithmetic	2
3.	Date manipulation functions.....	3
3.1.	Add_Months.....	3
3.1.	Months_Between.....	3
3.2.	Last_Day	3
3.3.	Next_Day.....	4
3.4.	Using To_Char to extract part of a date	4
4.	Using Round, Trunc, Greatest, Least with date values	5
5.	Things you should not do with dates	6

This document is focusing on the functions that Oracle provides to deal with dates. You should use these functions rather than trying to build your own functions or expressions. For example, if you want to get the month of a date value as the month named spelled out, you use

```
select To_char(sysdate, 'FMMonth') from dual;
TO_CHAR(S
-----
February
```

You do not extract the month as number and write a 12 part case expression to get the month name.

If you want to find a date two months after a specified date, you use the Add_Months function.

1. Extracting part of a date value

Demo 01: Extract date part

```
select hire_date
, EXTRACT (YEAR FROM hire_date) AS "YearHired"
, EXTRACT (MONTH FROM hire_date) AS "MonthHired"
, EXTRACT (DAY FROM hire_date) AS "DayHired"
from emp_employees
where rownum < 4;
```

HIRE_DATE	YearHired	MonthHired	DayHired
17-JUN-89	1989	6	17
30-MAR-01	2001	3	30
28-OCT-01	2001	10	28

Demo 02: We want the people hired in August of any year.

```
select emp_id, hire_date
from emp_employees
where EXTRACT (MONTH FROM hire_date) = 8;
```

EMP_ID	HIRE_DATE
201	25-AUG-04
103	01-AUG-10

Demo 03: Who has been hired in the current year?

```
select emp_id
, hire_date
from emp_employees
where EXTRACT (YEAR FROM hire_date) = EXTRACT (YEAR FROM sysdate);
no rows selected
```

Demo 04: Who has been hired in the year three years ago?

```
select emp_id
, hire_date
from emp_employees
where EXTRACT (YEAR FROM hire date) = EXTRACT (YEAR FROM sysdate) - 3;
EMP_ID HIRE_DATE
-----
206 15-JUN-13
204 15-JUN-13
```

Sometimes when you talk about date arithmetic, you have to double check the meaning. This is not always what people would mean by "who was hired in the last three years".

2. Date arithmetic

You can do arithmetic with date values. You can add or subtract a date value and a number. You can subtract two date values. You cannot do multiplication or division with date values.

From the example below you can see that the basic unit of time in Oracle is the day- if we add 1 to a date value we are adding 1 day. Note that this approach to date arithmetic is not common in other systems.

Demo 05: Date arithmetic.

```
select sysdate
, sysdate + 2 as Plus2
, sysdate - 40 as minus40
, sysdate + 285 as Plus285
from dual
;
SYSDATE PLUS2 MINUS40 PLUS285
-----
13-FEB-16 15-FEB-16 04-JAN-16 24-NOV-16
```

Demo 06: Date arithmetic.

```
select MyDate, MyDate - to_date('15-May-2003') DaysBetween
from z_tst_cal
where ID <=5
;
MYDATE DAYSBETWEEN
-----
01-JAN-03 -134
28-FEB-03 -76
11-MAY-03 -4
05-JUN-03 21
08-SEP-03 116
```

Demo 07: date arithmetic. The results will vary with the date this is run.

```
select emp_id, hire_date
, trunc(SYSDATE - hire_date)as "How Many Days Since Hired"
from emp_employees;
EMP_ID HIRE_DATE How Many Days Since Hired
-----
100 17-JUN-89 9737
145 30-MAR-01 5433
150 28-OCT-01 5221
155 05-MAR-04 4362
201 25-AUG-04 4189
. . . rows omitted
```

3. Date manipulation functions

3.1. Add_Months

Demo 08: ADD_MONTHS(date, count) adds (or subtracts) the specified number of months.

```
select MyDate
, ADD_MONTHS ( MyDate, 2)      AS Col1
, ADD_MONTHS ( MyDate, -1)     AS Col2
from z_tst_cal
where ID <=4
;
```

MYDATE	ADD_MONTHS (MYDATE, 2)	ADD_MONTHS (MYDATE, -1)
01-JAN-03	01-MAR-03	01-DEC-02
28-FEB-03	30-APR-03	31-JAN-03
11-MAY-03	11-JUL-03	11-APR-03
05-JUN-03	05-AUG-03	05-MAY-03

3.1. Months_Between

Demo 09: MONTHS_BETWEEN (date1, date2)—The time between two dates expressed as a decimal value. The fractional part of the return value assumes a 31 day month.

```
select MyDate
, MONTHS_BETWEEN ('15-May-2003', MyDate)           AS Col1
, FLOOR(MONTHS_BETWEEN ('15-May-2003', MyDate) ) AS Col2
from z_tst_cal
where ID <=5
;
```

MYDATE	COL1	COL2
01-JAN-03	4.4516129	4
28-FEB-03	2.58064516	2
11-MAY-03	.129032258	0
05-JUN-03	-.67741935	-1
08-SEP-03	-3.7741935	-4

Demo 10: MONTHS_BETWEEN with order dates.

```
select order_id
, order_date
from oe_orderHeaders
where months_between(date '2015-10-15', order date) between 0.75 and 2;

```

ORD_ID	ORD_DATE
414	20-AUG-15
415	23-AUG-15
605	05-SEP-15
606	07-SEP-15
607	15-SEP-15

3.2. Last_Day

Demo 11: LAST_DAY (date)— returns the last day of the month of the argument.

```
select MyDate
, LAST_DAY ( MyDate)      AS Col1
, LAST_DAY ( MyDate) +1   AS Col2
from z_tst_cal
where ID <=4
;
```

MYDATE	COL1	COL2
01-JAN-03	31-JAN-03	01-FEB-03
28-FEB-03	28-FEB-03	01-MAR-03
11-MAY-03	31-MAY-03	01-JUN-03
05-JUN-03	30-JUN-03	01-JUL-03

3.3. **Next_Day**

Demo 12: NEXT_DAY (date, 'DAYINDICTOR')— returns the next date of the indicated day of the week.
The return date will always be greater than the argument date.

```
select MyDate
, NEXT_DAY ( MyDate, 'SUN')      Col1
, NEXT_DAY ( MyDate, 'Friday')   Col2
, NEXT_DAY ( MyDate - 1, 'SUN')   Col3
, NEXT_DAY ( MyDate - 1, 'Friday') Col4
from z_tst_cal
where ID <=5
;
```

MYDATE	COL1	COL2	COL3	COL4
01-JAN-03	05-JAN-03	03-JAN-03	05-JAN-03	03-JAN-03
28-FEB-03	02-MAR-03	07-MAR-03	02-MAR-03	28-FEB-03
11-MAY-03	18-MAY-03	16-MAY-03	11-MAY-03	16-MAY-03
05-JUN-03	08-JUN-03	06-JUN-03	08-JUN-03	06-JUN-03
08-SEP-03	14-SEP-03	12-SEP-03	14-SEP-03	12-SEP-03

3.4. **Using To_Char to extract part of a date**

Demo 13: Another way to do this- To_Char returns string; add To_Number to get actual numeric values returned. This has to do two conversions if I want to get numbers.

```
select hire_date
, TO_NUMBER(TO_CHAR (hire_date, 'YYYY')) AS "YearHired"
, TO_NUMBER(TO_CHAR (hire_date, 'MM'))   AS "MonthHired"
from emp_employees;
```

Demo 14: Alternate syntax. But note that the To_Char returns a string value which you are now comparing to a numeric value requiring more data casting.

```
select emp_id, hire_date
from emp_employees
where TO_CHAR (hire_date, 'MM') = 8;
```

What do you suppose happens if you decide that you will change the Where clause to do a string to string test avoiding one of the data casting steps.

```
select emp_id, hire_date
from emp_employees
where TO_CHAR (hire_date, 'MM') = '8';
```

4. Using Round, Trunc, Greatest, Least with date values

The TRUNC function will take a date value and truncate the time component to 12:00 am (midnight) of that date. The ROUND function will set the time component to 12:00 a.m. of that date if the time part is before noon and to 12:00 a.m. of the next day if the time component is at or after noon.

If you use GREATEST, LEAST with date literals you need to include TO_DATE function. A date literal such as '12-JAN-02' looks just like a string literal.

Demo 15: The Round function will let you round date values to a specified precision. I had to wrap these dates in a TO_CHAR to show the full date values.

```
select ID
, TO_CHAR( MyDate, 'YYYY-MM-DD BC') Col0
, TO_CHAR( ROUND (MyDate, 'DD'), 'YYYY-MM-DD BC') Col1
, TO_CHAR( ROUND (MyDate, 'YYYY'), 'YYYY-MM-DD BC') Col2
, TO_CHAR( ROUND (MyDate, 'MM'), 'YYYY-MM-DD BC') Col3
from z_tst_cal ;
```

-- selected rows				
ID	COL0	COL1	COL2	COL3
1	2003-01-01 AD	2003-01-01 AD	2003-01-01 AD	2003-01-01 AD
2	2003-02-28 AD	2003-02-28 AD	2003-01-01 AD	2003-03-01 AD
3	2003-05-11 AD	2003-05-11 AD	2003-01-01 AD	2003-05-01 AD
4	2003-06-05 AD	2003-06-05 AD	2003-01-01 AD	2003-06-01 AD
10	4000-01-01 BC	4000-01-01 BC	4000-01-01 BC	4000-01-01 BC
11	9999-12-31 AD	9999-12-31 AD	0000-00-00 00	0000-00-00 00

Demo 16: The TRUNC function will truncate to a specified precision.

```
select ID
, TO_CHAR( MyDate, 'YYYY-MM-DD BC') Col0
, TO_CHAR( TRUNC (MyDate, 'DD'), 'YYYY-MM-DD BC') Col1
, TO_CHAR( TRUNC (MyDate, 'YYYY'), 'YYYY-MM-DD BC') Col2
, TO_CHAR( TRUNC (MyDate, 'MM'), 'YYYY-MM-DD BC') Col3
from z_tst_cal ;
```

-- selected rows				
ID	COL0	COL1	COL2	COL3
1	2003-01-01 AD	2003-01-01 AD	2003-01-01 AD	2003-01-01 AD
2	2003-02-28 AD	2003-02-28 AD	2003-01-01 AD	2003-02-01 AD
3	2003-05-11 AD	2003-05-11 AD	2003-01-01 AD	2003-05-01 AD
4	2003-06-05 AD	2003-06-05 AD	2003-01-01 AD	2003-06-01 AD
10	4000-01-01 BC	4000-01-01 BC	4000-01-01 BC	4000-01-01 BC
11	9999-12-31 AD	9999-12-31 AD	9999-01-01 AD	9999-12-01 AD

Demo 17: Coerce the first argument to a date value.

```
select ID, MyDate
, GREATEST( TO_DATE ('15-MAY-2003', 'dd-MON-yy'), MyDate)
from z_tst_cal
where ID <=4;
```

ID	MYDATE	GREATEST(
1	01-JAN-03	15-MAY-03
2	28-FEB-03	15-MAY-03
3	11-MAY-03	15-MAY-03
4	05-JUN-03	05-JUN-03

Demo 18: Notice the values returned here- these are string comparisons

```
select ID, MyDate
, GREATEST ('15-MAY-2003', MyDate)
from z_tst_cal
where ID < =4
;

```

ID	MYDATE	GREATEST('15-MAY-2003', MyDate)
1	01-JAN-03	15-MAY-2003
2	28-FEB-03	28-FEB-03
3	11-MAY-03	15-MAY-2003
4	05-JUN-03	15-MAY-2003

Demo 19: Why does this one work correctly?

```
select emp_id, hire_date
, GREATEST (hire_date, date '1990-01-01')
from emp_employees
where rownum < =6;

```

EMP_ID	HIRE_DATE	GREATEST(hire_date, date '1990-01-01')
100	17-JUN-89	01-JAN-90
145	30-MAR-01	30-MAR-01
150	28-OCT-01	28-OCT-01
155	05-MAR-04	05-MAR-04
201	25-AUG-04	25-AUG-04
101	17-JUN-08	17-JUN-08

Demo 20: Be certain that you understand this one.

```
select GREATEST ('5', 90, 456, 23, -4)
, GREATEST (23, '5', 456, 90, -4)
from DUAL;

```

GR	GREATEST(23, '5', 456, 90, -4)
90	456

5. Things you should not do with dates

As I write this it is Feb 13, 2016. I can write a query that gets the current month.

```
select extract(month from sysdate) from dual;

```

EXTRACT(MONTHFROMSYSDATE)
2

Suppose I want to calculate which month it will be two months from now. I could try the following (which is WRONG even if the answer is correct. The query is wrong.)

```
select extract(month from sysdate) +2 from dual;

```

EXTRACT(MONTHFROMSYSDATE) +2
4

I know, two months from the date I write this is April- but if this is correct then I should be able to use this expression to find the month 12 months from now.

```
select extract(month from sysdate) +12 from dual;

```

EXTRACT(MONTHFROMSYSDATE) +12
14

Our calendar does not have a month 14 so this is NOT the way to do date arithmetic. Twelve months from now is Feb (month 2). Our calendar is cyclic; after month 12 comes month 1 - not month 13. Some people will try to fix this by using a mod approach . But you can do this more clearly by using the date arithmetic functions.

Twelve months from now is Add_months (sysdate, 12)

```
select Add_months(sysdate, 12) from dual;
```

ADD_MONTH

13-FEB-17

And the month, 12 months from now, is

```
select extract(month from Add_months(sysdate, 12) ) from dual;
```

The year 12 months from now is simple. (deriving that calculation yourself is not worth your time.)

```
select extract(year from Add_months(sysdate, 12) ) from dual;
```

The year and the month, 19 months ago is

```
select
    extract(year from Add_months(sysdate, -19) ) as YR
    , extract(month from Add_months(sysdate, -19) ) as MN
from dual;
```

YR	MN
-----	-----
2014	7

Or

```
With CalcDate as (
    select Add_months(sysdate, -19) as dtm from dual
)
select
    extract(year from dtm ) as YR
    , extract(month from dtm ) as MN
from CalcDate;
```

Make the system do the date arithmetic.

After that, why is it ok to calculate the next year as extract(year from sysdate) + 1 ?

Our calendar does not use a cyclic year. We do not have a year where the next year goes to 1.

Table of Contents

1. FX.....	1
2. FM	3

Sometimes getting the exact date format you want is pretty tedious. This is some more tedious stuff that is not too important- until your boss needs a specific date format. When you think about it, Oracle has been putting together tools for business reporting for a long time- and it makes sense that they understand that sometimes you need a specific date format.

The format we discussed for To_Date allow some flexibility in the matching of the string and the date format.

For example: I can use a variety of punctuation marks between the date components and they are all accepted. I can use a 1 or 2 digit month or day value.

Select

```
To_Date ('1969-5/6', 'YYYY/MM/DD') as Col1
, To_Date ('1969.5.6', 'YYYY/MM/DD') as Col2
, To_Date ('1969 5 6', 'YYYY/MM/DD') as Col3
from dual;
```

COL1	COL2	COL3
06-MAY-69	06-MAY-69	06-MAY-69

1. FX

Sometimes that much flexibility is not desirable and you can add the FX Format Model Modifier to the format to say that you want the string to exactly match the format.

If I use the format model 'FXYYYY/MM/DD' none of those strings will match

```
SQL Error: ORA-01861: literal does not match format string
*Cause: Literals in the input must be the same length as literals in
the format string (with the exception of leading whitespace). If the
"FX" modifier has been toggled on, the literal must match exactly,
with no extra whitespace.
```

The following strings will also cause an error

Here we need a 2 digit month and a 2 digit year

```
select To_Date ('1969/5/6', 'FXYYYY/MM/DD') from dual;
```

```
SQL Error: ORA-01862: the numeric value does not match the length of the format item
*Cause: When the FX and FM format codes are specified for an input date,
then the number of digits must be exactly the number specified by the
format code. For example, 9 will not match the format specifier DD but
09 will.
```

Here I have a different literal as the component separator and this fails.

```
select To_Date ('1969-05-06', 'FXYYYY/MM/DD') from dual;
```

This version will work

```
select To_Date ('1969/05/06', 'FXYYYY/MM/DD') from dual;
```

The FX modifier is a toggle switch- so you can turn this specificity on and off during the format. This get a bit dense so don't worry too much about this- but follow along.

A toggle switch is one that turns a setting on and the same toggle turns it off.

So the following says that the string has to match the pattern YYYY/MM/ exactly but the DD part is looser- we could use 1 or 2 digits.

'FXYYYY/MM/FXDD'

The following are all acceptable matches

```
To_Date ('1969/05/6', 'FXYYYY/MM/FXDD')
To_Date ('1969/05/06', 'FXYYYY/MM/FXDD')
To_Date ('1969/05/16', 'FXYYYY/MM/FXDD')
To_Date ('1969/05/ 16', 'FXYYYY/MM/FXDD')
```

These fail

```
To_Date ('1969-05/6', 'FXYYYY/MM/FXDD')      incorrect separator
To_Date ('1969/5/06', 'FXYYYY/MM/FXDD')      month must be 2 digits
```

Try to figure this one out first

```
'YYYYFX/FXMMFX/FXDD'
```

To_Date ('69/05/06', 'YYYYFX/FXMMFX/FXDD')	OK
To_Date ('1969/05/06', 'YYYYFX/FXMMFX/FXDD')	OK
To_Date ('1969/5/6', 'YYYYFX/FXMMFX/FXDD')	OK
To_Date ('1969-05-06', 'YYYYFX/FXMMFX/FXDD')	Fails

The format starts off being loose about the number of digits for the year

```
'YYYY
```

Then it gets exact about the separator character

```
'YYYYFX/
```

Then it gets loose about the number of digits in the month

```
'YYYYFX/FXMM
```

Then it gets exact about the separator character

```
'YYYYFX/FXMMFX/
```

Then it gets loose about the number of digits in the day

```
'YYYYFX/FXMMFX/FXDD'
```

I doubt you would really want to do that- but you might want this model

```
'FXYYYYYFX/FXMMFX/FXDD'
```

When you read this, it says

must be exact about 4 digits for the year	'FXYYYY'
can be loose about the separator character	'FXYYYYYFX/
must be exact about 2 digits for the month	'FXYYYYYFX/FXMM
can be loose about the separator character	'FXYYYYYFX/FXMMFX/FX
must be exact about 2 digits for the day	'FXYYYYYFX/FXMMFX/FXDD'

2. FM

The FM modifier is also a toggle switch- it specifies if Oracle should use trailing blanks after variable length Month and Day values (which makes the date components align in "columns" and leading zero before numbers).

Assuming the English language and the Gregorian calendar, the longest month name is 9 characters (September) and the longest day name is 9 characters (Wednesday).

If you are displaying dates which of these formats do you want?

Version A

Monday July 4, 2011
Wednesday September 7, 2011
Wednesday July 27, 2011

Version B

Monday July 4, 2011
Wednesday September 7, 2011
Wednesday July 27, 2011

Version C

Monday July 04, 2011
Wednesday September 07, 2011
Wednesday July 27, 2011

Version D

MONDAY JULY 04, 2011
WEDNESDAY SEPTEMBER 07, 2011
WEDNESDAY JULY 27, 2011

```

version A: 'FMDay Month DD, YYYY'
version B: 'Day Month FMDD, YYYY'
version C: 'Day Month DD, YYYY'
version D: 'DAY FMMONTH FMDD, YYYY'
```

To read these, think of FM as the fill switch and the default is to fill; so the first time FM is used explicitly it means don't fill (I know, that sounds backwards!); the next time it is used in the format then it means fill.

Version A- don't fill and it stays in that mode

Version B- end fill with blanks until you get to FMDD, so the day numbers are not zero filled

Version C- the default which is to fill all components to a width of 9 for month and day name and zero fill the date

version D- the Day is filled, the month is not filled, and the day number is zero filled. And this is in all caps since the format is in caps.

Version A seems the best format for a date that would be printed on a letter; version C makes it easier to compare the date parts in a column-oriented report. I am not sure who would want version B.

You can use FM with numbers.

```
Select
  To_char (345,      '99999.00')    as c999
  , To_char (345,      '00000.00')    as c000
  , To_char (345,      'FM99999.00')  as cFM999
  , To_char (345,      'FM00000.00')  as cFM000
From dual;
```

C999	C000	CFM999	CFM000
345.00	00345.00	345.00	00345.00

The first column uses '99999.00' and has a single blank followed by 2 blanks for the first 2 digits of the format and then three digits for the value.

The second column uses '00000.00' and has a single blank followed by 5 digits for the value, using leading 0's as necessary.

The third column and fourth columns use the FM modifier which says do not fill with blanks so you do not get that extra blank at the start of the field.

Table of Contents

1. Ambiguous References.....	1
2. Table Aliases.....	1

Someone made a post about table aliases. He could follow the patterns but felt uncertain that he understood them. This is a common situation; we start by following a pattern and the query works. But we are not sure why. This gets into ambiguous references and into joins.

These examples use the vets database tables

1. Ambiguous References

Suppose we have a query that joins the vt_animals table and the vt_exam_headers table. And we want to display the animal id and the exam id. The following will not work. I get an error message that the column name an_id is ambiguous (unclear or inexact because a choice between alternatives has not been made). We have a column named an_id in the animals table and a column named an_id in the vt_exam_headers table. And we did not make it clear which one of these we wanted to use.

```
select an_id, ex_id
from vt_animals
join vt_exam_headers on vt_animals.an_id = vt_exam_headers.an_id;
```

In this example you may say that you don't care because this is an inner join and the value would be the same across the join. But SQL (like most programming systems) is fussy about things like that. So you have to provide a table reference for an_id in the Select. Now vt_animals.an_id is a fully qualified name.

```
select vt_animals.an_id, ex_id
from vt_animals
join vt_exam_headers on vt_animals.an_id = vt_exam_headers.an_id;
```

In the previous query ex_id is not fully qualified but it is not ambiguous since it only occurs in the exam_headers table. You could write the following:

```
select vt_animals.an_id, vt_exam_headers.ex_id
from vt_animals
join vt_exam_headers on vt_animals.an_id = vt_exam_headers.an_id;
```

An ambiguous column reference means that you have a column name in part of the query (it could be in the Select, the Where clause, the Order by clause- or any place else) and the column name appears in more than one of the tables in the From clause. In the tables I use, I try to avoid using the same column name in multiple tables but a lot of companies use the column names "id", "name", "city" etc in many of their tables.

2. Table Aliases

Let's use a single table from the vets database: vt_clients. I want to display a number of columns from this table.

```
select cl_id, cl_name_last, cl_name_first, cl_postal_code, cl_city, cl_phone
from vt_clients
order by cl_id;
```

That will work fine. I have only one table in the From clause, so all of the columns are from the vt_clients table.

Some companies have a rule that queries are written only with fully qualified names. So I write the query as:

```
select vt_clients.cl_id, vt_clients.cl_name_last, vt_clients.cl_name_first,
vt_clients.cl_postal_code, vt_clients.cl_city, vt_clients.cl_phone
from vt_clients
order by vt_clients.cl_id;
```

That certainly made the query longer. I could use a table alias to make the query shorter. A table alias is a substitute name for a table that is defined in the From clause. I often use the alias CL for the vt_clients table.

```
select CL.cl_id, CL.cl_name_last, CL.cl_name_first, CL.cl_postal_code,
CL.cl_city, CL.cl_phone
from vt_clients CL
order by CL.cl_id;
```

That is certainly shorter, the columns are fully qualified and I think it is easier to read. You will find a lot of disagreements on this topic on various web pages. Some people think it is wrong to qualify names when there is only one table in the From clause; some people think it is inefficient to qualify names when there is only one table in the From clause; other people think it is more efficient. When you read these, it usually comes down to a personal choice. People seldom show any data to support their arguments.

Now suppose we want to also show the animal id and animal name along with the client data. We need two tables.

We can do the following with fully qualified names - no table aliases:

```
select vt_animals.an_id, vt_clients.cl_id, vt_clients.cl_name_last,
vt_clients.cl_name_first, vt_clients.cl_postal_code, vt_clients.cl_city,
vt_clients.cl_phone, vt_animals.an_name
from vt_clients
join vt_animals on vt_animals.cl_id = vt_clients.cl_id;
```

Or the following with table aliases:

```
select AN.an_id, CL.cl_id, CL.cl_name_last, CL.cl_name_first,
CL.cl_postal_code, CL.cl_city, CL.cl_phone, AN.an_name
from vt_clients CL
join vt_animals AN on AN.cl_id = CL.cl_id;
```

Again I find the version with table aliases easier to read.

You are not required to qualify all column in the above query- only the ones that would cause an ambiguous reference. The following is OK. The only column name that appears in both tables is cl_id. Many SQL shops would have a style rule against this.

```
select an_id, CL.cl_id, cl_name_last, cl_name_first, cl_postal_code, cl_city,
cl_phone, an_name
from vt_clients CL
join vt_animals AN on AN.cl_id = CL.cl_id;
```

There are a few situation where you are required to use table aliases. One, mentioned in this unit's notes, is when you have to join a table to itself. There is no reasonable example of this with the vets database, so see the example in the notes under self join.

One rule regarding table aliases is that once you define an alias in the From clause you have to use in the rest of that query. We sometimes called the table alias an alternate name- it really is more of a substitute name.

The following will not run. You will get an error message about the column vt_clients.cl_name_last. This is one of those errors that does not seem fair- but that is the way that SQL works.

```
select CL.cl_id, vt_clients.cl_name_last
from vt_clients CL
order by CL.cl_id;
```

There is no rule that table aliases are limited to single letters. The alias can be longer than the table name if that is more meaningful. Our table names are reasonably meaningful, but some companies have table names like Sem234Zghyp and Sem243Zghyp and if you are trying to write a query that joins these two tables I would highly suggest using table aliases. And maybe even look for another place to work unless they have a good reason for those names.

There is a class rule that the table aliases must be meaningful. There is a rule in programming in general that the names of things should suggest their meaning. The use of table aliases a, b, c, d is not allowed (unless each of these is meaningful for that table). One reason people don't like table aliases is that people use aliases that are not meaningful; then the query is hard to understand. When you want to read a query, you should start with the From clause where you will find the aliases. The From clause drives the query. (The Select query really should have the From clause first- but it is too late to change that.)

If you want to type out the full table name for each column reference that is OK with me.

If you want to use table aliases that is OK with me.

If you want to qualify only the columns that would be ambiguous that is OK with me.

Table of Contents

1.	What is an outer join?.....	1
2.	Tables for this demo.....	1
3.	Join Types.....	2
3.1.	INNER JOIN	2
3.2.	LEFT JOIN	2
3.3.	RIGHT JOIN	3
3.4.	FULL JOIN.....	3
4.	Unmatched joins	3
5.	Joins and the vets database	4

1. What is an outer join?

In a previous unit, we discussed inner joins. We use joins to bring data together from two or more tables. With an inner join between two tables, we see data from both tables only when there are matching rows in the two tables. the matching is often on the primary key of one table and the foreign key of the other. For example, we might want a list of clients and their order information, showing **only clients with orders**- we join on the cl_id which occurs in both tables..

Sometimes we want to see all of the data from one of the tables even if there is no matching data in the other table. For example, we might want a list of clients and their order information, including clients with orders and clients without orders. That requires an outer join.

When we discuss outer joins, we may use the term "preserved" table. That is the table which has all of its rows returned even if there are no matching rows in the other table.

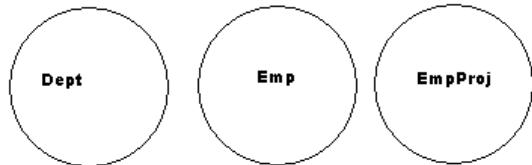
2. Tables for this demo

For this discussion I wanted a small set of tables. These are the tables we used last time for the Cartesian product demo-the cross join. Note that we have employees with no projects and a department with no employees and employees with no department. Although these tables did not have a primary key defined, there is only one row with the same d_id in the z_em_dept table, only one row with the same e_id in the z_em_emp table, and only one row with the same (p_id,e_id) combination in the z_em_empproj table. This is a simplification but most business tables should have a pk. (You can experiment with this by adding another row with d_id = 100 in the z_em_dept table.)

z_em_dept		z_em_emp			z_em_empproj	
D_ID	D_Name	E_ID	E_Name	D_ID	P_ID	E_ID
100	Manufacturing	1	Jones	150	ORDB-10	3
150	Accounting	2	Martin	150	ORDB-10	5
200	Marketing	3	Gates	250	Q4-SALES	2
250	Research	4	Anders	100	Q4-SALES	4
		5	Bossy	Null	ORDB-10	2
		6	Perkins	Null	Q4-SALES	5

In this discussion, I want to talk about the joins as concepts and results; we will get to the syntax in the next document.

Sometimes graphical representations help. Each of the tables is represented by a circle. (These are not actually Venn diagrams and this does not exactly follow set theory. Use the diagrams if they help.)



3. Join Types

We will mostly look at just the z_em_dept table and the z_em_emp table.

3.1. INNER JOIN

An inner join between these two tables using the d_id attribute will result in rows for employees who have a department and departments who have an employee. This includes the rows shown here.

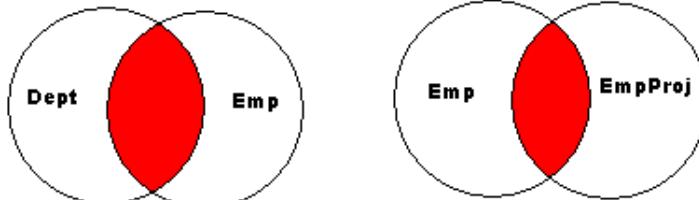
fromDept	D_Name	E_ID	E_Name	fromEmp
100	Manufacturing	4	Anders	100
150	Accounting	1	Jones	150
150	Accounting	2	Martin	150
250	Research	3	Gates	250

With an inner join you have the same result if you do the join in either order.

```
z_em_Dept    INNER JOIN z_em_Emp
z_em_Emp    INNER JOIN z_em_Dept
```

The red section of the diagrams represents the data returned by inner joins. From left to right:

Departments with employees
Employees assigned to projects

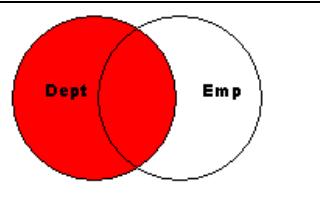


3.2. LEFT JOIN

This is a left join: z_em_Dept LEFT JOIN z_em_Emp

The result is all departments and any employees matched. We get one row each for the Manufacturing department and the Research department which each have one employee and two rows for Accounting which has two employees. The Marketing department has no employees and gets one row filled with nulls for the missing employee data.

fromDept	D_Name	E_ID	E_Name	fromEmp
100	Manufacturing	4	Anders	100
150	Accounting	1	Jones	150
150	Accounting	2	Martin	150
200	Marketing	NULL	NULL	NULL
250	Research	3	Gates	250

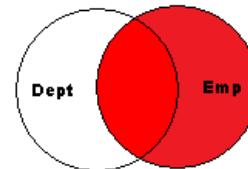


With an outer join, you get a different result if you do the join with the tables listed in the other order.

This is a left join: z_em_Emp LEFT JOIN z_em_Dept

The result is all employees and any departments matched. We have two employees without a department. (5, Bossy, null) and (6, Perkins, null). The null in the last column for these rows is the null from the table. The nulls in the first two columns for these rows are the nulls filled in by the outer join.

fromDept D_Name		E_ID	E_Name	fromEmp
NULL	NULL	5	Bossy	NULL
NULL	NULL	6	Perkins	NULL
100	Manufacturing	4	Anders	100
150	Accounting	1	Jones	150
150	Accounting	2	Martin	150
250	Research	3	Gates	250



3.3. RIGHT JOIN

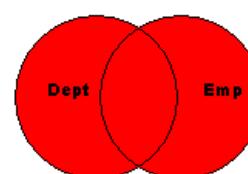
This is a right join: Dept RIGHT JOIN Employees. Note that the result set is the same as with : z_em_Emp LEFT JOIN z_em_Dept. You can use either syntax but you may find it easier at first to stick with either the Left join or the Right join.

fromDept D_Name		E_ID	E_Name	fromEmp
NULL	NULL	6	Perkins	NULL
NULL	NULL	5	Bossy	NULL
100	Manufacturing	4	Anders	100
150	Accounting	1	Jones	150
150	Accounting	2	Martin	150
250	Research	3	Gates	250

3.4. FULL JOIN

Another join is a full join which returns all departments and the matching employees and all employees and the matching departments. (Note we almost never have a task in the assignments where a Full join is the way to solve the task. It is included here for completeness.)

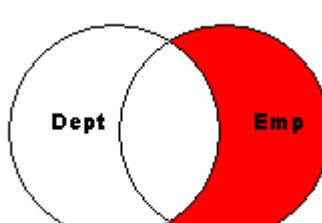
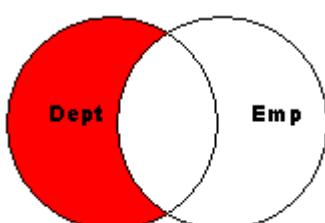
D_ID	D_Name	E_ID	E_Name
100	Manufacturing	4	Anders
150	Accounting	1	Jones
150	Accounting	2	Martin
200	Marketing	NULL	NULL
250	Research	3	Gates
NULL	NULL	5	Bossy
NULL	NULL	6	Perkins



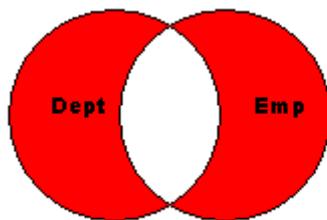
4. Unmatched joins

This is a common usage for the outer joins. It is not a different type of join but a way to use outer joins.

The first diagram represents departments without employees; the second diagram represents employees without departments.



The next represents departments with no employees and employees with no departments. This is not very common but the diagram is interesting.



5. Joins and the vets database

If you looked carefully at the create table statements for the tables z_em_dept, z_em_emp tabl, and z_emp_empproj, you might have noticed that there were no foreign keys created. We can still write joins because we have data values in common between the tables.

But most collections of tables are created with referential integrity and foreign keys - generally the foreign keys are set as Not Null. These settings help enforce business rules. The tables in the vets database do have referential integrity built into the table definitions. Let's consider the joins in the vets set of tables looking at just the tables for client and animals.

The clients table has a PK of cl_id. The animals table has a pk of an_id and a not null FK of cl_id which is tied to the clients table. That means that we cannot enter a row into the animals table unless that animal is associated with a client we already have in the client tables. (That is how the vet knows who is going to pay the bills- a good business rule.)

An inner join between clients and animals returns rows for clients who have animals. If a client does not have any animals currently(that means there is no row in the animals table with that cl_id), that client is not returned. It is also true that with the inner join an animal also has to have a client-but that rule is already built into the animals table.

If we do an outer join using clients left join animals, then the table expression includes clients with animals and clients without animals. The table expression includes a row for each valid combination of client and animal- if a client has 4 animals, that client gets 4 rows in the table expression. And the table expression includes one row for each client who has no animals. (The client is the preserved table.) One way to think of an outer join from the clients table to the animals tables is as a two step process- these two steps are done for us when you run the query.

First we get the clients with animals. Then the dbms will add in rows for the clients with no animals- clients where there is no matching row in the animals table. What should we display for the an_type column if the client has no animals? We might want to display a message- but what the dbms does for those rows is return a null in that attribute. The null is what SQL commonly uses for a situation where there is no data.

Now consider the opposite join- animals left join clients. The table expression includes animals with clients and animals without a client. We have animals with clients, but the rules for the animal table says that we do not have any animals without a client. So in this case the table expression is just the same as an inner join- clients with animals. You should not write the join animals left join clients, since an outer join may be less efficient than an inner join and with our tables the result will always be the inner join.

Sometimes people write outer join and then include a Where clause filter that eliminates all of the null-added rows, ending up with an inner join. That is also not efficient and needs to be avoided.

Use outer joins when you need them, but only when you need them.

Table of Contents

1. Syntax for outer joins (Left, Right, Full).....	1
2. Queries using the altgeld_mart tables	3
2.1. Customers and orders	3
2.2. Products and orders.....	5

These are the tables we are using. Note that we have employees with no projects and a department with no employees and employees with no department.

z_em_dept	
D_ID	D_Name
100	Manufacturing
150	Accounting
200	Marketing
250	Research

z_em_emp		
E_ID	E_Name	D_ID
1	Jones	150
2	Martin	150
3	Gates	250
4	Anders	100
5	Bossy	NULL
6	Perkins	NULL

z_em_empproj	
P_ID	E_ID
ORDB-10	3
ORDB-10	5
Q4-SALES	2
Q4-SALES	4
ORDB-10	2
Q4-SALES	5

1. Syntax for outer joins (Left, Right, Full)

Outer joins can use the syntax Left Join or Right Join. A left outer join written as

From tblA LEFT JOIN tblB

will include all rows from table tblA and any matching rows from tblB. The table to the left of the phrase Left Join will have all of its rows returned.

A right outer join written as

From tblA RIGHT JOIN tblB

will include all rows from table tblB and any matching rows from tblA. The table to the right of the phrase Right Join will have all of its rows returned.

The outer joins are not symmetric.

The word OUTER is optional; you can use Left Outer Join or Left Join.

You will still need to identify the joining columns and code the join phrase.

Demo 01: All departments; employees of those departments if they exist.

```
select d_id, d_name, e_id, e_name
from z_em_dept
LEFT JOIN z_em_emp using(d_id)
order by e_id;
```

D_ID	D_NAME	E_ID	E_NAME
150	Accounting	1	Jones
150	Accounting	2	Martin
250	Research	3	Gates
100	Manufacturing	4	Anders
200	Marketing		

Demo 02: All employees; assigned departments if they exist. Outer joins are not commutative

```
select d_id, d_name, e_id, e_name
from z_em_emp
LEFT JOIN z_em_dept using(d_id)
order by e_id;
```

D_ID	D_NAME	E_ID	E_NAME
150	Accounting	1	Jones
150	Accounting	2	Martin
250	Research	3	Gates
100	Manufacturing	4	Anders
		5	Bossy
		6	Perkins

Demo 03: All employees; assigned departments if they exist.

```
select d_id, d_name, e_id, e_name
from z_em_dept
RIGHT JOIN z_em_emp using(d_id)
order by e_id;
```

D_ID	D_NAME	E_ID	E_NAME
150	Accounting	1	Jones
150	Accounting	2	Martin
250	Research	3	Gates
100	Manufacturing	4	Anders
		5	Bossy
		6	Perkins

Oracle supports the Full Outer joins which includes all rows from each table and matches where it can.

Demo 04: all employees and all departments matching employees to their departments.

```
select d_id, d_name, e_id, e_name
from z_em_dept
FULL OUTER JOIN z_em_emp using(d_id)
order by e_id;
```

D_ID	D_NAME	E_ID	E_NAME
150	Accounting	1	Jones
150	Accounting	2	Martin
250	Research	3	Gates
100	Manufacturing	4	Anders
		5	Bossy
		6	Perkins
	Marketing		

Demo 05: Three table outer join. This is all of the departments and their employees if there are any in the department and the projects if the employees have a project.

```
select d_id, d_name, e_id, e_name, p_id
from z_em_dept
LEFT JOIN z_em_emp using(d_id)
LEFT JOIN z_em_empproj using(e_id)
order by e_id;
```

D_ID	D_NAME	E_ID	E_NAME	P_ID
150	Accounting	1	Jones	
150	Accounting	2	Martin	ORDB-10

150 Accounting	2 Martin	Q4-SALES
250 Research	3 Gates	ORDB-10
100 Manufacturing	4 Anders	Q4-SALES
200 Marketing		

Demo 06: Three table outer join. This is all of the employees and their departments if they have one and their projects if they have one

```
select e_id, e_name, d_id, d_name, p_id
from z_em_emp
LEFT JOIN z_em_dept using(d_id)
LEFT JOIN z_em_empproj using(e_id)
order by e_id;
```

E_ID	E_NAME	D_ID	D_NAME	P_ID
1	Jones	150	Accounting	
2	Martin	150	Accounting	Q4-SALES
2	Martin	150	Accounting	ORDB-10
3	Gates	250	Research	ORDB-10
4	Anders	100	Manufacturing	Q4-SALES
5	Bossy			ORDB-10
5	Bossy			Q4-SALES
6	Perkins			

Demo 07: Suppose we want to see all employees and their departments if they have one and the names of their projects if they have one. The following query does not do that. We start with an outer join but then use an inner join which eliminates employees with no projects.

```
select z_em_emp.e_id, e_name, z_em_dept.d_id, d_name, p_id
from z_em_emp
LEFT JOIN z_em_dept on z_em_dept.d_id = z_em_emp.d_id
join z_em_empproj on z_em_emp.e_id = z_em_empproj.e_id
order by z_em_emp.e_id;
```

E_ID	E_NAME	D_ID	D_NAME	P_ID
2	Martin	150	Accounting	Q4-SALES
2	Martin	150	Accounting	ORDB-10
3	Gates	250	Research	ORDB-10
4	Anders	100	Manufacturing	Q4-SALES
5	Bossy			ORDB-10
5	Bossy			Q4-SALES

2. Queries using the altgeld_mart tables

2.1. Customers and orders

Demo 08: Customers with orders. This uses an inner join. The cust_id filter is simply to reduce the volume of output.

```
select customer_id
, customer_name_last
, order_id
from cust_customers
join oe_orderHeaders using(customer_id)
where customer_id between 404900 and 409030
order by customer_id, order_id;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	ORDER_ID
404900	Williams	520
404950	Morris	110
404950	Morris	408
404950	Morris	411
404950	Morris	535
404950	Morris	540
404950	Morris	4510
405000	Day	116
408770	Clay	405
409030	Mazur	128
409030	Mazur	130
409030	Mazur	324

Demo 09: Customers with and without orders. This uses an outer join; Customers Left Join Order Headers.

That means we get customers with orders and if the customer has several orders, that customer gets multiple lines in the result set.

We also get rows for the customers in this cust_id range who have no orders and the column for their order id value is null- these customers each get one row.

```
select customer_id
, customer_name_last
, order_id
from cust_customers
LEFT JOIN oe_orderHeaders using(customer_id)
where customer_id between 404900 and 409030
order by customer_id, order_id
;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	ORDER_ID
404900	Williams	520
404950	Morris	110
404950	Morris	408
404950	Morris	411
404950	Morris	535
404950	Morris	540
404950	Morris	4510
405000	Day	116
408770	Clay	405
408777	Morise	
409010	Morris	
409020	Max	
409030	Mazur	128
409030	Mazur	130
409030	Mazur	324

Demo 10: Now consider this join. I change the join to a right join. The result set is the same as the inner join used previously. Why?

```
select customer_id
, customer_name_last
, order_id
from cust_customers
RIGHT JOIN oe_orderHeaders using(customer_id)
where customer_id between 404900 and 409030
order by customer_id, order_id
;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	ORDER_ID
404900	Williams	520
404950	Morris	110
404950	Morris	408
404950	Morris	411
404950	Morris	535
404950	Morris	540
404950	Morris	4510
405000	Day	116
408770	Clay	405
409030	Mazur	128
409030	Mazur	130
409030	Mazur	324

12 rows selected.

In our database we have a foreign key in the order headers table that refers back to the customer table and (by default) to the customer_id in the customer table.

```
create table oe_orderHeaders(
    order_id      integer not null
    , order_date   date    not null
    , cust_id      integer not null
    .
    ,
    constraint ord_cust_fk  foreign key (customer_id) references cust_customers
    . . . )
```

I also set the customer_id in the order headers table as Not null. This means that every row in the order headers table must have a value for the customer_id (it is Not null) and that customer_id in the order header must match a customer_id in the customers tables (foreign key reference).

The outer join in this query is asking for all orders whether or not they match a customer. But our database is set up so that every order header row is matched with a customer. So it does not make sense to ask to see order headers rows that do not match a customer. In this case you should use an inner join. Using an outer join when it is logically impossible to return unmatched rows is inefficient. Someone reading your query would assume you have made a mistake someplace but they would not know what the mistake is- is the database badly designed and allows the entry of orders that do not belong to a customer (who pays for those orders?), or did you get the join order incorrect?

2.2. Products and orders

These are limited to products in the MUS category to reduce the volume of output

Demo 11: First an inner join- these show products which have been ordered- each product id must match a product id on an order detail row

```
select pr.prod_id, pr.prod_desc, pr.catg_id, od.order_id
from prd_products pr
join oe_orderDetails od  on pr.prod_id = od.prod_id
where pr.catg_id in ('MUS')
order by pr.prod_id;
```

PROD_ID	PROD_DESC	CATG_ID	ORDER_ID
2014	Bix Beiderbecke - Tiger Rag	MUS	413
2014	Bix Beiderbecke - Tiger Rag	MUS	552
2014	Bix Beiderbecke - Tiger Rag	MUS	525
2014	Bix Beiderbecke - Tiger Rag	MUS	2218
2014	Bix Beiderbecke - Tiger Rag	MUS	715
2014	Bix Beiderbecke - Tiger Rag	MUS	3518
2412	David Newman - Davey Blue	MUS	525

2412 David Newman - Davey Blue	MUS	2225
2746 Charles Mingus - Blues & Politics	MUS	2218
2746 Charles Mingus - Blues & Politics	MUS	525
2747 Charles Mingus - Blues & Roots	MUS	520
2947 Ornette Coleman - Sound Grammar	MUS	525
2947 Ornette Coleman - Sound Grammar	MUS	2225
2984 John Coltrane - Lush Life	MUS	715
2984 John Coltrane - Lush Life	MUS	552
2984 John Coltrane - Lush Life	MUS	413
2984 John Coltrane - Lush Life	MUS	3518

17 rows selected.

Demo 12: How many products do we have in the MUS category?

We have 11 products; looking at the previous result set, 6 of these products were sold (Several were sold on more than one order.)

```
select pr.prod_id, pr.prod_desc, pr.catg_id
from prd_products pr
where catg_id in ('MUS')
order by pr.prod_id;
```

PROD_ID	PROD_DESC	CATG_ID
2014 Bix Beiderbecke - Tiger Rag	MUS	
2234 Charles Mingus - Pithecanthropus Erectus	MUS	
2337 John Coltrane - Blue Train	MUS	
2412 David Newman - Davey Blue	MUS	
2487 Stanley Turrentine - Don't Mess With Mr. T	MUS	
2746 Charles Mingus - Blues & Politics	MUS	
2747 Charles Mingus - Blues & Roots	MUS	
2933 David Newman - I Remember Brother Ray	MUS	
2947 Ornette Coleman - Sound Grammar	MUS	
2984 John Coltrane - Lush Life	MUS	
2987 Stanley Turrentine - Ballads	MUS	

11 rows selected.

Demo 13: We can use an outer join to get both ordered and un-ordered products. I have highlighted the rows where the order id is null; those are the products that were never sold.

```
select pr.prod_id, prod_desc, catg_id, order_id
from prd_products pr
LEFT JOIN oe_orderDetails od on pr.prod_id = od.prod_id
where catg_id in ('MUS')
order by pr.prod_id;
```

PROD_ID	PROD_DESC	CATG_ID	ORDER_ID
2014 Bix Beiderbecke - Tiger Rag	MUS	413	
2014 Bix Beiderbecke - Tiger Rag	MUS	552	
2014 Bix Beiderbecke - Tiger Rag	MUS	525	
2014 Bix Beiderbecke - Tiger Rag	MUS	2218	
2014 Bix Beiderbecke - Tiger Rag	MUS	715	
2014 Bix Beiderbecke - Tiger Rag	MUS	3518	
2234 Charles Mingus - Pithecanthropus Erectus	MUS		
2337 John Coltrane - Blue Train	MUS		
2412 David Newman - Davey Blue	MUS	525	
2412 David Newman - Davey Blue	MUS	2225	
2487 Stanley Turrentine - Don't Mess With Mr. T	MUS		
2746 Charles Mingus - Blues & Politics	MUS	2218	
2746 Charles Mingus - Blues & Politics	MUS	525	
2747 Charles Mingus - Blues & Roots	MUS	520	

2933 David Newman - I Remember Brother Ray	MUS	
2947 Ornette Coleman - Sound Grammar	MUS	525
2947 Ornette Coleman - Sound Grammar	MUS	2225
2984 John Coltrane - Lush Life	MUS	715
2984 John Coltrane - Lush Life	MUS	552
2984 John Coltrane - Lush Life	MUS	413
2984 John Coltrane - Lush Life	MUS	3518
2987 Stanley Turrentine - Ballads	MUS	
22 rows selected.		

Demo 14: This query gives us rows for the same products- why are we missing values in the first column which shows the product id? Every product has a product Id!

```
select od.prod_id, prod_desc, catg_id, order_id
from prd_products pr
LEFT JOIN oe_orderDetails od on pr.prod_id = od.prod_id
where catg_id in ('MUS')
order by od.prod_id;
```

PROD_ID	PROD_DESC	CATG_ID	ORDER_ID
2014 Bix Beiderbecke - Tiger Rag		MUS	3518
2014 Bix Beiderbecke - Tiger Rag		MUS	413
2014 Bix Beiderbecke - Tiger Rag		MUS	715
2014 Bix Beiderbecke - Tiger Rag		MUS	2218
2014 Bix Beiderbecke - Tiger Rag		MUS	525
2014 Bix Beiderbecke - Tiger Rag		MUS	552
2412 David Newman - Davey Blue		MUS	525
2412 David Newman - Davey Blue		MUS	2225
2746 Charles Mingus - Blues & Politics		MUS	525
2746 Charles Mingus - Blues & Politics		MUS	2218
2747 Charles Mingus - Blues & Roots		MUS	520
2947 Ornette Coleman - Sound Grammar		MUS	525
2947 Ornette Coleman - Sound Grammar		MUS	2225
2984 John Coltrane - Lush Life		MUS	3518
2984 John Coltrane - Lush Life		MUS	413
2984 John Coltrane - Lush Life		MUS	552
2984 John Coltrane - Lush Life		MUS	715
Stanley Turrentine - Ballads		MUS	
Charles Mingus - Pithecanthropus Erectus		MUS	
John Coltrane - Blue Train		MUS	
David Newman - I Remember Brother Ray		MUS	
Stanley Turrentine - Don't Mess With Mr. T		MUS	
22 rows selected.			

What I did is switch the column alias for the first column and for the sort key to use the order details table. If I am looking for the product id in the order details table, the products which are not ordered do not have a value for that column and display as nulls.

Table of Contents

1. Unmatched queries using outer join.....	1
2. Queries using the AltgeldMart tables	2
3. Unmatched queries using subqueries.....	3
4. What can go wrong?	4

Inner joins are great for finding Customers with Orders and for finding Products that have been ordered. But we often want to find customers who have no orders or products that no one has ordered. These are sometimes called unmatched queries since we are looking for customers in the customer table who have no matching rows in the order headers table. We will see several ways to do this. For now we will look at two approaches: (1) using the outer join and (2) using subqueries.

1. Unmatched queries using outer join

In the previous document we used the outer join to find employees **with and without** an assigned department. A variation on the outer join is a query to display only those employees who have no assigned department. Be careful to select the proper column for testing against null. With these tests you do not want to use the join with the Using (col) syntax because you have to specify the exact column you are looking for. Compare the following two queries. We want departments with no employees.

Demo 01: Unmatched rows. Departments which do not have any employees

```
select z_em_Emp.D_ID as "EM_Emp.D_ID"
, z_em_Dept.D_ID as "EM_Dept.D_ID"
, D_Name
from z_em_Dept
LEFT JOIN z_em_Emp ON z_em_Dept.d_id = z_em_Emp.D_ID
where z_em_Emp.D_ID IS NULL;
```

EM_Emp.D_ID	EM_Dept.D_ID	D_NAME
200		MARKETING

Demo 02: Unmatched rows. This syntax will not work since the USING clause means that we cannot qualify D_ID to specify the table name. Note that this does not present as an error; we simply get no rows returned.

```
select D_ID as "EM_Emp.D_ID"
, D_ID as "EM_Dept.D_ID"
, D_Name
from z_em_Dept
LEFT JOIN z_em_Emp Using(D_ID)
where D_ID IS NULL;
```

no rows selected

Demo 03: Unmatched rows. Take care which attribute you test. Since we are retrieving all data from the department table, we will not have nulls in the department table id attribute.

```
select z_em_Emp.D_ID as "EM_Emp.D_ID"
, z_em_Dept.D_ID as "EM_Dept.D_ID"
, D_Name
from z_em_Dept
LEFT JOIN z_em_Emp ON z_em_Dept.d_id = z_em_Emp.D_ID
where z_em_Dept.D_ID IS NULL
;
no rows selected
```

Demo 04: If you have troubles with setting up this type of query, run the query without the null filter first and examine the columns for the rows you want to return. Here we can see that we want to test the z_em_Emp.d_id column for nulls.

```
select z_em_Emp.D_ID as "EM_Emp.D_ID"
, z_em_Dept.D_ID as "EM_Dept.D_ID"
, D_Name
from z_em_Dept
LEFT JOIN z_em_Emp ON z_em_Dept.d_id = z_em_Emp.D_ID
;
```

EM_Emp.D_ID	EM_Dept.D_ID	D_NAME
150	150	Accounting
150	150	Accounting
250	250	Research
100	100	Manufacturing
200	200	Marketing

2. Queries using the AltgeldMart tables

Demo 05: Customers without orders. We have an outer join from customers left join order headers to get customers with and without orders and then we filter for just rows where the order id in the order headers table is null.

```
select CS.customer_id
, CS.customer_name_last
, OH.order_id
from cust_customers CS
left join oe_orderHeaders OH on CS.customer_id = OH.customer_id
where OH.order_id IS NULL
order by CS.customer_id;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	ORDER_ID
400801	Washington	
402110	Coltrane	
402120	McCoy	
402500	Jones	
403500	Stevenson	
403750	O'Leary	
403760	O'Leary	
404150	Dancer	
404180	Shay	
. . .	rows omitted	

Demo 06: If we try to find orders with no customers, we have no rows returned. Our database is set up to reject any order that is not associated with a customer. This would be a good query to run on poorly designed databases to locate orphaned rows.

```
select
CS.customer_id
, CS.customer_name_last
, OH.order_id
from oe_orderHeaders OH
Left Join cust_customers CS On CS.customer_id = OH.customer_id
where OH.customer_id Is Null;
```

no rows selected

Demo 07: What is the product name and list price for the products that are not selling? These would be products in the products table that do not appear on any order.

```
select
    catg_id
    , prod_id
    , prod_desc As product
    , prod_list_price
from prd_products
Left Join oe_orderDetails Using (prod_id)
where order_id Is Null
order by catg_id, prod_id;
```

CATG_ID	PROD_ID	PRODUCT	PROD_LIST_PRICE
APL	1126	Low Energy Washer Dryer combo	850
APL	4569	Sized for the apartment	349.95
GFD	5000	Cello bag of mixed fingerling potatoes	12.5
GFD	5001	Dundee Ginger Preserve 12 oz jar	5
HW	1160	Stand Mixer with attachments	149.99
HW	4575	GE model 34PG98	49.95
MUS	2234	Charles Mingus - Pithecanthropus Erectus	15.88
MUS	2337	John Coltrane - Blue Train	15.87
MUS	2487	Stanley Turrentine - Don't Mess With Mr. T	9.45
MUS	2933	David Newman - I Remember Brother Ray	12.45
MUS	2987	Stanley Turrentine - Ballads	15.87
PET	1142	Bird seed mix with sunflowers	2.5
PET	1143	Bird seed mix with more sunflower seeds	2.5
PET	4567	Our highest end cat tree- you gotta see this	549.99
PET	4568	Satin four-poster cat bed	549.99

3. Unmatched queries using subqueries

The unmatched pattern we have used gets customer with and without orders and then filters away the customers with orders.

Some people find the subquery syntax easier to understand. The subquery approach says to find rows where we have a value in one table and we do not have that value in another table. For example, customers in the customer table but that customer is not in the order-headers table.

Demo 08: These are customers without orders query done using a subquery.

The subquery gets the cust_id for customers with orders and then the main query filters those out using the NOT IN test. That gives us the customers with no orders.

```
select
    customer_id
    , customer_name_last
from cust_customers
where customer_id Not In (
    select customer_id
    from oe_orderHeaders
);
```

CUSTOMER_ID	CUSTOMER_NAME_LAST
400801	Washington
402110	Coltrane
402120	McCoy
402500	Jones

```

403500 Stevenson
403750 O'Leary
403760 O'Leary
404150 Dancer
404180 Shay
. . . rows omitted

```

Notice that we are comparing the customer_id in the customers table with a list of customer_id values from the order_headers table. We compare customer_id to customer_id. This is similar to the join logic.

Demo 09: What is the product name and list price for the products that are not selling? These would be products in the products table that do not appear on any order.

This query does not need table aliases for prod_id since each part of the query is referencing a single table.

```

select catg_id
, prod_id
, prod_desc as product
, prod_list_price
from prd_products
where prod_id NOT IN (
    select prod_id
    from oe_order_details)
order by catg_id, prod_id ;

```

CATG_ID	PROD_ID	PRODUCT	PROD_LIST_PRICE
APL	1126	Low Energy washer Dryer combo	850
APL	4569	Sized for the apartment	349.95
GFD	5000	Cello bag of mixed fingerling potatoes	12.5
GFD	5001	Dundee Ginger Preserve 12 oz jar	5
HW	1160	Stand Mixer with attachments	149.99
HW	4575	GE model 34PG98	49.95
MUS	2234	Charles Mingus - Pithecanthropus Erectus	15.88
MUS	2337	John Coltrane - Blue Train	15.87
MUS	2487	Stanley Turrentine - Don't Mess With Mr. T	9.45
MUS	2933	David Newman - I Remember Brother Ray	12.45
MUS	2987	Stanley Turrentine - Ballads	15.87
PET	1142	Bird seed mix with sunflowers	2.5
PET	1143	Bird seed mix with more sunflower seeds	2.5
PET	4567	Our highest end cat tree- you gotta see this	549.99
PET	4568	Satin four-poster cat bed	549.99

4. What can go wrong?

Suppose we want to find employees who are not associated with any orders. Remember that in the order headers table, the employee who took the order is referred to as the sales_rep. First do a left join to see what the data looks like.

Demo 10: Left join Employees to Orders

```

select
    emp_id
, name_last
, order_id
from emp_employees
Left Join oe_orderHeaders On emp_id = sales_rep_id
;

```

EMP_ID	NAME_LAST	ORDER_ID
100	King	
101	Koch	
102	D'Haa	
103	Hunol	
104	Ernst	
108	Green	
109	Fiet	
110	Chen	
145	Russ	112
145	Russ	540
145	Russ	2505
145	Russ	405
145	Russ	130
145	Russ	312
. . . rows omitted		

Some employees have served as a sales rep and some have taken more than one order. We could add a filter to find the rows where the Order_id is null.

Demo 11: Left join Employees to Orders with null order id. These are employees who are not associated with any order.

```
select emp_id, name_last
from emp_employees
Left Join oe_orderHeaders On emp_id = sales_rep_id
where order_id Is Null;
```

EMP_ID	NAME_LAST
100	King
101	Koch
102	D'Haa
103	Hunol
104	Ernst
108	Green
109	Fiet
110	Chen
146	Partne
. . . rows omitted	
18 rows selected.	

What if we try this with a subquery?

Demo 12: Subquery version 1 We filter for employee id values that are not in the appropriate column (sales_rep_id) in the order headers table. This returns no rows at all! Before you read on to the next demo, try to figure out why this might happen. (What is the usual villain when a query goes bad?)

```
select emp_id, name_last
from emp_employees
where emp_id Not In (
    select sales_rep_id
    from oe_orderHeaders
);
no rows selected
```

Remember that a Not In () predicate returns no rows if there is a null in the list.

Demo 13: Subquery version 2-Eliminate the nulls from the subquery.

```
select
    emp_id
,   name_last
from emp_employees
where emp_id Not In (
    select sales_rep_id
    from oe_orderHeaders
    where sales_rep_id Is Not Null
);
Same result set as with the outer join
```

So now the question is: why did the other subqueries work? We were filtering on an attribute that was a not null attribute.

Table of Contents

1. Associating tables on conditions other than equality.....	1
2. Self-Joins.....	1
3. Legacy comma inner join	4

Many of our joins are joining tables by matching the fk and pk of two tables and doing an equality match. There are a few more join conditions that may be useful.

1. Associating tables on conditions other than equality

Demo 01: This uses a join that involves two attributes to check if any items were sold at more than their list price. This does not use a Where clause- the testing is done in the join.

```
select
    PR.prod_id
    , quoted_price
    , prod_list_price
    , order_id
from oe_orderDetails OD
join prd_products PR On OD.prod_id = PR.prod_id
and quoted_price > prod_list_price;
```

PROD_ID	QUOTED_PRICE	PROD_LIST_PRICE	ORDER_ID
1010	175	150	390
1010	195	150	395
1010	175	150	550
1010	175	150	551
1010	175	150	609
1010	175	150	2120
1010	175	150	2121
1100	205	49.99	301
1150	7.25	4.99	223
1152	55.25	55	540
1152	55.25	55	2508

2. Self-Joins

You can join a table to itself. You need to use a table alias to distinguish the two copies of the table involved in the join. The following is the traditional self-join of employees and their managers

Demo 02: Employees and managers . Note that the first row here has no Manager. Employee 100 is at the top of the chart.

```
select
    M.emp_id || ' ' || M.name_last As "Manager"
    , E.emp_id || ' ' || E.name_last As "Supervises"
from emp_employees E
Left Join emp_employees M On m.emp_id = e.emp_mng
order by "Manager", "Supervises";
```

Manager	Supervises
	100 King
100 King	101 Koch
100 King	102 D'Haa
100 King	145 Russ
100 King	146 Partne

100 King	201 Harts
101 Koch	108 Green
101 Koch	162 Holme
101 Koch	200 Whale
101 Koch	203 Mays
101 Koch	205 Higgs
102 D'Haa	103 Hunol
103 Hunol	104 Ernst
. . . rows omitted	

This is another self-join. The following query returns pairs of employees who have the same job id. We are joining on the job id and also on an inequality between the employees' ids. If we do not add that second joining condition then each employee would be paired with themselves (since the job id values would match). The output shows one row if there are two employees with the same job id; and three rows if there are three employees with the same job id due to the pair matching.

Demo 03: Pairing Employees who have the same job id

```
select emp_1.job_id
, emp_1.emp_id, emp_2.emp_id
from emp_employees emp_1
join emp_employees emp_2
  on emp_1.job_id = emp_2.job_id
  and emp_1.emp_id < emp_2.emp_id
order by emp_1.job_id, emp_1.emp_id, emp_2.emp_id;
```

JOB_ID	EMP_ID	EMP_ID
8	150	155
8	150	207
8	155	207
16	101	108
16	101	161
16	101	162
16	101	200
16	101	203
16	101	205
16	108	161
16	108	162
16	108	200
16	108	203
16	108	205
16	161	162
16	161	200
16	161	203
16	161	205
16	162	200
16	162	203
16	162	205
16	200	203
16	200	205
16	203	205
32	104	109
32	104	110
32	104	160
32	104	204
32	104	206
32	109	110
32	109	160
32	109	204
32	109	206
32	110	160

32	110	204
32	110	206
32	160	204
32	160	206
32	204	206
64	102	103
64	102	146
64	103	146

42 rows selected

Demo 04: Finding employees who earn more than other employees. This has a lot of rows of output

```
select
    E1.emp_id, E1.salary , ' earns more than ' as " "
    , E2.emp_id ,E2.salary
from emp_employees E1 ,
     emp_employees E2
where E1.salary > E2.salary
order by E1.salary desc, E2.salary desc;
```

The output starts with employee 161 who has the highest salary and is matched with all other employees. The next set of rows starts with employee 100 who has the next highest salary. The last set of rows starts with employee 150 who earns more than only the employee(s) with the lowest salary- in our data set that is employee 201. Note there is no set of rows that start with this employee id.

EMP_ID	SALARY		EMP_ID	SALARY
161	120000	earns more than	100	100000
161	120000	earns more than	204	99090
161	120000	earns more than	101	98005
161	120000	earns more than	162	98000
161	120000	earns more than	206	88954
161	120000	earns more than	146	88954
161	120000	earns more than	205	75000
161	120000	earns more than	103	69000
161	120000	earns more than	160	65000
161	120000	earns more than	104	65000
161	120000	earns more than	109	65000
161	120000	earns more than	200	65000
161	120000	earns more than	203	64450
161	120000	earns more than	108	62000
161	120000	earns more than	102	60300
161	120000	earns more than	110	60300
161	120000	earns more than	145	59000
161	120000	earns more than	207	30000
161	120000	earns more than	155	29000
161	120000	earns more than	150	20000
161	120000	earns more than	201	15000
100	100000	earns more than	204	99090
100	100000	earns more than	101	98005
100	100000	earns more than	162	98000
100	100000	earns more than	206	88954
100	100000	earns more than	146	88954
100	100000	earns more than	205	75000
... rows omitted for many employees				
207	30000	earns more than	155	29000
207	30000	earns more than	150	20000
207	30000	earns more than	201	15000
155	29000	earns more than	150	20000
155	29000	earns more than	201	15000
150	20000	earns more than	201	15000

223 rows selected.

3. Legacy comma inner join

There is a traditional, legacy join that does the attribute matching in the Where clause. You will see this join in a lot of older code (and in a lot of code written now).

Logically this syntax does a Cartesian product and adds a filter for the records that match on the joining condition.

This join syntax is not allowed in this class for assignments. I want you to get used to using the more uniform join syntax using the Condition join.

Demo 05: A Column Name style inner join of orders and order details

```
select
    customer_id
, order_id
, prod_id
, quantity_ordered * quoted_price As "ExtCost"
from oe_orderHeaders
join oe_orderDetails Using (order_id)
order by customer_id, order_id;
```

CUSTOMER_ID	ORDER_ID	PROD_ID	ExtCost
400300	378	1120	2250
400300	378	1125	2250
401250	106	1060	255.95
401250	113	1080	22.5
401250	119	1070	225
401250	301	1100	205
. . . rows omitted			

Demo 06: Comma join: Using the join of orders and order details in the Where clause

```
select
    oe_orderHeaders.customer_id
, oe_orderHeaders.order_id
, oe_orderDetails.prod_id
, oe_orderDetails.quantity_ordered * oe_orderDetails.quoted_price As "ExtCost"
from oe_orderHeaders
, oe_orderDetails
where oe_orderHeaders.order_id = oe_orderDetails.order_id
order by oe_orderHeaders.customer_id, oe_orderHeaders.order_id;
```

The advantage of doing the join in the From clause is that it isolates the join issues from the Where clause filters. If you do the join in the Where clause then you need to take more care with other filters in the Where clause especially if you have both And and Or operators in the Where clause.

Oracle also supports an older outer join syntax in the Where clause which we do not discuss.

Table of Contents

1. Common table expression	1
2. Using multiple CTEs.....	3
2.1. CTE order of definition	4

A table expression determines a virtual table. We commonly see table expressions in the From clause of a query. We started with the simplest table expression- a single base table. (Remember a base table is a persistent table; we create the base table with a Create Table statement.)

We use Inner Joins to connect two or more base tables to create a virtual table. That join is a table expression.

```
select an_name, cl_name_last
from vt_animals an
join vt_clients cl on an.cl_id = cl.cl_id;
```

In this unit we added other table expressions using outer joins to create a virtual table. Those joins also define table expressions.

Now we are going to discuss a technique called a Common Table Expression which is available in Oracle and in SQL Server- but not yet in MySQL. This discussion is limited to the use of non-recursive CTEs.

1. Common table expression

Suppose you have a fairly complex query dealing with customer orders that you need to run only for a particular query. You would like to break the query down into smaller, more manageable chunks that you could test separately. One solution is to create a subquery that handles part of the query- perhaps the joining of the tables and give that subquery a name and then use that name in the From clause of the rest of the query

The Oracle Select statement has a clause called a With clause that lets us do that. First, a very simplistic example. If we want to see the ID and name of all of our customers with a first name of William, we would probably run the SQL statement shown here.

Demo 01: simple select

```
select
    customer_id
, customer_name_first || ' ' || customer_name_last As cust_name
from cust_customers
where customer_name_first = 'William';

CUSTOMER_ID CUST_NAME
-----
401890 William Northrep
402100 William Morise
404950 William Morris
409010 William Morris
409020 William Max
```

Demo 02: We could rewrite this using a CTE.

```
With custnames as (
    select customer_id
, customer_name_first || ' ' || customer_name_last as cust_name
from cust_customers
where customer_name_first = 'William'
)
select customer_id, cust_name
from custnames
;
```

We get the same result. At first it just looks like we made the select statement longer to no purpose. You should not use a CTE with a query this simple. But the demos always start simple.

What is happening is that the With clause defines a name (here **custnames**) and a subquery which can then be used by name in the From clause of the main Select. Oracle sometimes refers to this technique as a subquery-factoring clause; more often it is called a common table expression.

The select statement defined within the CTE is a subquery. It is enclosed in parentheses. It has a name.

The CTE does not exist after the query has finished executing.

Demo 03: We can use a CTE to encapsulate a Select and the column alias defined in the CTE can be used in the main query. We cannot define and use a column alias in the same Select/Where.

```
With ClNames as (
    select cl_state
    , cl_name_last || ' ' || cl_name_first as ClientName
    from vt_clients
)
select ClientName || ' lives in ' || cl_state
from ClNames ;
```

Demo 04: Using a more complex query. We can put the complexity of the join into the CTE and the main query body is much simpler.

Display the order date and line amount due for each detail line

```
With rpt_base as (
    select
        order_id
    , order_date
    , customer_id
    , quoted_price * quantity_ordered as itemTotal
    from oe_orderHeaders
    join oe_orderDetails using(order_id)
    join prd_products using(prod_id)
    where quoted_price > 0 and quantity_ordered > 0
)
select
    order_id
, order_date
, itemTotal
from rpt_base
where order_date < '01-AUG-2015'
order by order_date;
```

ORDER_ID	ORDER_DATE	ITEMTOTAL
522	05-APR-15	45
540	02-JUN-15	49.99
540	02-JUN-15	45
540	02-JUN-15	55.25
307	04-JUN-15	2250
307	04-JUN-15	2250
306	04-JUN-15	500
306	04-JUN-15	500
302	04-JUN-15	349.95
...	rows omitted	

Some people prefer this style of working out a query; they encapsulate part of the work in the CTE and then have a simpler query in the main Select.

2. Using multiple CTEs

You can have only one With clause in an SQL statement; but you can define more than one subquery. Give each one a name and separate them with commas.

Demo 05: Using a With clause to define two subqueries with calculated columns and column aliases

```
with t_cust as
  (select customer_id
   , customer_name_first || ' ' || customer_name_last as cust_name
   from cust_customers
   where customer_name_first = 'William'
  )
  , t_ord as
    (select order_id
     , order_date
     , customer_id
     , prod_id
     , quoted_price * quantity_ordered as ext_price
     from oe_orderHeaders
     join oe_orderDetails using (order_id) )
select
  customer_id
, cust_name
, prod_id
, ext_price
from t_cust
join t_ord Using (customer_id)
order By customer_id, prod_id;
```

CUSTOMER_ID	CUST_NAME	PROD_ID	EXT_PRICE
401890	William Northrep	1020	64.75
401890	William Northrep	1110	49.99
401890	William Northrep	1110	99.98
402100	William Morise	1000	200
402100	William Morise	1030	27
402100	William Morise	1080	25
402100	William Morise	1100	180
402100	William Morise	1120	1900
402100	William Morise	1130	625
. . . rows omitted.			

In this demo I joined the two CTE using the regular syntax we use for joins. I could also join a CTE to a base table. The fact that a CTE has a name makes it easier to use when deriving more complex table expressions. The expression you use in the From clause to set up the result table can use base tables or more complex table expressions.

Demo 06: Joining the common table expression to a base table

```
With t_ord as
  (select order_id
   , order_date
   , customer_id
   , prod_id
   , quoted_price * quantity_ordered as ext_price
   from oe_orderHeaders
   join oe_orderDetails using (order_id) )
```

```

select
  customer_id
, customer_name_last
, prod_id
, ext_price
from cust_customers
join t_ord Using (customer_id);

```

CUSTOMER_ID	CUSTOMER_NAME_LAST	PROD_ID	EXT_PRICE
403000	Williams	1030	300
403000	Williams	1020	155.4
403000	Williams	1010	750
401250	Morse	1060	255.95
403050	Hamilton	1110	49.99
403000	Williams	1080	22.5
403000	Williams	1130	149.99
404950	Morris	1090	149.99
404950	Morris	1130	149.99
403000	Williams	1150	249.5
...	rows omitted		

For these types of queries it is a matter of preference where you write this as a simple multi-table join or as a series of CTEs. Often, people who have done more programming like the CTE style of writing queries.

In these examples, we are using the CTE as a way to move the complexity of a subquery away from the main query providing visual separation.

2.1. CTE order of definition

It is legal for one CTE to refer to another CTE as long as the expressions are defined in the proper order. This is a trivial example of such a CTE. The first expression filters for the customer name and the second expression uses the first to concatenate the name components. As we write more complex queries, you may prefer this style as a step-by-step approach to solving a problem. You might then decide to combine the CTE clauses into a single expression.

Demo 07: Using a CTE that refers to another CTE

```

With cte1 as (
  select customer_id, customer_name_first, customer_name_last
  from cust_customers
  where customer_name_first = 'William'
)
, cte2 as (
  select customer_name_first || ' ' || customer_name_last as cust_name
  from cte1)
select cust_name
from cte2;

```

CUST_NAME
William Northrep
William Morise
William Morris
William Morris
William Max

What we are seeing here is more of the trend to modularize your code, even within a single SQL statement.

We will use CTE in some of the demos for the rest of the semester including using a CTE to create a subquery that can be referred to multiple times in the main query.

RowNum is a pseudo column that can be used to number and limit the rows as they are returned to SQL*Plus. You can also think of RowNum as a function that returns a row number value. The use of RowNum is limited to Oracle SQL.

The tests that work correctly are limited to testing if RowNum is < value or <= value. Think of RowNum as counting off the rows as they are passed into the result set until the row count reaches the desired number.

RowNum is applied to the rows **before** any Order By or Distinct. This means the results might not be what you had hoped to achieve.

Demo 01: These are a few of the rows from the products table sorted by list price with the most expensive first.

```
select prod_id, prod_name, prod_list_price
from prd_products
order by prod_list_price desc;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
1126	WasherDryer	850
4568	Deluxe Cat Bed	549.99
4567	Deluxe Cat Tree	549.99
1120	Washer	549.99
1125	Dryer	500
1040	Treadmill	349.95
4569	Mini Dryer	349.95
1050	Stationary bike	269.95
1060	Mountain bike	255.95
1010	Weights	150
. . . rows omitted		

Demo 02: Now we try to get the five most expensive products using RowNum

```
select prod_id, prod_name, prod_list_price
from prd_products
where ROWNUM <= 5
order by prod_list_price desc;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
1010	Weights	150
1000	Hand Mixer	125
1030	Basketball	29.95
1070	Iron	25.5
1020	Dartboard	12.95

5 rows selected.

That is not what we wanted. What RowNum does is take the first 5 rows that come from the prd_products table and are passed to the result set and then it sorts those 5 rows. We will eventually solve the problem of the five most expensive items.

The next two pair of demos also point out what is happening. The first gets 10 rows from the products table using rownum <= 10. The second adds a Distinct and only 8 rows of the 10 rows are returned.

Demo 03: With rownum <= 10

```
select prod_list_price
from prd_products
where ROWNUM <= 10;
```

```
PROD_LIST_PRICE
-----
125
150
12.95
29.95
25.5
25.5
25.5
25
149.99
49.99
```

10 rows selected.

Demo 04: Including a Distinct and only 8 rows are returned

10 rows were taken from the original result set and that temporary result set was then filtered by the Distinct keyword, leaving only 8 rows.

```
select DISTINCT prod_list_price
from prd_products
where ROWNUM <= 10;
```

```
PROD_LIST_PRICE
-----
25
149.99
25.5
49.99
12.95
125
29.95
150
```

8 rows selected.

Demo 05: Including RowNum in the select may help you see what is happening. We get 5 rows coming from the table in whatever order the database engine delivers them. After the 5 rows are retrieved, the order by clause is applied- here the sort is by the price. We do get rownum values 1, 2, 3, 4, and 5.

```
select ROWNUM, prod_id, prod_name, prod_list_price
from prd_products
where rownum <= 5
order by prod_list_price desc;
```

ROWNUM	PROD_ID	PROD_NAME	PROD_LIST_PRICE
2	1010	Weights	150
1	1000	Hand Mixer	125
4	1030	Basketball	29.95
5	1070	Iron	25.5
3	1020	Dartboard	12.95

Some other tests might help you see how this works. All of these return 0 rows.

Demo 06:

```
select rownum, prod_id, prod_name, prod_list_price
from prd_products
where rownum = 5;
```

Demo 07:

```
select rownum, prod_id, prod_name, prod_list_price
from prd_products
where rownum between 5 and 10;
```

Demo 08:

```
select rownum, prod_id, prod_name, prod_list_price
from prd_products
where rownum >= 5;
```

The tests that work as we probably want are limited to testing if RowNum is **equal to or less than a value**.

RowNum is not the same as a Top or Limit keyword that you might be familiar with from another dbms.

One common use of RowNum is to limit the output to a few rows to see what type of data we have in a large table.

Learning to recognize patterns is important when working with data and data access. For this class, you need to recognize patterns in business requests for data and also sql patterns that are helpful for solving the business tasks.

Patterns help you to learn to recognize similarities between tasks (in the bluntest term- which demo is like a task on an assignment or exam). You may have solved the business task already. There is not a lot of difference between "Finding the names of the clients and the names of all of their animals" and "Finding the name of a publisher and the titles of all of the books they publish".

We have covered a number of comparatively simple types of queries. That does not mean they were always easy to learn; but they were queries that answered simpler questions- such as "Show me the names of the clients who have a cat. " or "Show me the id and names of the dogs that are more than 10 years old." We are now ready to move into more interesting queries.

- Show me the names of the clients who have a cat and a dog.
- Show me the names of the clients who have a cat but do not have a dog.
- Which staff person has done the most exams in the current year to date?
- Show me the most recent exam date for each of the reptiles.

What I would like you to do is think of similar types of queries in another field. For example, in the medical field, you might think of questions/queries such as:

- Show me the clients who are prescribed both medication A and medication B. We are concerned that these two medications are incompatible.
- Show me the clients who are taking medication C but not medication D. Maybe we just found out that medication C works better if the person also takes medication D.
- What is the most recent ALT blood test for each of our clients?
- Show me the top 100 clients in terms of their most recent ALT score.

What I would like you to see is that there are certain patterns of queries that are logically the same across various databases.

- Which vet clients have a cat but do not have a dog?
- Which of our Altgeld Mart customers have an order in the previous year but do not have an order in the current year?
- Which of our students have completed any of the 'CS15%A' classes but have not enrolled in any of the 'CS15%P' classes?

Patterns and Implementations

We discussed the unmatched rows pattern in the previous unit and implemented this with an outer join and a null test and also with a Not In subquery test where we had to consider the possibility of nulls in the subquery. We will see other ways to implement this logic over the semester. In the assignments, I often require a specific technique so that you have a chance to see and learn techniques. As the semester progresses you should keep a list of patterns and various implementation possibilities. The more tools you have in your tool box the better.

Examples:

1. (vets) Display all clients by name and the names of their reptiles. Include only clients who have a reptile. This uses two tables and a filter on the animal type. You could use an inner join. We need both the client name and the animal name.
2. (vets) Display all clients by name who have a reptile. This uses two tables and a filter on the animal type. You could use an inner join or a subquery.
3. (vets) Display all reptiles with no exam data. This uses two tables and a filter on the animal type. You could use an outer join or a subquery.

Table of Contents

1. Demo	1
2. Terminology and some general rules.....	2
3. Subquery using an IN list test.....	2
4. The ANY operator.....	5
5. The comparison operators.....	5
5.1. subquery with an equality test.....	5
5.1. subquery with other comparison tests	7

The use of subqueries is a technique that we will develop over the semester. In this document we look at some queries that have a main query and a subquery that is used in a test in the Where clause. I am going to start with a demo and then develop the rules for the subqueries.

1. Demo

These use the Altgeld Mart database. Suppose we want to find all of the customers who have any orders. We could do this with a join.

Demo 01: Inner join- this gives 97 rows with my current data set

```
select cs.customer_id, customer_name_last
from cust_customers cs
join oe_orderHeaders oh on cs.customer_id = oh.customer_id;
```

If I add the Distinct keyword then I get one row per customer who has an order- 21 rows.

```
select DISTINCT cs.customer_id, customer_name_last
from cust_customers cs
join oe_orderHeaders oh on cs.customer_id = oh.customer_id;
```

Demo 02: I could also do this with a subquery. This returns 21 rows, one row per customer who has an order.

```
select customer_id, customer_name_last
from cust_customers
where customer_id in (
    select customer_id
    from oe_orderHeaders);
```

The subquery is:

```
(select cust_id from oe_orderHeaders)
```

The subquery is executed and returns a list of all of the customer_id values from the order headers table- a list of customers with orders. That list is used by the IN filter so that we only get customers from the customer table who have an order. I do not have to use Distinct in either the main query or in the subquery. If a customer has more than one order, they still will appear only once in the result because the main query is pulling from the customers table.

I could display the order_date in the queries with the join, but I cannot display the order_date in the query using a subquery.

2. Terminology and some general rules

A subquery is a Select query that is enclosed in parentheses and is used within another statement. Subqueries can be used in Select, Insert, Update or Delete statements. For now, we will limit the discussion to subqueries in Select statements.

You can use the term **main query** for the top level Select. You can also refer to these two Selects as the **inner query** (the subquery) and the **outer query**. You can nest subqueries to many levels (255 levels in Oracle).

If a table appears only in a subquery and not in the outer query, then columns from the table in the inner query cannot be included in the output (the select list of the outer query).

A subquery can return 1 or more columns and 0 or more rows. But for certain uses of subqueries you need to restrict the subquery output.

3. Subquery using an IN list test

Suppose we want to find all of the orders for sporting goods items; we want a list of the order_id values for an order that included one or more sporting goods items.

We could do this with an inner join.

Demo 03: a query to find the order id of all orders that include sporting goods items.

```
select distinct ord_id
from oe_order_details OD
join prd_products PR on OD.prod_id = PR.prod_id
where catg_id = 'SPG'
order by ord_id;
```

ORD_ID
105
106
117
120
121
128
... rows omitted

Demo 04: We can also do this with a subquery that finds a list of product ids that are sporting goods from the products table and then delivers that list to the outer query which finds those product ids in the order_details table.

```
select DISTINCT order_id
from oe_orderDetails
where prod_id in (
    select prod_id
    from prd_products
    where catg_id = 'SPG'
)
order by order_id
;
```

Since there is more than one product id for sporting goods, we need to use the IN list syntax.

For this query we do want to use Distinct in the main query. We could have an order that includes more than one sporting goods items and our description said we wanted the order_id values for an order that included one or more sporting goods items. If we leave off the word Distinct in the outer select, then an order id will appear multiple times if a single order includes more than one sporting goods item line.

With the subquery approach, the only attributes that can be displayed are those found in the tables in the From clause of the main query. We could display attributes from the order detail tables but not the attributes from the products table.

If you compare the previous two query you can see that the attribute prod_id is used in the joins and also as the matching attribute in the subquery test. We are still "joining" the two tables based on the product id.

One of the nice thing about these subqueries is that they are stand alone queries that we could run separately to check our syntax and logic.

Demo 05: Suppose we ran this query first; we would get a row for each sporting goods item in the products table.

```
select prod_id, catg_id
from prd_products
where catg_id = 'SPG'
```

PROD_ID	CATG_I
1010	SPG
1020	SPG
1030	SPG
1040	SPG
1050	SPG
1060	SPG

6 rows selected.

Demo 06: But if we used that Select directly as a subquery we get an error.(Not one of Oracle's best error messages!)

```
select DISTINCT order_id
from oe_orderDetails
where prod_id in (
    select prod_id, catg_id
    from prd_products
    where catg_id = 'SPG'
)
order by order_id
;
ORA-00913: too many values
```

Demo 07: This finds sporting good products that were sold at their list price.

```
select order_id, prod_id
from oe_orderDetails
where (prod_id, quoted_price) in (
    select prod_id, prod_list_price
    from prd_products
    where catg_id = 'SPG'
)
order by order_id;
```

We are not limited in a subquery to a single table.

Demo 08: Suppose we want to see the customers who purchased a sporting goods item in Nov 2015.
We could do this with a subquery.

```
select customer_id, customer_name_last
from cust_customers CS
where customer_id in (
    select customer_id
    from oe_orderHeaders OH
    join oe_orderDetails OD on OH.order_id = OD.order_id
    join prd_products PR on OD.prod_id = PR.prod_id
    where to_char(order_date, 'YYYY-MM') = '2015-11'
    and catg_id = 'SPG');
```

Demo 09: first version using multiple subqueries(no joins). This has three subqueries. It does not yet filter on sporting goods or on the order date

```
select customer_id, customer_name_last
from cust_customers CS
where customer_id in (
    select customer_id
    from oe_orderHeaders OH
    where order_id IN (
        select order_id
        from oe_orderDetails OD
        where prod_id in (
            select prod_id
            from prd_products PR
        )
    )
);
```

Now we need to think about where to place the filters on sporting goods and on the order date.

The test on catg_id is simple; that attribute is in the products table so we put that filter in the last of the subqueries. (select prod_id from prd_products PR where catg_id = 'SPG').

The attribute for the order date is in the order headers table so we need the test at that level. I find it much easier to deal with the parentheses if I keep the simple filter before the In (subquery filter).

Demo 10: The test on order date is moved into the first subquery with a AND test

```
select customer_id, customer_name_last
from cust_customers CS
where customer_id in (
    select customer_id
    from oe_orderHeaders OH
    where to_char(order_date, 'YYYY-MM') = '2015-11'
    and order_id IN (
        select order_id
        from oe_orderDetails OD
        where prod_id in (
            select prod_id
            from prd_products PR
            where catg_id = 'SPG'
        )
    )
);
```

It is possible to run this query with the order_date filters in the innermost subquery. The inner subquery can reference columns in the outer queries. This can cause confusion when an identifier appears at more than one

level (SQL does not get confused- but you might). Generally avoid these types of tests unless you are very careful.

These examples have use the filter IN (list) . You should also consider using filters for NOT IN (list)'

Remember with the subquery approach, take care to consider if any of the subqueries would return a null. That requires special handling.

4. The ANY operator

This is another way to write the In list subquery test. It is not as common, but you may find it easier to read.

Demo 11: Compare to demo 04. We are finding order id values for orders that include a sporting goods item.

```
select distinct order_id
from oe_orderDetails
where prod_id = ANY (
    select prod_id
    from prd_products
    where catg_id = 'SPG'
)
order by order_id;
```

Demo 12: Oracle also allows this operator with a simple list of values

```
select prod_id , catg_id
from prd_products
where catg_id = ANY('SPG', 'MUS');
```

5. The comparison operators

5.1. subquery with an equality test

Demo 13: Suppose we want to find order id values for orders that include a sporting goods item and we use the following where we use an equality test with the subquery. This gives us an error message.

```
select distinct order_id
from oe_orderDetails
where prod_id = (
    select prod_id
    from prd_products
    where catg_id = 'SPG'
)
order by order_id;
```

ORA-01427: single-row subquery returns more than one row
--

This is not allowed when the subquery follows a comparison operator: =, !=, <, <= , >, >= .

If you run the subquery by itself, you get 6 rows returned. So this query is trying to test if the prod_id value in the order details table equals 6 different values at the same time. An IN list test is actually an OR test so demo 4 tests if the prod_id value in the order details table equals the first value brought back by the subquery or the second value brought back by the subquery, etc. (This is also why you run into troubles when the subquery could contain a null.)

We do not want an equality test in the outer query if the subquery could return multiple rows.

Demo 14: What if the subquery returns no rows? The subquery returns an empty virtual table and the outer query finds no matches and also returns an empty table. This is not considered an error.

```
select distinct order_id
from oe_orderDetails
where prod_id = (
    select prod_id
    from prd_products
    where catg_id = 'QWE'
)
order by order_id;
no rows selected
```

Suppose we want to find all of the people who work in the same department as employee 162. We could write a query to find that department id.

```
select dept_id
from emp_employees
where emp_id = 162
```

And then write a query to find employees in that department.

```
select emp_id
, name_last as "Employee"
, dept_id
from emp_employees
where dept_id =      <-- put the department number here
```

But we can also build these two queries into one query by using a subquery.

Demo 15: A simple subquery- we want to find employees who work in the same department as employee with ID 162 . Since we are looking at the data in the employee table in two different ways, both the main and the subquery use the employees table.

```
select emp_id, name_last as "Employee"
from emp_employees
where dept_id = (
    select dept_id
    from emp_employees
    where emp_id = 162
)
```

EMP_ID	EMPLOYEE
162	Holme
200	Whale
207	Russ

The inner query, the subquery, is a complete query that could stand by itself. It returns a single value- the department id where this employee works. The subquery is enclosed in parentheses and the result of the subquery is used by the outer query to return the other employees from that department.

Demo 16: If we really want to return the **other** people from that department, we can eliminate employee 162 from the final result set. Note that the filter $emp_id \neq 162$ is written outside the subquery so that it filters the final result set.

```
select emp_id, name_last as "Employee"
from emp_employees
where dept_id = (
    select dept_id
    from emp_employees
    where emp_id = 162
)
```

```

select dept_id
from emp_employees
where emp_id = 162
)
and emp_id <> 162
;

```

EMP_ID	EMPLOYEE
200	Whale
207	Russ

With this filter we have to take care that the subquery returns a single value- one row and one column, because we are using it in an = test.

5.1. subquery with other comparison tests

We could use this technique to find employees who earn more than employee 162.

The subquery to find the salary for employee 162 would be

```

select salary
from emp_employees
where emp_id = 162

```

Each employee has exactly one salary value and we have only one employee 162 since emp_id is the primary key. This subquery returns one value.

Demo 17: Using the subquery with an > test

```

select emp_id, name_last as "Employee", salary
from emp_employees
where salary > (
    select salary
    from emp_employees
    where emp_id = 162
);

```

EMP_ID	Employee	SALARY
100	King	100000
101	Koch	98005
161	Dewal	120000
204	King	99090

Try this query with some different employee id value- who earns more than employee 104? More than employee 161?

Demo 18: Now try this query where the subquery filters on the department id. You will get an error message

```
select emp_id, name_last as "Employee", salary  
from emp_employees  
where salary > (  
    select salary  
    from emp_employees  
    where dept_id = 210  
) ;
```

```
ORA-01427: single-row subquery returns more than one row
```

This is a good error message. The subquery is

```
select salary  
from emp_employees  
where dept_id = 210
```

We have two employees in dept 210, so the result of that subquery is

```
SALARY  
-----  
69000.00  
55000.00  
  
2 rows selected
```

So are we asking who earns more than 69000 or more than 5500? This is an invalid query; you cannot use an = or a > test against a subquery that returns more than one row.

Table of Contents

1. Departments and employees.....	1
1.1. Multi-matches	3
1.2. Non-matches and mixed matches.....	4
2. Demos with customers and orders and products ordered.....	6
3. The Multi-match pattern.....	9

There is another type of query- a multi-match query- that we will see implemented in various ways over the semester. We have looked at this pattern already in a simpler format.

This is not an easy topic. You should read through this once- sort of as a story. Then go back and study this. This keeps popping up this semester often. You might want to read the last section of this document first.

1. Departments and employees

The Where clause we have been using is called a row filter because it is used to examine the data in the From clause's virtual table **one row at a time** to see if that row will be allowed in the final result.

We are going to start with the employee table. This table is set up with the assumption that each employee is identified by an employee id; that is the primary key for the employee table. That means each employee gets exactly one row in the employee table. The definition of a row in the employee table has one attribute for the employee's job (job_id) which is a not null column and one attribute for dept_id which is also not null. This means that for each employee we store only one value for job_id, the employee's current job and one value for dept_id, the employee's current department.

Demo 01: We have currently 22 employees and each employee has a job id value and a dept_id..

EMP_ID	DEPT_ID	JOB_ID
100	10	1
201	20	2
101	30	16
108	30	16
109	30	32
110	30	32
203	30	16
204	30	32
205	30	16
206	30	32
162	35	16
200	35	16
207	35	8
145	80	4
150	80	8
155	80	8
103	210	64
104	210	32
102	215	64
146	215	64
160	215	32
161	215	16

These are the rows in the departments table. Note that dept_id 90 and 95 do not have any employees.

DEPT_ID	DEPT_NAME
10	Administration
20	Marketing
30	Development
35	Cloud Computing
80	Sales
90	Shipping
95	Logistics

210 IT Support
215 IT Support

Demo 02: We can filter for employees with job_id 16 and each row in the employees table will be looked at to see if the job_id for that row is 16 or not.

```
select emp_id, dept_id, job_id
from emp_employees
where job_id = 16
order by dept_id, emp_id;
```

EMP_ID	DEPT_ID	JOB_ID
101	30	16
108	30	16
161	215	16
162	35	16
200	35	16
203	30	16
205	30	16

7 rows selected

Demo 03: We can use the Or operator to find employees with job id 16 or with job id 32. We could also use an IN list

```
select emp_id, dept_id, job_id
from emp_employees
where job_id = 16 or job_id = 32
order by emp_id;
```

EMP_ID	DEPT_ID	JOB_ID
101	30	16
104	210	32
108	30	16
109	30	32
110	30	32
160	215	32
161	215	16
162	35	16
200	35	16
203	30	16
204	30	32
205	30	16
206	30	32

13 rows selected

Demo 04: If we use the AND operator to find employees with job id 16 and with job id 32, we get no rows returned because each row in the employee table is looked at one row at a time, and no row has the value 16 and the value 32 for job id at the same time.

```
select emp_id, dept_id, job_id
from emp_employees
where job_id = 16 AND job_id = 32
order by emp_id;
```

no rows selected)

1.1. Multi-matches

Now we want to look at employees and their jobs from a department point of view.

Demo 05: Suppose we want to see which departments have an employee with job id 16. We can do this with the filter we had earlier, show just the department id and use distinct to remove duplicate rows- these would be departments where we have more than one employee with job 16.

```
select distinct dept_id
from emp_employees
where job_id = 16 ;
```

DEPT_ID
30
35
215

Demo 06: Same query for departments with employees with job 32

```
select distinct dept_id
from emp_employees
where job_id = 32 ;
```

DEPT_ID
30
210
215

Demo 07: This gives us departments where we have employees who have either job 16 or job 32. Compare this result with the results of the previous two queries.

```
select distinct dept_id
from emp_employees
where job_id = 16 or job_id = 32;
```

DEPT_ID
30
35
210
215

But suppose we try to use the AND operator to find departments which have both employees with job id 16 and employees with job id 32. We get no rows for the same reason we did in the earlier AND filter since no employee has two different job id values. So the And test is never met.

Demo 08:

```
select distinct dept_id
from emp_employees
where job_id = 16 AND job_id = 32
;
```

no rows selected)

This is one of those rules that seems trivial when you read it, but it causes all kinds of errors on assignments and on exams. If you are looking at a single table as the table expression, each cell has only a value. The job_id cell cannot have two different values **in the same row**.

We have to approach this from another way. We are still thinking about row filters, but we are asking the question about the **departments** so we start with the department table. Our query starts with a row from the department table and asks two questions

- (1) are you a department that has an employee with job id 16
- and (2) are you a department that has an employee with job id 32.

These are two separate questions. If this department row meets both tests then it is passed into the result. We are asking two questions here and both questions have to be met - that is why I am calling these multiple-match queries.

Demo 09: Compare this result to the previous ones. This query says show me the departments where the department has at least one employee with job 16 and the department has at least one employee with job 32. The department has to pass both of these tests to get into the output.
This uses both the department table and the employee table.

```
select dept_id
from emp_departments
where dept_id in (
    select dept_id
    from emp_employees
    where job_id = 16 )
and dept_id in (
    select dept_id
    from emp_employees
    where job_id = 32 );
```

DEPT_ID

30
215

There is a version in the demos that uses only the employee table, but I think this version will be clearer for some other examples.

You should be able to think of how to write queries for the following:

Show the departments that have employees with job id 16 and employees with job id 32 and employees with job id 8.

Show the departments that have employees with job id 16 and employees with either job id 32 or job id 8.

1.2. Non-matches and mixed matches

Now let's try queries that do non-matches- find department which do not have an employee with job_id 16. Before we start we need to think about department 90 and 95; they do not have any employees. Should they be returned by our query? In one sense- they should- if the department has no employees then it certainly has no employees with job_id 16. But that might not be the intent- maybe we really are supposed to find department with employees but not with employees with job_id 16. You probably should verify what is actually wanted with the person asking you to write the query.

Compare the following two queries and figure out why one returns dept 90 and 95 and the other doesn't and figure out why I included Distinct in one of these and not in the other. If you have troubles with this- ask in the forum.

Demo 10:

```
select dept_id
from emp_departments
where dept_id NOT IN (
    select dept_id
```

```
from emp_employees
where job_id = 16 )
order by dept_id;
```

DEPT_ID

10
20
80
90
95
210

Demo 11:

```
select distinct dept_id
from emp_employees
where dept_id NOT IN (
    select dept_id
    from emp_employees
    where job_id = 16 )
order by dept_id;
```

DEPT_ID

10
20
80
210

Demo 12: Why is this query incorrect for our task? What is this query actually returning?

```
select distinct dept_id
from emp_employees
where dept_id IN (
    select dept_id
    from emp_employees
    where job_id != 16 )
order by dept_id;
```

DEPT_ID

10
20
30
35
80
210
215

Demo 13: Show the departments that have employees with job id 16 and have no employees with job id 32.

```
select dept_id
from emp_departments
where dept_id IN (
    select dept_id
    from emp_employees
    where job_id = 16 )
and dept_id NOT IN (
    select dept_id
    from emp_employees
    where job_id = 32 )
order by dept_id;
```

DEPT_ID

35

These are not different subqueries than used in the previous documents; we are just using them in new ways. Some of these can be written more simply, but for now I want you to see the pattern of subquery testing.

2. Demos with customers and orders and products ordered

These examples will go through similar examples with slightly more complex settings. Remember some of these examples will look reasonable at first- but may be logically incorrect.

When you create a query you need to consider that the Where clause operates on a single row of the virtual table produced by the From clause.

We want to find customers who have purchased both a stationary bike (product id 1050) and a stationary bike (product id 1060) on the same order. Your first attempt might be the following which is not correct.

Demo 14: Using an In List test

```
select OH.customer_id, order_id, OD.prod_id
from oe_orderHeaders OH
join oe_orderDetails OD using (order_id)
where prod_id in (1050, 1060)
order by OH.customer_id, order_id, OD.prod_id
;
```

CUSTOMER_ID	ORDER_ID	PROD_ID
401250	106	1060
403000	505	1060
403000	511	1060
403000	536	1050
404100	605	1060
404100	805	1060
404950	411	1060
404950	535	1050
408770	405	1050
408770	405	1060
409030	128	1060
903000	312	1050
903000	312	1060
903000	312	1060

14 rows selected.

Looking at the result the first customer returned is cust_id 401250 and that customer ordered product 1060 but not product 1050. We do not want that customer id returned because he did not buy both products. The In List filter is the equivalent of an OR test.

Demo 15: Using an OR test

```
select OH.customer_id, order_id, OD.prod_id
from oe_orderHeaders OH
join oe_orderDetails OD using (order_id)
where prod_id = 1050
OR prod_id = 1060
order by OH.customer_id, order_id, OD.prod_id;
```

Demo 16: You might then try an AND operator since you want customers who bought product 1050 AND product 1060. But that returns no rows.

```
select OH.customer_id, order_id, OD.prod_id
from oe_orderHeaders OH
join oe_orderDetails OD using (order_id)
where prod_id = 1050
and prod_id = 1060
order by OH.customer_id, order_id, OD.prod_id;
```

No rows selected

Demo 17: If you look at the rows in the virtual table created by the FROM clause, we have a series of rows with a single column for the product ID. These are some of those rows. I added rownum so we can talk about the rows.

```
select rownum, OH.cust_id, ord_id, OD.prod_id
from oe_order_headers OH
join oe_order_details OD using (ord_id);
```

ROWNUM	CUSTOMER_ID	ORDER_ID	PROD_ID
1	403000	105	1020
2	403000	105	1030
3	403000	105	1010
4	401250	106	1060
5	403050	107	1110
6	403000	108	1080
7	403000	109	1130
8	404950	110	1090
9	404950	110	1130
10	403000	111	1150
11	403000	111	1141
12	401890	112	1110
...			

Note that each row has one cust_id value and one ord_id value and one prod_id value.

With a Where clause each of those rows is checked, one row at a time.

If our Where clause is

```
Where prod_id = 1050 OR prod_id = 1060
```

Then the first row evaluates to False and is not returned. The second row evaluates to False and is not returned.

The third row evaluates to False and is not returned. The fourth row evaluates to True and is returned. **But the tests in the Where clause never looks at more than one row at a time.**

If our Where clause is

```
Where prod_id = 1050 AND prod_id = 1060
```

Then we are looking for rows where the single value for prod_id in a row is *both* 1050 and 1060. This is never going to happen with our From clause.

But we can solve this problem. We are really asking a question about orders- **so we should start with the order headers table.**

Demo 18: Using two subqueries gives us the correct result. This query says show me the orders where the order has at least one detail row with an order for product 1050 and the (same) order with has at least one detail row with an order for product 1060. The order id has to pass both of these tests to get into the output.

The ord_id being tested in the main query is coming from the order headers table.

```
select OH.customer_id, order_id
from oe_orderHeaders OH
where order_id in (
    select order_id
    from oe_orderDetails OD
    where prod_id = 1050)
and order_id in (
    select order_id
    from oe_orderDetails OD
    where prod_id = 1060)
order by OH.customer_id, order_id;
```

CUSTOMER_ID	ORDER_ID
408770	405
903000	312

We are looking at each row in the oe_order_headers rows and using a Where clause with an AND test. We can read that Where clause as - we want an order id that is in the details table for orders for product 1050 and is also in the details table for orders for product 1060- which is what we want.

If we run just the first subquery:

```
select order_id
from oe_orderDetails OD
where prod_id = 1050
```

ORDER_ID
312
405
535
536

4 rows selected.

If we run the second subquery:

```
select order_id
from oe_orderDetails OD
where prod_id = 1060;
```

ORDER_ID
106
128
312
312
405
411
505
511
605
805

10 rows selected.

What we want is the order id values that are in both of those tables.

Now suppose we want to find customers who bought both of these products but not necessarily on the same order. **This is a question about customers.** Now we want to test for the customer id twice- once for an order for product 1050 and once for an order for product 1060.

Demo 19: Using two subqueries gives us the correct result.

```
select customer_id, customer_name_last
from cust_customers
where customer_id in (
    select customer_id
    from oe_orderHeaders OH
    join oe_orderDetails OD using (order_id)
    where prod_id = 1050)
and customer_id in (
    select customer_id
    from oe_orderHeaders OH
    join oe_orderDetails OD using (order_id)
    where prod_id = 1060)
order by customer_id;
```

CUST_ID
403000
404950
408770
903000

4 rows selected.

Demo 20: You might want to consider using a CTE for a query like this. You write the join of order headers and order details once and refer to it twice.

```
select customer_id, customer_name_last
from cust_customers
where customer_id in (
    select customer_id
    from oe_orderHeaders OH
    join oe_orderDetails OD using (order_id)
    where prod_id = 1050)
and customer_id in (
    select customer_id
    from oe_orderHeaders OH
    join oe_orderDetails OD using (order_id)
    where prod_id = 1060)
order by customer_id;
```

3. The Multi-match pattern

These are not trivial queries. But they are useful queries. The first thing you need to do is recognize this pattern. We want to find customers who purchased product 1050 **and also** product 1060. That is not the same as finding customers who purchased product 1050 or product 1060. The customer has to pass two distinct tests to get into the result.

And it is critical to determine if this is a question that is mainly about customers, or about orders or about departments.

It is easier to recognize patterns if you think of other examples:

1. We want clients at the vet clinic who have both a hedgehog and a porcupine.
2. We want animals that had an exam last year and also had an exam this year.
3. We want students who passed CS 110A and 110B.
4. We want to hire people who have both SQL Server experience and Oracle experience.

Then you should expand the patterns a bit.

5. We want clients at the vet clinic who have a hedgehog but not a porcupine. This is still two separate tests.
6. We want animals that had an exam last year but do not have an exam this year. This is a positive match and a negative match.
7. We want students who passed CS 111A and 111B and 111C. This is three matches.
8. We want clients at the vet clinic who have a hedgehog and a porcupine and a dormouse. This is three matches.
9. We want students who passed (CS 110A or CS 111A) and 110B.
10. We want clients at the vet clinic who have (a hedgehog or a porcupine) and a dormouse.
11. We want clients at the vet clinic who have (a hedgehog or a porcupine) but not a dormouse.

The more you do this, the easier it will be to recognize this pattern on the midterm exam and on the final exam. You should also recognize this pattern with respect to targeted advertising and medical decisions (which patients are over 65 years old and do not have a flu shoot- which is rather like example 5.)

Then you have to remember that you cannot solve these problems with a simple table expression in the From clause and a filter that uses an OR test, or an In list- which is an Or test, or a AND test. These are more complex. The next thing to think through is what you are actually testing. In the last demo, we want to find customers who bought both of these products. So the filter is on the **customer** (customer_id). The tests ask if the **customer** is in the list of people who purchased product 1050 and if the **customer** is in the list of people who purchased product 1060.

Table of Contents

1. Using a single subquery.....	1
2. Using multiple subqueries.....	2
2.1. Nesting subqueries.....	4

A table expression determines a virtual table. We commonly see table expressions in the From clause of a query. We started with the simplest table expression- a single base table. (Remember a base table is a persistent table; we create the base table with a Create Table statement.).

Then we added Inner Joins and Outer Joins to connect two or more base tables to create a virtual table. That join is a table expression.

```
select an_name, cl_name_last
from vt_animals an
join vt_clients cl on an.cl_id = cl.cl_id;
```

Now we are going to discuss a technique that uses a subquery as a table expression. This is sometimes called an inline view.

The use of CTE to compartmentalize a complex query is elegant and fairly easy to read and write with a little practice. The use of a CTE is similar to the use of an inline view with the added advantage that you can use the CTE more than once in the query and it may seem cleaner to define the CTE at the top of the query rather than in the middle of the From clause.

1. Using a single subquery

Suppose you have a fairly complex query dealing with customer orders that you need to run only for a particular query. You would like to break the query down into smaller, more manageable chunks that you could test separately. One solution is to create a subquery that handles part of the query and then use that query in the From clause of the main query. Since we are using this subquery as a table expression, the subquery can have multiple columns and multiple rows. We also will provide a table alias to have a name for the subquery table expression.

This query does not work since you cannot use the column alias as a column name in the same Select clause,

```
select cl_name_last || ' ' || cl_name_first as ClientName
, ClientName || ' lives in ' || cl_state
from vt_clients;
```

Demo 01: Using a subquery in the From clause. The subquery exposes the alias ClientName which we can then use in the Where clause of the main query. The subquery table alias is ClientNames

```
select ClientName || ' lives in ' || cl_state as "Message"
from (
    select cl_name_first || ' ' || cl_name_last as ClientName
    , cl_state
    from vt_clients
) ClientNames;
```

```
Message
-----
Stanley Turrentine lives in CA
Wes Montgomery lives in OH
Theo Monk lives in NY
Coleman Hawkins lives in OH
...
Edger Boston lives in MA
Sue Biederbecke lives in IL
Sam Biederbecke lives in CA
Biederbecke lives in IL
Biederbecke lives in CA
```

The subquery is shown here. It is a Select that exposes the cl_state and an expression named ClientName

```
select cl_name_first || ' ' || cl_name_last as ClientName
, cl_state
from vt_clients
```

The subquery is enclosed in parentheses, given a table alias, and placed in the From clause of the main query. The main query can use the exposed columns from the subquery. That allows us to use the calculated column by referencing its alias.

We could also focus on dealing with that space that shows up with a null first name in the subquery.

Demo 02: This is a more complex subquery that assembles the data for the orders and exposes three columns which are used in the main query.

```
select order_id
, order_date
, itemTotal
from (
    select
        order_id
        , order_date
        , customer_id
        , quoted_price * quantity_ordered as itemTotal
    from oe_orderHeaders
    join oe_orderDetails using(order_id)
    join prd_products using(prod_id)
    where quoted_price > 0 and quantity_ordered > 0
    Order by order_id
) rpt_base
where order_date < '01-NOV-2015'
;
```

ORDER_ID	ORDER_DATE	ITEMTOTAL
105	01-OCT-15	155.4
105	01-OCT-15	300
105	01-OCT-15	750
106	01-OCT-15	255.95
107	02-OCT-15	49.99
108	02-OCT-15	22.5
109	12-OCT-15	149.99
110	12-OCT-15	149.99
110	12-OCT-15	149.99
301	04-JUN-15	205
302	04-JUN-15	120
...		

2. Using multiple subqueries

Demo 03: This uses two subqueries and joins them. Each subquery has a name. The subqueries produce virtual tables and we are just joining the two virtual tables.

```
select
    customer_id
    , customer_name
```

```

, prod_id
, ext_price
from (
    select
        customer_id
        , customer_name_first || ' ' || customer_name_last as customer_name
    from cust_customers
    where lower(customer_name_first) = 'william'
) t_cust
join (
    select
        order_id
        , order_date
        , customer_id
        , prod_id
        , quoted_price * quantity_ordered as ext_price
    from oe_orderHeaders
    join oe_orderDetails using (order_id)
) t_ord using (customer_id)
;

```

CUSTOMER_ID	CUSTOMER_NAME	PROD_ID	EXT_PRICE
404950	William Morris	1090	149.99
404950	William Morris	1130	149.99
401890	William Northrep	1110	99.98
402100	William Morise	1130	625
402100	William Morise	1000	200
402100	William Morise	1120	1900
402100	William Morise	1080	25
402100	William Morise	1100	180
402100	William Morise	1150	19.96
402100	William Morise	1141	300

The From clause here is

```

From (subQuery1) t_cust
Join (subQuery2) t_ord using (cust_id)

```

The demo has the Using (cust_id) syntax for the join. We could also code the join with the syntax

```

Select . . .
From (subQuery1) t_cust
Join (subQuery2) t_ord on t_cust.cust_id = t_ord.cust_id;

```

Since we are joining the two virtual tables on the cust_id values, each subquery needs to expose that column. The first subquery contributes the cust_name and the second subquery contributes the prod_id and the ext_price.

Demo 04: Joining a subquery table expression to a base table

```

select customer_id
, customer_name_last
, prod_id
, ext_price
from cust_customers
join (select customer_id
        , prod_id
        , quoted_price * quantity_ordered as ext_price
    from oe_orderHeaders OH
    join oe_orderDetails OD on OH.order_id = OD.order_id
) t_ord using (customer_id)
;

```

2.1. Nesting subqueries

Demo 05: This nests two subqueries in the From clause. As it stands it is simply a complex way to get customers with the first name William, but it does show nested subqueries

```
select customer_name
from (
    select customer_name_first || ' ' || customer_name_last as customer_name
    from (
        select
            customer_id
            , customer_name_first
            , customer_name_last
        from cust_customers
        where lower(customer_name_first) = 'william'
    ) tbl_william
) tbl_concat
;
CUSTOMER_NAME
-----
William Northrep
William Morise
William Morris
William Morris
William Max
```

One difference we will see later where the CTE is even more of an advantage is that we can define a CTE once in the query and then use it multiple times in the query. If we need to do that logic with a subquery, we would have to repeat that subquery multiple times in the query.

Table of Contents

1. Getting started with test data.....	1
2. Test Tables and Rows	3
3. Queries	5
4. Things you should NOT do for these patterns	8

As you develop more complex queries you will want to consider setting up a test plan to help decide if your query is working correctly. The first- and sometimes the hardest - step is to figure out what the task asks for. I have been trying to write the tasks using more of an English approach than an SQL approach. For example, I ask you to display the client's last name, not the cl_name_last attribute. Often setting up a test plan will help you understand the task description. The next step could be to set up a few sets of data that meet (or do not meet) the task description and state what result you expect. Now you have to decide what to do with that data. You might think of adding the data sets to tables we already have- which can create its own problems. Or you can set up a few tables that are very similar to the database tables we already have, but simplified.

Let's take a few of the query patterns from the previous documents and work out some tasks related to these patterns. These are some basic patterns using the vets tables.

1. We want clients at the vet clinic who have a hedgehog.
2. We want clients at the vet clinic who have both a hedgehog and a porcupine.
3. We want clients at the vet clinic who have a hedgehog but not a porcupine.
4. We want clients at the vet clinic who have a hedgehog and a porcupine and a dormouse.
5. We want clients at the vet clinic who have (a hedgehog or a porcupine) and a dormouse.
6. We want clients at the vet clinic who have (a hedgehog or a porcupine) but not a dormouse.
7. We want clients at the vet clinic who do not have a hedgehog.

1. Getting started with test data

What we want to do here is start with some data that we could use to check our understanding of the question to be answered. We need to include data that does not exist in our vets tables. Maybe we don't have any clients with a hedgehog!

I started with a simple question- we want clients who have a hedgehog. Assume we will have to display the client id and name. We will know if a client has a hedgehog if the client's id is in the animal table for a row where an_type = 'hedgehog'.

The an_type is not null in the animals table. We do not have to worry about a possibility of a client having an animal but we do not know what type of animal it is.

So for each client there are two possibilities- either they have a hedgehog or they do not have a hedgehog. We do not care how many hedgehogs they might have- just if they have at least one or if they do not have any hedgehogs. (It might be a good idea to check this assumption.)

We can start with a simple tabular display- These are easy to set up in a spreadsheet program- or on a piece of paper. These are called truth tables and are very helpful in thinking about a question.

We have only one logical predicate: 'This client has a hedgehog'.

P1 = this client has a hedgehog.

This is our table of possibilities. The simplest tables generally look like they are too easy to bother with. But start easy.

P1
TRUE
FALSE

Question 2 wants clients with both a hedgehog and a porcupine. That involves two tests. I will call these predicates P2A and P2B

P2A = this client has a hedgehog.

P2B = this client has a porcupine.

P2 = this client has both a hedgehog and a porcupine

P2 = P2A AND P2B

Since there are 2 possible values for P2A and 2 possible values for P2B, there are 4 possible combinations of values.

P2A	P2B	P2A AND P2B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Now you are probably thinking about sql- stop that and think about how many clients we need to have for testing.

For question 1 we need two clients- one with a hedgehog and one without a hedgehog.

For question 2 we need four clients- one with a hedgehog and a porcupine, one with a hedgehog and no porcupine, one with a porcupine and no hedgehog, and one with neither a hedgehog, nor a porcupine.

We can use the same set of record for question 3.

P3A = this client has a hedgehog.

P3B = this client has a porcupine.

P3 = this client has a hedgehog and this client does not have a porcupine

P3 = P3A AND (NOT P3B)

P3A	P3B	NOT P3B	P3A AND (NOT P3B)
TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE

Now you need to imagine that someone is going to write the query and they do not write very good SQL.

Perhaps they write the query to test for exactly one hedgehog. So we might want a few more rows of tests- a client with 4 hedgehogs and 2 porcupines

a client with 3 hedgehogs and no porcupines

a client with only one dog.

a client with no animals at all

When I get to the questions that test for hedgehogs, porcupines and dormice, then I will have a bigger table.

P4A = this client has a hedgehog.

P4B = this client has a porcupine.

P4C = this client has a dormouse.

P4 = this client has a hedgehog and a porcupine and a dormouse

P4 = P4A AND P4B AND P4C

P5 = ?

P6 = ?

This is 8 possible patterns

	P4A	P4B	P4C	P4	P5	P6
1	TRUE	TRUE	TRUE	TRUE		
2	TRUE	TRUE	FALSE	FALSE		
3	TRUE	FALSE	TRUE	FALSE		
4	TRUE	FALSE	FALSE	FALSE		
5	FALSE	TRUE	TRUE	FALSE		
6	FALSE	TRUE	FALSE	FALSE		
7	FALSE	FALSE	TRUE	FALSE		
8	FALSE	FALSE	FALSE	FALSE		

2. Test Tables and Rows

Now I could start adding rows to my tables for vt_clients and vt_animals- but those tables have a lot of columns I really do not care about- there is nothing in these questions that needs to know what city the client lives in or the animal dob. So I am going to set up two test tables. The test table for the animals will have attributes for an_id, cl_id, and an_type. The test table for the clients will have a cl_id attribute and an attribute for client name. I need to use data types to match those in the vt_clients and vt_animals tables. I also need to keep the relationship between these two tables.

I want to keep the tables as simple as possible- but no simpler.

Demo 01: I kept the pk and fk constraints and any constraint on those attributes.

```
create table tst_clients(
    cl_id          int
    , cl_name_last  varchar(25)      not null
    , constraint tst_clients_pk primary key (cl_id)
);

create table tst_animals(
    an_id          int
    , an_type        varchar(25)      not null
    , cl_id          int              not null
    , constraint tst_animals_clients_fk foreign key (cl_id) references tst_clients(cl_id)
    , constraint tst_animals_pk primary key (an_id)
);
```

Demo 02: Inserts - I am using the client name as a code to the animals for that client. The name 'H_P_D_' means this is a client with a Hedgehog and a porcupine and a Dormouse. If you look at some of the sample runs, you can see that the underscores make that column easier to read and scan.

```
insert into tst_clients values (1, 'H_P_D_');
insert into tst_animals values (10, 'hedgehog', 1);
insert into tst_animals values (11, 'porcupine', 1);
insert into tst_animals values (12, 'dormouse', 1);

insert into tst_clients values (2, 'H_P_____');
insert into tst_animals values (13, 'hedgehog', 2);
insert into tst_animals values (14, 'porcupine', 2);
```

```

insert into tst_clients values (3, 'H_____D____');
insert into tst_animals values (15, 'hedgehog', 3);
insert into tst_animals values (16, 'dormouse', 3);

insert into tst_clients values (4, 'H_____P_____');
insert into tst_animals values (17, 'hedgehog', 4);

insert into tst_clients values (5, '_____P_D____');
insert into tst_animals values (18, 'porcupine', 5);
insert into tst_animals values (19, 'dormouse', 5);

insert into tst_clients values (6, '____P_____');
insert into tst_animals values (20, 'porcupine', 6);

insert into tst_clients values (7, '_____D____');
insert into tst_animals values (21, 'dormouse', 7);

insert into tst_clients values (8, '_____');

-- And a few additional inserts
insert into tst_clients values (9, 'H4_P2_____');
insert into tst_animals values (22, 'hedgehog', 9);
insert into tst_animals values (23, 'hedgehog', 9);
insert into tst_animals values (24, 'hedgehog', 9);
insert into tst_animals values (25, 'hedgehog', 9);
insert into tst_animals values (26, 'porcupine', 9);
insert into tst_animals values (27, 'porcupine', 9);

insert into tst_clients values (10, 'H3_____');
insert into tst_animals values (28, 'hedgehog', 10);
insert into tst_animals values (29, 'hedgehog', 10);
insert into tst_animals values (30, 'hedgehog', 10);

insert into tst_clients values (11, 'dog');
insert into tst_animals values (31, 'dog', 11);

```

Before you start writing and running queries, decide what you want the result to look like. It is very easy to just be happy to get any result- you need the correct result.

```
select * from tst_clients;
```

cl_id	cl_name_last
1	H_____P_D____
2	H_____P_____
3	H_____D____
4	H_____
5	_____P_D____
6	_____P_____
7	_____D____
8	_____
9	H4_P2_____
10	H3_____
11	dog

Note that client 8 does not come back with our joined tables because that client has no animals.

```
select cl.cl_id, cl_name_last, an_id, an_type
from tst_clients cl
join tst_animals an  on cl.cl_id = an.cl_id
;
```

cl_id	cl_name_last	an_id	an_type
1	H P D	10	hedgehog
1	H P D	11	porcupine
1	H P D	12	dormouse
2	H P	13	hedgehog
2	H P	14	porcupine
3	H D	15	hedgehog
3	H D	16	dormouse
4	H	17	hedgehog
5	P D	18	porcupine
5	P D	19	dormouse
6	P	20	porcupine
7	D	21	dormouse
9	H4 P2	22	hedgehog
9	H4 P2	23	hedgehog
9	H4 P2	24	hedgehog
9	H4 P2	25	hedgehog
9	H4 P2	26	porcupine
9	H4 P2	27	porcupine
10	H3	28	hedgehog
10	H3	29	hedgehog
10	H3	30	hedgehog
11	dog	31	dog

22 rows selected

3. Queries

Demo 03: Question 1: we should get cl_ids 1,2,3,4,9,10

```
select cl.cl_id, cl.cl_name_last
from tst_clients cl
join tst_animals an on cl.cl_id = an.cl_id
where an_type = 'hedgehog'
;
```

cl_id	cl_name_last
1	H P D
2	H P
3	H D
4	H
9	H4 P2
10	H3
10	H3
10	H3

There is no reason to display a client multiple times (such as client 9). We could fix this with Distinct in the select or by using a subquery. Since we are talking about subqueries, we will use that approach. The columns I want to display are both in the clients table so that is the table in the main query.

Demo 04: Question 1: we should get cl_ids 1,2,3,4,9,10. Note that the subquery is P1 with a value of True. Assuming I did my cl_name_last values correctly, I can scan down these rows and see 'H' is each column. I should also check that the rows that are not displayed are client which do not have hedgehogs. In this case I might just wish to compare the list for the entire table and check the clients no returned. It is easy to forget the "negative test".

```
select cl.cl_id, cl.cl_name_last
from tst_clients cl
where cl.cl_id in (
    select cl_id
    from tst_animals
    where an_type = 'hedgehog');
```

cl_id	cl_name_last
1	H P D
2	H P
3	H D
4	H
9	H4 P2
10	H3

Demo 05: Question 2: We want clients who have both a hedgehog and a porcupine; we should get cl_ids 1,2,9.
The client needs to pass two tests: P2A and P2B. Visually compare fpor the negative test- client 3 does not show up- is that correct?

```
select cl.cl_id, cl.cl_name_last
from tst_clients cl
where cl.cl_id in (
    select cl_id
    from tst_animals
    where an_type = 'hedgehog')
and cl_id in (
    select cl_id
    from tst_animals
    where an_type = 'porcupine');
```

cl_id	cl_name_last
1	H P D
2	H P
9	H4 P2

At this point you might wish to consider the tasks and determine the rows that should be returned **before** you write the query. It is very easy to assume the query is correct if it runs.

Demo 06: Question 3: We want clients who have a hedgehog and not a porcupine; we should get cl_ids 3,4,10.
The client needs to pass two tests: P2A and P2B.

```
select cl.cl_id, cl.cl_name_last
from tst_clients cl
where cl.cl_id in (
    select cl_id
    from tst_animals
    where an_type = 'hedgehog')
and cl_id not in (
    select cl_id
    from tst_animals
    where an_type = 'porcupine')
;
```

cl_id	cl_name_last
3	H D
4	H
10	H3

Demo 07: Question 4 is very much like question 2. Three predicates; three subqueries.

```
select cl_id, cl_name_last
from tst_clients
where
? ? ?
```

Demo 08: Question 5: We want clients who have (a hedgehog or a porcupine) and a dormouse. we should get cl_ids 1,3,5 Three predicates; three subqueries; take care with the order of precedence.

```
select cl_id, cl_name_last
from tst_clients
where
(
    cl_id in (
        select cl_id
        from tst_animals
        where an_type = 'hedgehog')
    OR cl_id in (
        select cl_id
        from tst_animals
        where an_type = 'porcupine')
)
and cl_id in (
    select cl_id
    from tst_animals
    where an_type = 'dormouse')
;
```

cl_id	cl_name_last
1	H_P_D
3	H_D
5	P_D

Demo 09: Question 6: We want clients who have (a hedgehog or a porcupine) and NOT a dormouse. We should get cl_ids 2, 4, 6, 9, 10 Three predicates; three subqueries; take care with the order of precedence.

```
select cl_id, cl_name_last
from tst_clients
where
(
    cl_id in (
        select cl_id
        from tst_animals
        where an_type = 'hedgehog')
    OR cl_id in (
        select cl_id
        from tst_animals
        where an_type = 'porcupine')
)
AND cl_id NOT in (
    select cl_id
    from tst_animals
    where an_type = 'dormouse')
;
```

cl_id	cl_name_last
2	H _____ P
4	H _____
6	P _____
9	H4_P2_____
10	H3_____

Demo 10: Question 7 We want clients who do not have a hedgehog. We should get cl_id.5, 6, 7, 8, 11

```
select cl_id, cl_name_last
from tst_clients
where
    cl_id NOT in (
        select cl_id
        from tst_animals
        where an_type = 'hedgehog');
```

cl_id	cl_name_last
5	P_D_____
6	P_____
7	D_____
8	_____
11	dog

Since I kept the attribute names and types matching those of the vt_clients and vt_animals tables, I should be able to simply adjust the table name to run against those tables.

4. Things you should NOT do for these patterns

It is always dangerous to show you queries that do not do the processing we want. Sometimes people just copy these as demos. These are **incorrect** logic for question 2.

The following gets clients with either a hedgehog or a porcupine. This is not a multi-match pattern. Using an In filter {where an_type IN ('hedgehog', 'porcupine')} doesn't help since an IN list is an OR test.

```
select distinct cl.cl_id, cl.cl_name_last
from tst_clients cl
join tst_animals an on cl.cl_id = an.cl_id
where an_type = 'hedgehog'
or an_type = 'porcupine'
order by cl_id;
```

cl_id	cl_name_last
1	H_P_D_____
2	H_P_____
3	H_D_____
4	H_____
5	P_D_____
6	P_____
9	H4_P2_____
10	H3_____

The following gets no rows since you are testing for an animal that is both a porcupine and a hedgehog at the same time. Each of our animals has only one value for an_type. The way this is written, it uses a row filter testing one row at a time.

```
select distinct cl.cl_id, cl.cl_name_last
from tst_clients cl
join tst_animals an on cl.cl_id = an.cl_id
where an_type = 'hedgehog'
and an_type = 'porcupine'
order by cl_id
;
no rows selected
```

The following gets no rows because it is also testing for an animal that has two types at the same time. Note that here the subquery test is on the an_id; it should be on the cl_id. We want clients who have a hedgehog and a porcupine, not animals that are a hedgehog and a porcupine.

```
select distinct cl.cl_id, cl.cl_name_last
from tst_clients cl
join tst_animals an on cl.cl_id = an.cl_id
where an_id in (
    select an_id
    from tst_animals
    where an_type = 'hedgehog')
and an_id in (
    select an_id
    from tst_animals
    where an_type = 'porcupine')
;
no rows selected
```

Table of Contents

1. Creating a Simple View.....	1
2. Using the View	1
3. A more complex view.....	2

We will look at views in more detail soon, but the assignment for this unit is easier with a view. A view is a select query that is given a name and the view definition is stored in the database. You can then use the view as a data source in a From clause. For the assignment you can just think of the view as a table expression and you can

- select columns from the view
- use filters against the columns in the view
- join the view to base tables using the same syntax you always use

The view we will use in the assignment and also in the subqueries demo contains a join of several tables and the purpose of the view is to hide that complexity from the queries we want to run.

1. Creating a Simple View

Demo 01: This is a very simple view that just exposes three columns for the products table for some of the rows. With Oracle you can use a "Create or Replace View" syntax

```
create or replace View prd_HW_APL
as (
  select
    prod_id
    , prod_name
    , prod_list_price
  from prd_products
  where catg_id in ('APL', 'HW')
);
```

2. Using the View

Demo 02: select data from the view

```
select *
from prd_HW_APL;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
1000	Hand Mixer	125
1070	Iron	25.5
1071	Iron	25.5
1072	Iron	25.5
1080	Cornpopper	25
1090	Gas grill	149.99
1100	Blender	49.99
1160	Mixer Deluxe	149.99
4569	Mini Dryer	349.95
4575	Electric can opener	49.95
1110	Pancake griddle	49.99
1120	Washer	549.99
1125	Dryer	500
1126	WasherDryer	850
1130	Mini Freezer	149.99

Demo 03: select data from the view with a filter

```
select *
from prd_HW_APL
where prod_list_price > 100;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
1000	Hand Mixer	125
1090	Gas grill	149.99
1160	Mixer Deluxe	149.99
4569	Mini Dryer	349.95
1120	Washer	549.99
1125	Dryer	500
1126	WasherDryer	850
1130	Mini Freezer	149.99

Demo 04: select data from the view with a join to a base table

```
select
    PR.prod_id
    , quantity_ordered * quoted_price as extPrice
    , order_id
from prd_HW_APL PR
join oe_OrderDetails OD On PR.prod_id = OD.prod_id
where order_id Between 110 and 115;;
```

PROD_ID	EXTPRICE	ORD_ID
1090	149.99	110
1130	149.99	110
1110	99.98	112
1080	22.5	113
1130	625	114
1000	200	115
1120	1900	115
1080	25	115
1100	180	115

3. A more complex view

The next view is more complex, including several joins and filters and renaming the column names

Demo 05: a more complex view

```
create or replace View oe_customer_orders
as (
    select
        OH.order_id as invoice
        , OH.order_date as orderDate
        , OH.customer_id as custID
        , PR.catg_id as category
        , OD.prod_id as itemPurchased
    from oe_orderHeaders OH
    join oe_OrderDetails OD On OH.order_id = OD.order_id
    join prd_products PR On OD.prod_id = PR.prod_id
    where OD.quoted_price > 0
    and OD.quantity_ordered > 0
);
```

Demo 06: When you use the second view, refer to the column names defined in the view. Note that if an order has more than one detail line, the view returns more than one row

```
select
    invoice
    , itemPurchased
from oe_customer_orders
where extract(month from orderDate) = 6;
```

INVOICE	ITEMPURCHASED
301	1100
302	1140
302	1040
303	1000
306	1120
306	1125
307	1120
307	1125
312	1040
312	1060
312	1060
312	1050
313	1000
324	4576
378	1125
378	1120
390	1010
395	1010
540	1110
540	1080
540	1152

Table of Contents

1.	Theoretical concepts	1
2.	SQL Concepts and Rules	3
3.	Some Simple Set operations	6
3.1.	Union	6
3.2.	Union All	7
3.3.	Intersect	7
3.4.	MINUS	Error! Bookmark not defined.
4.	Different ways to accomplish a task	8

We have seen a few ways to produce a result based on data from more than one table. With a join of two tables, each row in the result can have data from the two tables being joined. The set operators provide a different way to associate data from two tables. The set operators are UNION ALL, UNION, INTERSECT and EXCEPT. With a Union operator, each row in the result comes from one or the other of the two tables. The rows being "unioned" are returned combined into a single result.

1. Theoretical concepts

First we should consider the terms Set and Bag

A Set is a collection of data values, without any ordering and with no repeated values. A Bag (or MultiSet). is a collection of data values, without any ordering but it can have repeated values.

Demo

Suppose we have the following collections: (You could think of this as two children with their bug collections)

Collection_1 (ant, ant, ant, beetle, cricket, cricket)



Collection_2 (ant, cricket, earwig, flea, cricket)



These are multisets/bags since they have duplicates.

One way to combine these two collections is just to dump them all into one bag

Result_Collection_3 (ant, ant, ant, ant, beetle, cricket, cricket, cricket, cricket, earwig, flea)



Another way to combine these two collections is just to get one of each type and put them together

Result_Collection_4 (ant, beetle, cricket, earwig, flea)



Result_Collection_3 is called a Union All collection and Result_Collection_4 is a Union Distinct; Distinct suppresses duplicates.

We could make a collection of all values that are in both sets

Result_Collection_5 (ant, cricket)



That is called an Intersection.

But we could think of this in a slightly different way and come up with the following

Result_Collection_6 (ant, cricket, cricket)



because there are two crickets in Collection_1 and two crickets in Collection_2. This is also an Intersection.

We go to collection _1 and match ant (1) and ant(2), then we match 2-crickets(1) and 2 crickets(2)

We can classify Result_Collection_5 as an Intersection Distinct (no duplicates) and Result_Collection_6 as an Intersection All.

So far all of these operations have been commutative- that means that Collection_1 Union Collection_2 has the same meaning as Collection_2 Union Collection_1. Intersection is also commutative. There is another way to work with these collections and that is shown here

Result_Collection_7 (beetle) This is the values in Collection_1 that are not in Collection_2. This is not commutative.



Result_Collection_8 (earwig, flea) is the values in Collection_2 that are not in Collection_1. This uses the Except Distinct operator.



Result_Collection_9 (ant, ant, beetle) is the non-distinct set of values in Collection_1 that are not in Collection_2; this uses Except All.



Now consider what should happen if there are nulls in the original collections. (maybe the children have unidentified bugs.)

Collection_1v2 (ant, ant, ant, beetle, cricket, cricket, unknown-insect)

Collection_2v2 (ant, cricket, earwig, flea, cricket, unknown-insect, unknown-insect)

A Union Distinct result set contains only a single Null. Although we know that nulls are not equal to each other, for many situations SQL lumps the nulls together. A Union All results set contain a null for each null in one of the original collections.

If you have used Venn Diagrams before and found them helpful, you might want to look at the end of this document next and then come back to the SQL.

2. SQL Concepts and Rules

To combine the result sets of two Select statements with a set operator, the two result sets must be **union compatible**. The result sets

- must have the same number of attributes and
- must have the same (or convertible) data types for the corresponding columns.

Although it is not a syntax requirement that the corresponding columns have the same domain, you have the responsibility to make the output meaningful.

Demo 01: Suppose we wanted to get some data for the snakes and lizards from the vets animals table. We would write the query:

```
select cl_id, an_id, an_name, an_type
from vt_animals
where an_type in ('snake', 'lizard');
```

CL_ID	AN_ID	AN_NAME	AN_TYPE
411	15401	Pinkie	lizard
3561	17025	25	lizard
7152	17026	3P#_26	lizard
7152	17027	3P#_25	lizard
3561	21004	Gutsy	snake
1852	21007	Pop	snake

Demo 02: We could write this as a Union query. There is no particular advantage in using the Union here- but SQL often has more than one way to solve a task

```
select cl_id, an_id, an_name, an_type
from vt_animals
where an_type = 'snake'
UNION
select cl_id, an_id, an_name, an_type
from vt_animals
where an_type = 'lizard'
order by cl_id;
```

Demo 03: A common mistake in using a set operator is not have the same number of columns.

```
select cl_id, an_id, an_name, an_type
from vt_animals
where an_type = 'snake'
UNION
select cl_id, an_id, an_name
from vt_animals
where an_type = 'lizard'
```

```
order by cl_id;
```

```
SQL Error: ORA-01789: query block has incorrect number of result columns
```

Demo 04: The following will give us an error because an_type is a string (in the first part) and an_dob is a date value (in the second part)

```
select cl_id, an_id, an_name, an_type
from vt_animals
where an_type = 'snake'
UNION
select cl_id, an_id, an_name, an_dob
from vt_animals
where an_type = 'lizard'
order by cl_id;
```

```
SQL Error: ORA-01790: expression must have same datatype as corresponding expression
```

Demo 05: This is not a very sensible query but I now have two string columns, so this works. The output is not very meaningful to most people.

```
select cl_id, an_id, an_name, an_type
from vt_animals
where an_type = 'snake'
UNION
select cl_id, an_id, an_name , to_char(an_dob, 'YYYY-MM-DD')
from vt_animals
```

AN_TYPE	AN_ID	AN_NAME	CL_ID	ID
2010-03-15		Pinkie	15401	411
snake		Pop 22	21007	1852
2013-01-10		25	17025	3561
snake		Gutsy	21004	3561
2010-01-10		3P#_26	17026	7152
2010-01-10		3P#_25	17027	7152

6 rows selected

```
= 'lizard'
order by cl_id;
```

Suppose we have different numbers of columns to display in each part of the set query. We can always play tricks such as adding a literal column to one part of the union.

Suppose I had two tables (the blue table and the orange table) with some differences in the attributes.

ID	FirstName	MiddleName	LastName	DOB

ID	FirstName	LastName	Phone

If I want to combine these tables with the set operators, I have several choices.

- I could select only the columns found in both tables.

```
select ID, FirstName, LastName
from tblBlue
Union
select ID, FirstName, LastName
from tblOrange
```

ID	FirstName	LastName

ID	FirstName	LastName

- I could return a default value- probably nulls, for the columns found in one table or the other.

```
select ID, FirstName, MiddleName, LastName , DOB, null
from tblBlue
Union
select ID, FirstName, null, LastName , null, Phone
from tblOrange
```

ID	FirstName	MiddleNa	LastName	DOB	Phone
					null

ID	FirstName	MiddleNa	LastName	DOB	Phone
		null		null	
		null		null	
		null		null	
		null		null	
		null		null	
		null		null	

In the return table, the column headers and the Order By clause are based on the first Select columns and aliases. Oracle follows the standards in that the individual selects cannot be ordered- but the final result can be sorted.

If you have a query that uses more than one of these operators, the order of operations is top to bottom. You can use parentheses to change this order. Be careful with this- it is nonstandard and may change with future Oracle versions- use parentheses.

Demo 06: A somewhat more useful example of a Union query. Suppose we want the name of all the people the vet clinic deals with- both staff and clients. You can also see that the column aliases come from the first query.

```
select stf_name_last as LastName, stf_name_first as FirstName
, 'staff' as Position
from vt_staff
UNION
select cl_name_last, cl_name_first
, 'client'
from vt_clients
order by LastName, FirstName
```

Sample rows

LASTNAME	FIRSTNAME	POSITION
...		
Dolittle	Eliza	staff
Drake	Donald	client
Gordon	Dexter	staff
Halvers	Pat	staff
Harris	Eddie	client
Hawkins	Coleman	client
Helfen	Sandy	staff
Horn	Shirley	staff

UNION ALL- returns all of the rows from each of the queries.

UNION- returns all of the rows but removes any duplicates.

INTERSECT- returns rows that are part of both of the return sets for the component queries.

MINUS- returns rows that were returned by the first Select and not by the second. (The ansi standard term is Except; Oracle uses the keyword Minus)

Oracle does not implement Intersect All and Except All

3. Some Simple Set operations

In these demos we only want to get the client id for clients with different types of animals.

3.1. Union

Demo 07: This uses a Union; we get 4 rows returned. (In the earlier demos we got 6 rows for snakes and lizards)

```
select cl_id
from vt_animals
where an_type = 'snake'
UNION
select cl_id
from vt_animals
where an_type = 'lizard'
order by cl_id;
```

CL_ID
411
1852
3561
7152

3.2. Union All

Demo 08: the Union operator removes duplicate rows; the Union All operators leaves in duplicate rows.

Union All is more efficient if having duplicate rows is not a problem for the desired result.

```
select cl_id
from vt_animals
where an_type = 'snake'
UNION ALL
select cl_id
from vt_animals
where an_type = 'lizard'
order by cl_id;
```

CL_ID
411
1852
3561
3561
7152
7152

3.3. Intersect

Demo 09: The Intersect operator returns the rows that are in both query results. Here these are people with both a lizard and a snake. The intersect operator is commutative- we get the same result if we ask for people who have a snake and have a lizard as when we ask for people who have a lizard and have a snake.

```
select cl_id
from vt_animals
where an_type = 'snake'
INTERSECT
select cl_id
from vt_animals
where an_type = 'lizard'
order by cl_id;
```

CL_ID
3561

3.4. Minus

Demo 10: The Minus operator returns rows that are in one query result, but not in the other query result. This is no commutative. We get different results if we ask for people who have a snake and do not have a lizard then when we ask for people who have a lizard and do not have a snake.

Snake - no lizard

```
select cl_id
from vt_animals
where an_type = 'snake'
MINUS
select cl_id
from vt_animals
where an_type = 'lizard'
order by cl_id;
```

CL_ID

1852

Demo 11: Lizard- no snake

```
select cl_id
from vt_animals
where an_type = 'lizard'
MINUS
select cl_id
from vt_animals
where an_type = 'snake'
order by cl_id;
```

CL_ID

411
7152

4. Different ways to accomplish a task

Demo 12: A union is similar to an OR- clients who have a snake or a lizard

```
select Distinct cl_id
from vt_animals
where cl_id in (
    select cl_id
    from vt_animals
    where an_type = 'snake'
)
OR cl_id in (
    select cl_id
    from vt_animals
    where an_type = 'lizard'
)
order by cl_id;
```

CL_ID

411
1852
3561
7152

Demo 13: An intersect is similar to an AND- clients who have a snake and a lizard.

```
select Distinct cl_id
from vt_animals
where cl_id in (
    select cl_id
    from vt_animals
    where an_type = 'snake'
)
AND cl_id in (
    select cl_id
    from vt_animals
    where an_type = 'lizard'
)
```

```
order by cl_id;
CL_ID
-----
3561
```

Demo 14: The MINUS is similar to an IN and Not IN- clients who have a snake and do not have a lizard

```
select Distinct cl_id
from vt_animals
where cl_id in (
    select cl_id
    from vt_animals
    where an_type = 'snake'
)
AND cl_id NOT in (
    select cl_id
    from vt_animals
    where an_type = 'lizard'
)
order by cl_id;
```

```
CL_ID
-----
1852
```

You are not limited to one set operator. You can write queries to find who has a cat and a dog and a dormouse.
Who has a cat but not a bird and not a hedgehog.

This document has dealt with very simple set queries to establish the logic patterns. We will look at a few more complex queries in the next document.

Table of Contents

1. Demos for Housewares	1
2. Casting to handle syntax rules	3
3. Using multiple set operators.....	3

These queries will use set operators to look at which products were ordered in different months. To keep the output down to something we can understand more easily I will limit this to Houseware products(catg_id = 'HW') and to items purchased in the last three months of 2015.

1. Demos for Housewares

Demo 01: Query to show the relevant orders

```
select order_id, prod_id, catg_id, prod_name, order_date
from oe_orderHeaders OH
join oe_orderDetails OD using(order_id)
join prd_products PR using (prod_id)
where catg_id = 'HW'
and extract( month from order_date) in (10,11,12)
and extract(year from order_date) = 2015
order by order_date;
```

ORDER_ID	PROD_ID	CATG_ID	PROD_NAME	ORDER_DATE
108	1080	HW	Cornpopper	02-OCT-15
107	1110	HW	Pancake griddle	02-OCT-15
110	1090	HW	Gas grill	12-OCT-15
113	1080	HW	Cornpopper	08-NOV-15
112	1110	HW	Pancake griddle	08-NOV-15
115	1000	HW	Hand Mixer	08-NOV-15
115	1080	HW	Cornpopper	08-NOV-15
115	1100	HW	Blender	08-NOV-15
408	1071	HW	Iron	20-NOV-15
119	1070	HW	Iron	28-NOV-15
126	1100	HW	Blender	15-DEC-15
127	1110	HW	Pancake griddle	15-DEC-15
127	1080	HW	Cornpopper	15-DEC-15
127	1100	HW	Blender	15-DEC-15
130	1090	HW	Gas grill	30-DEC-15

15 rows selected

I could use this query as the basis for the rest of the demos but it is rather long; I could put it in a CTE and make the rest of the query simpler to read. In this case since I want to use it several time, I am going to create it as a view. I will also add a calculated column for the month since I will use that in the demos.

Demo 02: The view definition.

```
create view orderData as
select order_id, prod_id, catg_id, prod_name, order_date
, extract( month from order_date) as order_month
from oe_orderHeaders OH
join oe_orderDetails OD using(order_id)
join prd_products PR using (prod_id)
where catg_id = 'HW'
and extract( month from order_date) in (10,11,12)
and extract(year from order_date) = 2015
order by order_date;;
```

Demo 03: Union for items purchased in either November or December

```
select prod_id, catg_id, prod_name
from orderData
where order_month = 11
UNION
select prod_id, catg_id, prod_name
from orderData
where order_month = 12;
```

PROD_ID	CATG_ID	PROD_NAME
1000	HW	Hand Mixer
1070	HW	Iron
1071	HW	Iron
1080	HW	Cornpopper
1090	HW	Gas grill
1100	HW	Blender
1110	HW	Pancake griddle

Demo 04: Union for items purchased in both November and December

```
select prod_id, catg_id, prod_name
from orderData
where order_month = 11
INTERSECT
select prod_id, catg_id, prod_name
from orderData
where order_month = 12;
```

prod_id	catg_id	prod_name
1080	HW	Cornpopper
1100	HW	Blender
1110	HW	Pancake griddle

Demo 05: Orders which contained both a Blender(product 1100) and Pancake Griddle (product 1110)

```
select order_id, order_month
from orderData
where prod_id = 1100
INTERSECT
select order_id, order_month
from orderData
where prod_id = 1110;
```

ORDER_ID	ORDER_MONTH
127	12

Demo 06: Why do we get no rows if we also display the product id?

```
select order_id, order_month, prod_id
from orderData
where prod_id = 1100
INTERSECT
select order_id, order_month, prod_id
from orderData
where prod_id = 1110;
```

no rows selected

When we use a view (or a CTE) these queries become as simple as the ones in the previous document.

2. Casting to handle syntax rules

The rules for Set operations is that the various select sets must have the same number of columns and the columns must be type compatible. You can use casting functions to handle this.

Demo 07: Using a Union query to display two types of data. Note the use of the CAST function to make the first column union compatible.

The first part of the query gives us product id and prices and the second gives us a descriptive text and an average price.

```

select cast(prod_id as varchar(6) ) AS "Product ID"
, prod_list_price as "List Price"
from Product.products
where catg_id = 'APL'
UNION ALL
select
'---- avg Price for all Appliances ----'
, avg(prod_list_price)
from Product.products
where catg_id = 'APL'
;

```

Product ID	List Price
4569	349.95
1120	549.99
1125	500
1126	850
1130	149.99
---- avg Price for all Appliances ----	
	479.986

If you do not use the cast in the first select, you get an error message.

```
SQL Error: ORA-01790: expression must have same datatype as corresponding expression
```

The first select written without the cast -- select prod_id AS "Product ID" produces integer values and then Oracle tries to convert the literal in the second select to an integer. The first select statement sets the data types for the results columns.

3. Using multiple set operators

When you use multiple set operators you need to be concerned about the order of precedence for the operators. This is the same situation as when you use + and * in the same arithmetic expression- which operator is done first.

The rule for Oracle is that the operators are executed in the order in which they appear (top to bottom).

This is not standard ANSI, and Oracle says this might be changed in the future. Use parentheses to change the default order.

Here are a few examples. There are not many rows in the view- compare these row by row.

Demo 08: Items orders in Nov but not in Oct and not in Dec

```
select prod_id, catg_id, prod_name
from orderData
where order_month = 11
minus
select prod_id, catg_id, prod_name
from orderData
where order_month = 12
minus
select prod_id, catg_id, prod_name
from orderData
where order_month = 10;
```

PROD_ID	CATG_ID	PROD_NAME
1000	HW	Hand Mixer
1070	HW	Iron
1071	HW	Iron

Demo 09: Items order in Nov but (not in both Oct and Dec)

```
select prod_id, catg_id, prod_name
from orderData
where order_month = 11
minus
(
    select prod_id, catg_id, prod_name
    from orderData
    where order_month = 12
    intersect
    select prod_id, catg_id, prod_name
    from orderData
    where order_month = 10
)
```

prod_id	catg_id	prod_name
1000	HW	Hand Mixer
1070	HW	Iron
1071	HW	Iron
1100	HW	Blender

(4 row(s) affected)

Demo 10: items that were ordered in Nov; take out the items order in Dec; add in the items order in Oct.

```
select prod_id, catg_id, prod_name
from orderData
where order_month = 11
minus
select prod_id, catg_id, prod_name
from orderData
where order_month = 12
union
select prod_id, catg_id, prod_name
from orderData
where order_month = 10
;
```

prod_id	catg_id	prod_name
1000	HW	Hand Mixer
1070	HW	Iron
1071	HW	Iron
1080	HW	Cornpopper
1090	HW	Gas grill
1110	HW	Pancake griddle

(6 row(s) affected)

Demo 11: Why is the result the same as one of the previous demos in this set?

```
select prod_id, catg_id, prod_name
from orderData
where order_month = 11
minus
(
  select prod_id, catg_id, prod_name
  from orderData
  where order_month = 12
  union
  select prod_id, catg_id, prod_name
  from orderData
  where order_month = 10
);
```

prod_id	catg_id	prod_name
1000	HW	Hand Mixer
1070	HW	Iron
1071	HW	Iron

(3 row(s) affected)

Table of Contents

1. Classification of subqueries.....	1
1.1. Classification of subqueries by usage.....	1
1.2. Classification of subqueries by return table shape	1
1.3. Classification as correlated or non correlated	2

We have done a bit of work with subqueries in the Where clause so some of this is a review.

The purpose of using a subquery is to let the subquery determine values from the database that can then be used by another query to get the final desired result. We want to automate this as a single query rather than running two separate queries and manually transferring the answer from one query to the second.

Subqueries are also called nested queries. There are a variety of places where you can use subqueries.

Subqueries can return different shapes of result tables. And subqueries can be a self contained query or rely on the outer query for processing. So first we will talk about the various types of subqueries and then go on to the examples.

SQL is a redundant language; sometimes there is more than one way to accomplish a task. Subqueries can be used instead of joins for some queries. In the past, subqueries were considered to be more efficient than joins. But with the current database engines, joins may be just as, or more, efficient. In many queries, joins can be easier to understand than subqueries.

This unit will include some of the subquery type but not all of them.

1. Classification of subqueries

There are several overlapping ways to classify subqueries.

1.1. Classification of subqueries by usage

A subquery is a Select query that is placed inside another query. We use the terms inner and outer query for this; we also use the terms parent query and child query. Queries can be nested several layers deep.

You can nest a subquery within the

- Where clause of the parent query
- Select clause of the parent query
- From clause of the parent query
- Having clause of the parent
- Where clause of Update, Insert, and Delete statements; we have seen examples of this in the section on Action queries.

1.2. Classification of subqueries by return table shape

Subqueries can be classified by their return table:

- scalar- return a single value (one row and one column) This is still a table.



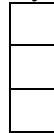
You can test a scalar subquery with an equality test and you can use a scalar query in places where a single value is allowed.

- Row subquery -return a multi-column, single row table



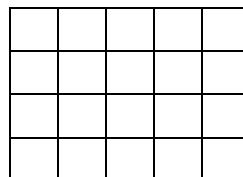
This is a subquery that can return multiple columns but only one row. In some dbms this can be used to test a set of columns against a set of values.

- multi-row, single column- return a single column and possibly multiple rows



If you have a subquery that can return multiple rows with a single column, then you have to use the subquery with a test where multiple rows are acceptable. You often use these subqueries in a filter clause of a query. You cannot test this return set with the equality (=) operator but you can use the IN list operator. It makes sense to ask if a column value is IN the data returned by a multi-row subquery.

- multi-column-return a multi-column, multi-row table



This is a subquery that can return multiple columns and multiple rows- so it is essentially returning a virtual table that could be used as a table expression in the From clause. In this case the subquery is sometimes called an in-line view. The subquery does not have to be the only table expression in the From clause; you can join the subquery result to a regular table to do a join.

1.3. Classification as correlated or non correlated

Subqueries can be classified as:

- non-correlated
- correlated

A non-correlated subquery can "stand by itself". When you think about a non-correlated query, you should think of the inner query being evaluated and returning a value, possibly a table value, which is then given to the parent query.

With a correlated query, the child query cannot stand on its own. The child query refers to columns in the parent query. With a correlated query, you should think of the parent query operating on its first row, then evaluating the child query; then the parent query works on its next row and re-evaluates the child query. This means that a correlated subquery can be very inefficient; you should use other techniques if possible.

Table of Contents

1. Intro.....	1
2. Nesting subqueries.....	1
3. Testing with equality tests	3
3.1. Guaranteeing a scalar result	4
3.2. Subquery that returns no rows	4
4. Testing with the In List filter.....	5
5. Subqueries versus Joins.....	7
6. Subqueries for unmatched rows.....	7

1. Intro

Much of this will be review but review is good as we are going into more complex subqueries. This discussion will also discuss techniques that people often have difficulties with as we use subqueries. We are focusing on the use of subqueries in the Where clause of a Selects query.

Terms: I will use the term **main query** for the **top level query**. In the following query(demo 01) the Select . . . From cust_customers is the main query; this query determines the columns that can be exposed by the query.

The query `Select customer_id From oe_orderHeaders` is the **subquery**.

Demo 01: A simple subquery

```
select customer_name_last
from cust_customers
where customer_id in (
    select customer_id
    from oe_orderHeaders
);
```

CUSTOMER_NAME_LAST

```
-----
McGold
Morse
Northrep
Morise
Williams
Otis
. . .
```

2. Nesting subqueries

For many of the demos I will use a single level of subquery. But you are not limited to a single level. You can nest subqueries.

Demo 02: This uses two subqueries to display customers who ordered a specific product- product id 1020

```
select customer_name_last
, customer_name_first
, customer_id
from cust_customers
where customer_id IN (
    select customer_id
    from oe_orderHeaders
    where order_id IN (
        select order_id
        from oe_orderDetails
        where prod_id = 1020) )
```

```
order by customer_id;
```

CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST	CUSTOMER_ID
Northrep	William	401890
Williams	Sally	403000
Button	D. K.	404100
Williams	Al	404900
Clay	Clem	408770
Martin	Joan	409150

Let's take this query step by step. With the subqueries we are using now, we can start with the inner most query. This query uses the order details table and gets all of the order id for orders for a particular product

```
select order_id
from oe_orderDetails
where prod_id = 1020;
```

These are the order id values for that product.

ORDER_ID
414
519
520
716
529
605
608
105
405
3808
3516

When we use this as a subquery with an IN list test, this output will be used as a list: (519, 520, 716, 529, 414, 605, 608, 105, 405, 3808, 3516).

So this is effectively what the query is doing.

```
select customer_id
from oe_orderHeaders
where order_id IN (519,520,716, 529,414,605,608,105,405,3808,3516)
```

This is the two level subquery. Note that the subquery is enclosed within parentheses.

```
select customer_id
from oe_order_headers
where order_id IN (
    select order_id
    from oe_order_details
    where prod_id = 1020) ;
```

CUSTOMER_ID
403000
408770
409150
409150
401890
404900
403000
404100
403000
409150
403000

The result is again used as an IN list (403000, 408770, 409150, 409150, 401890, 404900, 403000, 404100, 403000, 409150, 403000). Note that the 2-level subquery has the customer id 403000 appearing several times in the result. This is not a problem. Some people use a Distinct in the subquery but that is not needed and in some systems can make the query less efficient since it would need to do extra work to remove duplicates.

Now we are up to the top level of the original query which is effectively doing

```
select customer_name_last
, customer_name_first
, customer_id
from cust_customers
where customer_id IN (403000, 408770, 409150, 409150, 401890, 404900, 403000,
404100, 403000, 409150, 403000);
```

You probably have realized that we could also do this as a three table join.

Demo 03: This is a three table join and we need distinct to remove duplicates. We did not need distinct in the subquery since the top level table expression is just the customer table and each customer appears only once in that table.

```
select Distinct
  CS.customer_name_last
, CS.customer_name_first
, customer_id
from cust_customers CS
join oe_orderHeaders OH using(customer_id)
join oe_orderDetails OD using(order_id)
where OD.prod_id = 1020
order by customer_id;
```

Demo 04: Find customers who have ordered any appliance - using the category id APL

This is in the demo- but try to figure it out for yourself first. Reading answers is not as helpful as discovering solutions.

3. Testing with equality tests

If the subquery is guaranteed to return a single row and a single column, then we can test the subquery result with the equality operators (and with the operators \neq , \geq , $>$, \leq , $<$).

We sometimes call this a scalar subquery; but we know that every result set is a table- what we are saying is that this subquery will return a table with a single row and a single column.

We can ensure a single column by having only one column expression in the Select clause of the subquery.

How do we guarantee that the subquery returns exactly one row? We can filter the subquery on a column that is declared as Unique in its table- often this will be the PK column(s). Sometimes we see examples of queries that return only one row for the current set of data- but that is not enough for robust code. We need to use a subquery that will **always** return a single row of data for any set of rows in the table.

Demo 05: We can use a subquery with `customer_id =` since the inner query returns one row and one column. This is filtering on the PK of the order headers table so we know we get one customer id.

```
select customer_id, customer_name_last, customer_name_first
from cust_customers
where customer_id = (
```

```
select customer_id
from oe_orderHeaders
where order_id = 115
);

```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
402100	Morise	William

We know that in the order table, the order_id is the pk. So we can have only one row in the order table with order_id 115. The subquery will return the single customer_id for that order. We can use that returned value in the Where clause of the parent query to find that customer's information in the customer table.

3.1. Guaranteeing a scalar result

This is an area where there is some confusion, particularly in assignments where your subquery runs and gives what seems to be a correct answer but the query is still incorrect.

Suppose we want to find all customers who ordered a particular product. We decide to set a variable for the product id. Then we are going to use a subquery and do a join of two tables in the subquery.

Demo 06: This runs and produces a single row of output with my current set of data

```
Variable prod_id number;
exec :prod_id := 5004;

select customer_id, customer_name_last, customer_name_first
from cust_customers
where customer_id = (
    select customer_id
    from oe_orderHeaders OH
    join oe_orderDetails OD on OH.order_id = OD.order_id
    where prod_id = :prod_id
);

```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
403100	Stevenson	James

```
exec :prod_id := 1080;
/
```

But if you change the value of the variable to 1080 and rerun the query you get an error.

ORA-01427: single-row subquery returns more than one row
--

Why does this happen? With the product id 5004 we have only one order so the subquery returns one customer id. With the product id 1080 we have several orders so the subquery returns several customer ids. That causes the test customer_id = filter to crash. The query is not logically correct since it assumes a single order and that assumption is not supported by the table design.

3.2. Subquery that returns no rows

Suppose we want to ask the following question about employees and managers. We want a list of all of the employees who have the same manager as a specific employee.

First we will use a variable for the employee id we are using. The subquery gets the manager ID for that employee. This subquery returns only a single value since one employee has at most one manager in the

definition of our tables. That id is passed up to the outer query which lists all people managed by that person and skips the employee with the originally specified id.

Demo 07: Who is managed by the same person who manages employee 145?

```
variable empId number;
execute :empId := 145;

select emp_id
from emp_employees
where emp_id <> :empId
and emp_mng = (
    select emp_mng
    from emp_employees
    where emp_id = :empId )
;

EMP_ID
-----
201
101
102
146
```

Demo 08: Now we change the employee ID and we do not get any rows returned. We do have an employee 100 but he does not have a manager.

```
execute :empId := 100;
/
no rows selected
```

Demo 09: This ID value also does not return any rows. We do not have an employee with ID 408

```
execute :empId := 408;
/
no rows selected
```

We cannot distinguish these two conditions from the output.

4. Testing with the In List filter

Suppose we try to run the following query. The subquery can return multiple rows and the query fails. So we should not test this with an equals test, but we can use an IN list test. We are not saying that the subquery must return multiple rows. We cannot determine that by looking at the query. It is possible that our current set of data has no orders dated 2015-10-01; there could be exactly one such order, or there could be many such orders and the subquery returns multiple rows. We need to write queries that work with any valid set of data in the table—not with a particular collection of rows.

Demo 10: This query fails

```
variable dtm varchar2(11)
exec :dtm := '01-Oct-2015';

select customer_id, customer_name_last, customer_name_first
from cust_customers
where customer_id = (
    select customer_id
    from oe_orderHeaders
```

```

    where order_date = to_date(:dtm)
)
;

```

If you run a query with a subquery that returns multiple rows and you use an equality test, you get an error message.

ORA-01427: single-row subquery returns more than one row
--

Demo 11: Change this to an IN subquery test

```

exec :dtm := '01-Oct-2015'
select
  customer_id
, customer_name_last
, customer_name_first
from cust_customers
where customer_id In (
  select customer_id
  from oe_orderHeaders
  where order_date = to_date(:dtm)
);

```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
403000	Williams	Sally
404950	Morris	William

Demo 12: Suppose we run the following query with a date that does not match any orders; then we do not get any rows in the result set. Remember this is not an error- we simply do not have any such data.

```

exec :dtm := '08-AUG-1888';
/

```

no rows selected

Demo 13: If we change the query to a NOT IN query, then all customers are returned with our data set

```

select customer_id, customer_name_last, customer_name_first
from cust_customers
where customer_id NOT IN (
  select customer_id
  from oe_orderHeaders
  where order_date = to_date(:dtm)
);

```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
400801	Washington	Geo
401250	Morse	Samuel
401890	Northrep	William
402100	Morise	William
402110	Coltrane	John
402120	McCoy	Tyner
402500	Jones	Elton John
. . .	rows omitted	

5. Subqueries versus Joins

The next two queries both filter for orders in Dec 2015- one returns 8 rows and the other query returns 5. You need to know the business needs for the query to determine which is the correct query.

Demo 14: This does not use a subquery. It uses a join. Depending on our needs ,we might want to include a Distinct in the select clause.

```
select customer_id
, customer_name_last, customer_name_first
from cust_customers
join oe_orderHeaders using(customer_id)
where extract ( month from order_date) = 12
and extract( year from order_date) = 2015
order by customer_id;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
409030	Mazur	Barry
409030	Mazur	Barry
409150	Martin	Joan
409160	Martin	Jane
409160	Martin	Jane
409190	Prince	
915001	Adams	Abigail
915001	Adams	Abigail

8 rows selected

Demo 15: This uses a subquery. Use customer_id IN since the inner query can return multiple rows. This will bring back only one row per customer even if they have multiple orders in Dec 2014. This query cannot show the order_date since it is not a column in the parent (top level) query.

```
select customer_id, customer_name_last, customer_name_first
from cust_customers
where customer_id IN (
    select customer_id
    from oe_orderHeaders
    where extract( month from order_date) = 12 and
        extract( year from order_date) = 2015);
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
915001	Adams	Abigail
409030	Mazur	Barry
409150	Martin	Joan
409160	Martin	Jane
409190	Prince	

6. Subqueries for unmatched rows

We used outer joins and tested for null values to find unmatched rows- such as customers with no orders. We can also do this with subqueries.

Many people find the Not In subquery approach easier to understand than the outer join.

Demo 16: This is an outer join query that returns customers who have no orders.

```
select CS.customer_id, CS.customer_name_last
from cust_customers CS
left join oe_orderHeaders OH on CS.customer_id = OH.customer_id
```

```
where OH.order_id is null;
CUSTOMER_ID CUSTOMER_NAME_LAST
-----
400801 Washington
402110 Coltrane
402120 McCoy
402500 Jones
. . . rows omitted
```

Demo 17: This is a subquery to accomplish the same task.

```
select customer_id, customer_name_last
from cust_customers
where customer_id not in (
    select customer_id
    from oe_orderHeaders
);
```

Demo 18: Do we have any order rows for which there are no associated order detail rows?

```
select order_id, order_date
from oe_orderHeaders
where order_id not in (
    select order_id
    from oe_orderDetails
)
;

----- ORDER_ID ORDER_DATE
----- -----
116 12-NOV-15
123 05-DEC-15
2506 12-JAN-16
```

Table of Contents

1. The Max and Min functions	1
2. The Sum function	2
3. The Avg function	2
4. What about nulls?	3
5. Using aggregates and a Where clause and a Join	4
6. Aggregate functions and non aggregated columns	5
6.1. Find the winner queries	5
7. Stats_mode and Median	7
8. Mistakes you should not make	9

So far we have been using Select statements to return result sets that display individual rows from our tables. We have included filters so that we return only some of the rows but the result set were always focused on individual rows from the tables. The Where clauses filtered on one row at a time based on the table expression produced by the From clause.

But think about what companies such as Amazon need to know to do business. They do need to know detail information so that they can charge me for the one book that I purchased, but they also need to know about large groups of data- how many people purchased that book? How many books do we carry that no one has purchased in the last 6 months? What is the average amount due for our orders? What was the total sales for last month?

These type of questions ask for **summary information** about our data. SQL provides **aggregate functions** to answer these questions. The aggregate functions are also called multi-row functions or group functions because they return a single value for each group of multiple rows.

The SQL aggregate functions we cover in this class are Max, Min, Sum, Avg, Median, and Stats_Mod ,and Count. Oracle has additional aggregate functions.

In this unit we will discuss using aggregate functions applied to the entire table expression as filtered by the Where clause. We could find the highest price of all of our products; we could filter for pet supplies and find the highest price for pet supplies product.

1. The Max and Min functions

We will start with some examples where we consider the entire table expression collection of rows as a single group. We use the Max function to get the largest value for a column in the table and the Min function to get the smallest value. These queries are using the tables associated with the AltgeldMart database.

Demo 01: max and min order id from the order headers table

```
select max(order_id) as Max_OrderID
, min(order_id) as Min_OrderID
from oe_orderHeaders
;
```

MAX_ORDERID	MIN_ORDERID
4511	105

First notice that we get one row returned. We have almost 100 rows in the table, but we are applying the max and min functions to the entire table. The order_id column is numeric so this is just the biggest number and the smallest number for the order id values.

Demo 02: We can use Max, and Min with numeric data, with dates, and with character data. With character data, max and min use the sorting order for the data.

```
select max(order_date) as Max_OrderDate
, min(order_date) as Min_OrderDate
, max(order_mode) as Max_OrderMode
, min(order_mode) as Min_OrderMode
from oe_orderHeaders;
```

MAX_ORDERDATE	MIN_ORDERDATE	MAX_ORDERMODE	MIN_ORDERMODE
12-MAY-16	05-APR-15	ONLINE	DIRECT

2. The Sum function

The Sum function adds the values in the table column. Sum is used with numeric data.

Demo 03: getting the total of the quantities sold

```
select sum(quantity_ordered) as SumQuantity
from oe_orderDetails;
```

SUMQUANTITY
950

Demo 04: getting the total amount due for all orders in the table.

```
select sum(quantity_ordered * quoted_price) as totalAmountDue
from oe_orderDetails;
```

TOTALAMOUNTDUE
84989.98

Demo 05: What happens if you try to use Sum with a date type or a character type column? You get an error message that you cannot use Sum with those data types.

```
select sum(ord_mode)
from oe_order_headers;
```

ORA-01722: invalid number

```
select sum(ord_date)
from oe_order_headers;
```

ORA-00932: inconsistent datatypes: expected NUMBER got DATE

3. The Avg function

The average function returns the average(mean) of the values. Avg is used with numeric data.

Demo 06: Examples of the avg function

```
select Avg(quantity_ordered) as AvgQuantity
, Avg(quantity_ordered * quoted_price) as AvgAmountDue
from oe_orderDetails;
```

AVGQUANTITY	AVGAMOUNTDUE
5.16304348	461.902065

4. What about nulls?

We always need to consider nulls.

Demo 07: This is a small table I created for testing.

```
create table z_tst_aggr(
    id      integer primary key
    , col_int integer null
);
```

Demo 08: If I do these aggregates on an empty table, I get nulls. But I do get a row of nulls.

```
select max(col_int) as theMax, sum(col_int) as theSum, avg(col_int) as theAvg
from z_tst_aggr;
```

THEMAX	THESUM	THEAVG

1 row selected

Now I insert 8 rows. most of these have a null in the second column.

ID	COL_INT
1	10
2	15
3	
4	
5	
6	
7	
8	

Run the aggregates again.

```
select max(col_int) as theMax, sum(col_int) as theSum, avg(col_int) as theAvg
from z_tst_aggr;
```

THEMAX	THESUM	THEAVG
15	25	12.5

What these functions do is ignore nulls and just work with the non-null data values.

You would have a logical point if you argued that these function should return nulls; in this case we have 8 rows, 6 of the rows have unknown values and the average function simply returns the average of two rows. So you have to think of these functions as returning the Max, Sum, Avg of the values they know.

Some people will argue that the average function should take the sum of the values (25) and divide it by the number of rows (8) and the avg would then be about 3. But this means that you are treating the unknown values as zero- which may not be a good business rule.

You can code the avg function call to specifically treat the nulls as zero.

```
select max(col_int) as theMax, sum(col_int) as theSum
, avg(coalesce(col_int,0)) as theAvg
from z_tst_aggr;
```

THEMAX	THESUM	THEAVG
15	25	3.125

Don't get mad at the average function and do not assume all unknown numeric values are zero.

Suppose we are using the avg function to find the average salary for all of our employees and some rows have a null for salary. Does the result returned by avg make business sense? This depends on the situation. Suppose we have 40,000 rows in our table and every row except for two have a value for salary. In that case you might decide that it is safe to ignore the fact that we are missing two rows of data and that the value returned by avg makes sense. SQL cannot help you with that decision; you can use sql to find out what percent of rows are missing data; you can use sql to find the average of the known values. But if those two rows were the rows for the two executives in the company who are hiding their salary values because they are paid so very much more than everyone else, then the value returned by avg does not reflect reality.

On the other hand if we have 40,000 rows in our table and 30,000 of those rows are missing a value for salary, then I don't think I would put much trust that the value returned by avg reflects reality.

Business decision always need to be made by the business experts- not the sql coder. The sql person can help supply info to help the business experts make decisions.

5. Using aggregates and a Where clause and a Join

Demo 09: Using aggregates and a criterion. What is the average list price of the houseware items we carry?

```
select Avg (prod_list_price) as "AvgPrice"
from prd_products
where catg_id = 'HW';
-----  
AvgPrice  
-----  
67.641
```

If we change the filter to CEQ we get a null because we do not have any rows with that category; the filtered table expression in the From and Where clause is empty. But we do get a row- it is just a row with a null.

Demo 10: Using aggregates and a join. For the houseware items we have on an existing order, what is the average list price, the average quoted price and the average extended cost?

```
select
  Avg (prod_list_price) as "AvgListPrice"
, Avg (quoted_price) as "AvgQuotedPrice"
, Avg (quoted_price* quantity_ordered) as "AvgExtendedCost"
, Sum (quoted_price* quantity_ordered) as "TotalExtendedCost"
from prd_products
join oe_orderDetails on prd_products.prod_id = oe_orderDetails.prod_id
where catg_id = 'HW'
;  
-----  
AvgListPrice AvgQuotedPrice AvgExtendedCost TotalExtendedCost  
-----  
54.302037 54.0988889 154.728889 8355.36
```

Note that for each of these queries, we have used aggregate functions in the Select clause- and only aggregate functions in the Select clause and we have one row in the result set. An aggregate function is different than a single row function. An aggregate function takes a group (here the whole table) and produces a **single answer** for that group. We can include a Where clause which filters the table produced by the From clause and that filtered set of rows becomes the group. The Where clause is carried out before the aggregates are calculated in the Select.

Demo 11: You can do some additional work in the Select clause such as applying a function such as round to the aggregated value or combining aggregated values.

```
select round(Avg(prod_list_price),0) as "Avg Price"
, Max(prod_list_price) - Min(prod_list_price) as "Price Range"
from prd_products;
```

6. Aggregate functions and non aggregated columns

You might want to see the name of the most expensive item we sell- but this type of query cannot tell you that. A function, including an aggregate function, returns a single value for its arguments. We could have two or more items selling at that high price. The query is written to return one row for all of the items grouped together and so it cannot show the product id.

Demo 12: What is the highest list price for any item we sell?

```
select Max(prod_list_price) as "Max Price"
from prd_products
```

;

Max Price

850

It is possible to display a literal and an aggregate function.

```
select sysdate as RunDate
, MAX(prod_list_price) As "Max Price"
from prd_products;
```

RUNDATE	Max Price
-----	-----
19_MAR-16	850

6.1. Find the winner queries

Demo 13: What is the highest price for any item we sell and what is the item- this one does not work and we get an error message.

```
select prod_id, Max(prod_list_price) as "Max Price"
from prd_products
;
```

, prod_id
ORA-00937: not a single-group group function

This is an important feature of the way most dbms implement the aggregate functions. We are considering the table as a whole and we can ask for the highest value we have for the list price- that is a characteristic of the table as a whole- but we cannot ask for a prod_id since each row (each product) has its own value for that column and there is no single prod_id that represent the entire table.

The following does not work to find the most expensive product. You are not allowed to use an aggregate function in a Where clause this way. It looks like it should work- after all you can determine the Max (product list price) and that is a single value but the Where clause is for single row filters and aggregates work on groups.

Demo 14: THIS DOES NOT WORK

```
select prod_id
, prod_name
, prod_list_price
from prd_products
where prod_list_price = MAX(prod_list_price)
where prod_list_price = MAX(prod_list_price)
      *
```

ERROR at line 5:
ORA-00934: group function is not allowed here

But we have done some simple subqueries that we can use to handle this.

Demo 15: Using a subquery; the subquery returns the amount of the highest price and the outer query displays all items with that price.

With the current data set there is only one match.

```
select prod_id
, prod_name
, prod_list_price
from prd_products
where prod_list_price = (
    select MAX(prod_list_price) as "Largest Price"
    from prd_products);
```

prod_id	prod_name	prod_list_price
1126	WasherDryer	850.00

Demo 16: Suppose we want to find the highest priced sporting goods items. This runs but the result is incorrect. The mini dryer is not a sporting goods item

```
select prod_id
, prod_name
from prd_products
where prod_list_price = (
    select MAX(prod_list_price) as "Largest Price"
    from prd_products
    where catg_id = 'SPG');
```

prod_id	prod_name
1040	Treadmill
4569	Mini Dryer

This query gives us two items- but if we check, one of them is actually an appliance item. So we need to filter in the outer query for category as well.

Demo 17: Suppose we want to find the highest priced sporting goods items. Corrected query.

```
variable catg_id varchar2(3);
exec :catg_id := 'SPG';

select prod_id
, prod_name
from prd_products
where catg_id = :catg_id
and prod_list_price = (
    select MAX(prod_list_price)
    from prd_products
    where catg_id = :catg_id);
```

prod_id	prod_name
1040	Treadmill

Demo 18: If we change the variable to PET and run the same query, we get more than one item tied for the most expensive in that category.

```
exec :catg_id := 'PET';
/
```

prod_id	prod_name
4567	Deluxe Cat Tree
4568	Deluxe Cat Bed

7. Stats_mode and Median

These are two aggregate functions that Oracle implements.

The median is the middle value of a set of numbers or dates or strings and the stats_mode is the most frequent value. We will start with a small test table. Both of these function ignore nulls, so I do not have any nulls in the test data.

```
Select *
from z_tst_median_mode
order by id;
```

ID	COL_INT	COL_STR
1	10	red
2	12	blue
3	10	blue
4	25	red
5	50	red
6	25	orange
7	7	blue

The queries will specify which rows to include in the set of values.

Stats_mode. The mode is the value that occurs most frequently

Demo 19: Mode of the first 5 rows. We have three col_str values for red, so that is the most frequent. In the col_int column in the first 5 rows, 10 is the most frequent value.

```
With dataSource as (
  select col_str, col_int
  from z_tst_median_mode
  where id between 1 and 5
)
select Stats_Mode(col_str) as "MODE", Stats_Mode(col_int) as "MODE"
from dataSource;
```

MODE	MODE
red	10

Demo 20: Change the cte to use where id between 1 and 7. With 7 rows the values 'red' and 'blue' each occur 3 times and the numbers 10 and 25 each occur 2 times. The rule is that if there is more than one mode, then Oracle picks one of the modes and returns that value. This means that stats_mode is a non deterministic function- you cannot guarantee the result even with the same set of table data. It would be equally valid for Oracle to return red or 25.

MODE	MODE
blue	10

Median: this function takes a set of numeric or datetime values. You can think of this as putting the values in sorted order and picking the value in the middle

Demo 21: getting the median of the first 5 numbers. Those values are (10, 10, 12, 25, 50)

```
With dataSource as (
    select col_int
    from z_tst_median_mode
    where id between 1 and 5
)
select Median(col_int) as "MEDIAN"
from dataSource;
```

MEDIAN

12

Demo 22: getting the median of the first 6 numbers. Those values are (10, 10, 12, 25, 25, 50) Now we have 2 values in the middle 12, 25; median returns the average of these two middle values.

MEDIAN

18.5

This is the Oracle technique for finding multiple modes. You should be able to read through this SQL and understand it.

```
select col_str
from (
    select col_str, count(col_str) as cnt1
    from z_tst_median_mode
    group by col_str
)
where cnt1 = (
    select max(cnt2)
    from (
        select count(col_str) as cnt2
        from z_tst_median_mode
        group by col_str
    )
);
COL_STR
-----
red
blue
```

Demo 23: Which of our products was ordered the most often? **This does not find ties.** It uses the order details table because that contains the product id of the products that were ordered.

```
select stats_mode (prod_id)
from oe_orderDetails
```

STATS_MODE (PROD_ID)

1080

Demo 24: Which customers ordered that "most ordered product"?

```
select cust_id, cust_name_last
from cust_customers
where cust_id in (
    select cust_id
    from oe_orderHeaders
    join oe_orderDetails using (ord_id)
    where prod_id = (
        select stats_mode (prod_id)
        from oe_orderDetails
    )
)
;
CUST_ID CUST_NAME_LAST
-----
401250 Morse
402100 Morise
403000 Williams
404100 Button
404950 Morris
409150 Martin
900300 McGold
915001 Adams
```

Working through that query; from the innermost subquery out to the main.

This finds the product id for the most frequently ordered item.

```
select stats_mode (prod_id)
from oe_orderDetails
```

This find the customer ids for customers who ordered that product

```
select cust_id
from oe_order_headers
join oe_orderDetails using (ord_id)
where prod_id = ( . . . )
```

The main query finds the customers with id in that list.

```
select cust_id, cust_name_last
from cust_customers
where cust_id in ( . . . )
```

8. Mistakes you should not make

1. Some people think that if a query runs and produces results, the query is in some sense ok. This is not true. For example, the function Sum will add up numeric values. So you could run this query and get a result. But the result has no meaning in a business sense. There is no point to adding up customer id numbers.

```
select sum(customer_id)
from cust_customers;
SUM(CUST_ID)
-----
15262819
```

You are responsible to writing queries that are meaningful.

2. The Where clause cannot contain an aggregate function. The Where clause filters on single rows; the aggregates work with groups of rows. See the Having clause in the next unit.

4. These functions take an argument that is either a column in the table, or an expression based on a column. So you can write `Select Max(prod_list_price) as "Max Price" from prd_products;` But you cannot write `Select Max(45,90) from dual;` to have SQL find the larger value.

```
Select Max(45,90) from dual
*
ERROR at line 1:
ORA-00909: invalid number of arguments
```

Oracle does have a Greatest function- which accepts a string of values. It is not the same as the Max function which accepts column expressions.

```
Select greatest(45,90) from dual;
```

GREATEST(45,90)

90

Table of Contents

1. Count(*).....	1
2. Count(expression)	1
3. Count(Distinct expression).....	2
4. Count with Inner and with Outer Joins	5
5. Summary	6

I am putting this discussion of Count in a separate document because Count has some useful features different than Max, Min, Sum, and Avg.

1. Count(*)

The expression count(*) will return the number of rows in the table expression. If the table is empty- has no rows- count(*) will return 0. If there are rows, count(*) will tell you how many rows are in the table.

Demo 01: Number of rows in the customer table

```
Select COUNT(*) As "Number of customers"
From customer_customers;
```

2. Count(expression)

Count(expression) will determine the number of rows in the table expression where that expression is not null.

Demo 02: Using the Count aggregate function with a column attribute.

At the time this query was run, we had 34 customers- all customers have a customer_id value since that is the primary key for this table; 33 of those customers have a value for first name.

```
select
    COUNT(customer_id) As "Number of customers"
    , COUNT(customer_name_first) As "Number with a first name"
from customer_customers;
```

Number of customers	Number with a first name
34	33

If the table is empty, or if all of the rows have a null for the expression, count(expression) will return 0.

Remember that Max, Min, Sum, and Avg returned a null in that case. If the table expression is empty, we know how many rows it has- 0.

Demo 03: Currently the largest salary value in the employee table is 120000. If I run the following query filtering for salary values equal to 200000. I get an empty set

```
select salary
from emp_employees
where salary = 200000;
no rows selected
```

But if I do the count and sum of those salaries I get a count of 0 and a sum of null.

```
select sum(salary) as SumSalary, count(salary) as CountSalary
from emp_employees
where salary = 200000;
SumSalary CountSalary
-----
0
1 row selected
```

Some people think this does not make a lot of sense, but that is the way these functions work.

You will sometimes hear people suggest that you use an expression such as count(1) to count rows, or count(customer_id) with the customer table to count rows - customer_id is the pk and is never null. Those will work but there is no advantage to doing this. You could count any non null column- such as Count(customer_name_last) but that is confusing and someone might have changed that column to be a nullable column without checking all queries that use that table. Use a sensible column for count- if you are supposed to count customers, use count(customer_id); if you are supposed to count products, use count(prod_id); if you are supposed to count employees, use count(emp_id). But also see the next section of count(distinct...)

Count(1) works because 1 is a numeric literal- but you could use count (0) or count(3.14159) or count('Hi There') the same way since they are all literals. (If you are working at a job and your boss says to use count(1) instead of count(*), or if that is a company standard- go ahead. it won't hurt the query result.)

The count function requires exactly one argument which can be an expression

Demo 04: Suppose we wanted to know how many customers we have and how many have a last name starting with the letters 'Mor'.

```
select Count(customer_name_last) as CustCount
, count(case when customer_name_last like 'Mor%' then 1 end) as CustNameMorX
from customer customers;
```

CUSTCOUNT	CUSTNAMEMORX
34	5

3. Count(Distinct expression)

Suppose we want to know how many products we have currently on order. It is important to realize that a simple request like this could have different interpretations. Suppose we have only these rows in our order details table.

Order_id	Line_Item_id	Prod_id	Quantity	Quoted_price
1	1	P12	2	5.00
1	2	T20	4	12.00
1	3	R67	1	25.00
2	1	P12	2	5.00
3	1	P12	8	6.50
3	2	R67	5	25.00

The question "how many products do we have currently on order ?" could be interpreted to mean "how many line items do we have ?" ; in this example we have 6 line items- we could answer this by using Count(*). The question could also be interpreted to mean "what is the total quantity of items that we have on order?"; this would be $2 + 4 + 1 + 2 + 8 + 5 = 22$ – for that use the function Sum (quantity). Another interpretation is "how many different products do we have on order "; this would be products P12, R67, and T20; we have three different products currently on order. This is answered using the Count (Distinct prod_id) function.

Demo 05: Using count distinct . We have 184 rows in the order headers table and 34 different products have been ordered.

```
select count(distinct prod_id) as CountDistinctProducts
, count(*) as CountProducts
from oe_orderDetails;

```

CountDistinctProducts	CountProducts
34	184

Note that the use of this keyword Distinct is different in syntax and meaning than the use of Distinct with a Select statement to return only unique rows.

When you add Distinct **inside** the argument list to an **aggregate** function, the function will determine the answer by using only unique values.

Demo 06: How many orders do we have? We can count the order ids in the order headers table.

```
select count(order_id) as "CountOrders"
from oe_order_headers;

```

CountOrders
97

Demo 07: If we count the order id in the order detail tables, we count any order with multiple order lines more than once. Use distinct to count each order id only once.

```
select count( order_id) as "CountOrders"
, count(distinct order_id) as "CountDistinctOrders"
from oe_orderDetails;

```

CountOrders	CountDistinctOrders
184	94

If you are wondering about the value 97 in demo 06 and the value 94 in demo 07, we have three orderheaders rows which have no detail rows.. It is a business decision if we should count those as Orders- they have an order id but there is nothing ordered.

Demo 08: How many customers do we have with orders? We can count the distinct customer ids in the order headers table. If we count customer_id, then we count customer with multiple orders multiple times. We have only 34 customers.

```
select count(distinct customer_id), count(customer_id)
from oe_order_headers;

```

UsingDistinct	WithoutDistinct
21	97

Suppose we want to find out how many different combinations of customers and shipping modes we have. I am not going to count the orders where there is no info on the shipping modes.

This does not work

```
select count( distinct customer_id, shipping_mode)
from oe_order_headers
where shipping_mode is not null
;
```

ORA-00909: invalid number of arguments

Demo 09: But I can run this query to see the distinct combinations of customer ids and shipping modes

```
select distinct customer_id, shipping_mode_id
from oe_order_headers
where shipping_mode_id is not null;
customer_id shipping_mode
-----
400300 USPS1
401250 FEDEX1
401250 FEDEX2
401890 USPS1
402100 USPS1
403000 FEDEX1
403000 UPSGR
403000 USPS1
. . .
```

Demo 10: And then I can put that into a CTE to get the count

```
With CTE as (
    select distinct customer_id, shipping_mode_id
    from oe_order_headers
    where shipping_mode_id is not null
)
select COUNT(*)
from CTE;
COUNT(*)
-----
33
```

Demo 11: We have about 100 rows in the order headers table. We want to see how many different months are included in those orders. The To_Char function is used to extract just the year and month from the order date.

```
select count( distinct To_char( order_date, 'YYYY-MM')) as "number of months"
from oe_orderHeaders
;
number of months
-----
12
```

Demo 12: Use a CTE to make it easier to work with the aggregates. The CTE does the Distinct processing.

```
With orderdates as (
    select distinct to_char(order_date, 'YYYY-MM') as YrMnth
    from oe_orderHeaders
)
select count (YrMnth) as "number of months"
, min(YrMnth) as "earliest month"
, max(YrMnth) as "latest month"
from orderdates;
number of months earliest latest
-----
12 2015_04 2016-05
```

You can use Distinct with any of the aggregates- but don't use it if you really don't need it. If you are looking for the price of the most expensive item we sell, Max(price) will return the same value as Max(Distinct price).

I have never seen a good business reason for calculating sum(distinct prod_list_price) or for calculating sum(prod_list_price) from the products table.

4. Count with Inner and with Outer Joins

Suppose we join the tables products and order details to see which products have been ordered. If we do this with an inner join then we get only products which have been ordered and if we use a left join (Products Left Join order_details) then we get products with orders and products without orders. This means you need to take more care with aggregate functions. The Count function is the easiest function for illustrating this point.

Demo 13: Inner join

```
select PR.prod_id, OD.prod_id, PR.catg_id
from prd_products PR
join oe_orderDetails OD on Pr.prod_id = OD.prod_id
where pr.prod_id between 1140 and 1150
order by PR.prod_id;
```

PROD_ID	PROD_ID	CATG_ID
1140	1140	PET
1141	1141	PET
1150	1150	PET
16 rows selected		

Demo 14: Outer join- we get 2 more rows - products 1142, 1143 with no orders

```
select PR.prod_id, OD.prod_id, PR.catg_id
from prd_products PR
left join oe_orderDetails OD on Pr.prod_id = OD.prod_id
where pr.prod_id between 1140 and 1150
order by PR.prod_id;
```

PROD_ID	PROD_ID	CATG_ID
1140	1140	PET
1141	1141	PET
1142		PET
1143		PET
1150	1150	PET
18 rows selected		

Demo 15: using the inner join and aggregates

```
select count(*) as "RowCount"
, count(distinct order_id) as "OrderCount"
, count(distinct PR.prod_id) as "PR_DistProdCount"
, count(distinct OD.prod_id) as "OD_DistProdCount"
from prd_products PR
join oe_orderDetails OD on PR.prod_id = OD.prod_id
where pr.prod_id between 1140 and 1150
;
```

RowCount	OrderCount	PR_DistProdCount	OD_DistProdCount
16	11	3	3

Demo 16: using the outer join and aggregates

```
select count(*) as "RowCount"
, count(distinct order_id) as "OrderCount"
, count(distinct PR.prod_id) as "PR_DistProdCount"
, count(distinct OD.prod_id) as "OD_DistProdCount"
from prd_products PR
left join oe_orderDetails OD on PR.prod_id = OD.prod_id
where pr.prod_id between 1140 and 1150
;
```

RowCount	OrderCount	PR_DistProdCount	OD_DistProdCount
18	11	5	3

The differences between the results for Demo 13 and 14.:

RowCount With the outer join we get two more rows because we include the rows for the two products with no orders

OrderCount These values are the same, we are counting the order_id values which occur only in the order details table. Since the rows for the products with no orders have a null in that column, they do not get counted.

PR_DistProdCount With this column we are counting the product id value from the products table. With the outer join we do count the product id for the two products with no orders.

OD_DistProdCount. With this column we are counting the product id value from the order details table. Since the rows for the products with no orders have a null in that column, they do not get counted

5. Summary

Count (*) counts rows.

Count (col) counts the values for that column, and does not count Nulls.

Count will always return a numeric integer answer- if the table is empty or there are no matching rows, it will return 0. You don't need to use coalesce with count.

Sum, Avg, Max and Min will return nulls when there are no matching rows

Sum, Avg, Max and Min ignore nulls when doing their calculations

This is a point in the semester where it makes sense to realize that SQL is like every other computer system. It is built on a system of rules and it implements those rules. The rules it has will not always agree with what you think they should be- but it makes no sense to fight a compiler. You need to know the rules the system uses and decide if those rules make sense for the processing you need to do.

Table of Contents

1.	Partitioning the Table Expression with Group By	1
1.1.	Multiple-column grouping	3
1.2.	Grouping by expressions.....	4
2.	Using Groups and Aggregate Functions	5
3.	Using the Having Clause.....	8
3.1.	Grouping on additional columns.....	10
4.	Group By and Nulls.....	12
5.	Sorting	12
6.	SQL layout	13

In the previous documents, we did aggregates on the table as a whole- getting one row of data from the query. We also have the ability to take a table and partition it into a set of sub-tables. With a partition, each row of the table is put into exactly one of the sub-tables- into one of the groups. We can then use an aggregate function with each of these subsets.

Amazon might want to know the total sales by month for each month of the year or average shipping costs it incurs by zip code. This requires partitioning the data in the tables by some characteristic (sales month or zip code) and then calculating summary data for each of these groups.

This uses a new clause in our Select statement- the **Group By** clause. This would let us find the most expensive product in each of our product categories; we can group by the category and use Max for each group.

At times we may want to see aggregate values only if the aggregate itself meets a specified criterion- for this we have another new clause- the **Having** clause. For example, we might want to see customers who have more than 5 orders in our tables.

At the end of this unit, we will have the following model for the Select statement.

```
select . . .
from . . .
where . . .
group by . . .
having . . .
order by . . .
```

In a previous document, we mentioned the Logical processing order. You need to understand that in order to understand how your query is processed. Adding in these two new clauses the order is:

1. The FROM clause is evaluated first
2. The WHERE clause
3. The GROUP BY clause
4. The HAVING clause
5. The SELECT clause
6. The ORDER BY clause is done last

When the query is presented to the dbms, the parser and optimizer determine the actual steps used in the execution and you do not directly control that. But you should consider the statement as being processed in that order.

1. Partitioning the Table Expression with Group By

When you partition a table expression, you create a series of virtual sub-tables/partitions into which each row from the parent table will be placed; each table row appears in only one of these groups. You can partition the table on the basis of one or more attributes or expressions. When you apply an aggregate function to a partitioned table, you will get a separate summary value for each sub-table rather than for the table as a whole.

We start here with just the Group By clause without any aggregates.

Demo 01: Displaying all departments— one per employee.

```
select dept_id  
from emp_employees  
order by dept_id;
```

dept_id

10
20
30
30
30
30
30
30
30
35
35
35
80
80
80
210
210
215
215
215
215

Demo 02: Displaying one of each different department represented in the employee table.

Here we group by the dept_id and return one row for each grouping
We could also do this by using the keyword Distinct.

```
select dept_id  
from emp_employees  
group by dept_id  
order by dept_id;
```

dept_id

10
20
30
35
80
210
215

Demo 03: But we cannot show the emp_id since there is no emp_id value that represents the department as a group.

```
select dept_id, emp_id  
from emp_employees  
group by dept_id;
```

```
select dept_id, emp_id  
      *  
ERROR at line 1:  
ORA-00979: not a GROUP BY expression
```

1.1. Multiple-column grouping

In the following example, we have two columns in the Group By clause and we get more groups. The more grouping columns we have, the finer the distinction we are making between the way that we categorize a row and the more potential groups we will have. If we were to group by the pk columns, we would make a group for each row in the table. This is generally not a good idea.

Note that we get a sub-table only for data combinations that actually exist in our table. We have at least two rows with different job_id values in department 80 so we get two sub-tables for department 80 but for dept 10 we have only one job_id. This is **not** a Cartesian product of all possible combinations of dept_id and job_id.

Demo 04: Using two grouping attributes gives us more sub-tables.

```
select dept_id, job_id
from emp_employees
group by dept_id, job_id
order by dept_id, job_id ;
```

DEPT_ID	JOB_ID
10	1
20	2
30	16
30	32
35	8
35	16
80	4
80	8
210	32
210	64
215	16
215	32
215	64

Demo 05: If we group first by the job_id and then by the dept_id, we get the same result set because we are creating the same set of subtables. We get a group for each combination of dept id and job id,

```
select dept_id, job_id
from emp_employees
group by job_id, dept_id
order by dept_id, job_id ;
```

Demo 06: You can sort the result set after you group it.

```
select dept_id, job_id
from emp_employees
group by job_id, dept_id
order by job_id, dept_id;
```

DEPT_ID	JOB_ID
10	1
20	2
80	4
35	8
80	8
30	16
35	16
215	16
30	32
210	32
215	32
210	64
215	64

Note that Oracle does not promise that the ordering of the result set is determined by any grouping. If you want the result set sorted, then use an Order By clause.

In these queries, we have been displaying the grouping attributes. That is not a requirement, but is generally helpful.

Demo 07: This would be a confusing display since it is not obvious why some department id values appear more than once.

```
select dept_id
from emp_employees
group by job_id, dept_id
order by dept_id;
```

dept_id

10
20
30
30
35
35
80
80
210
210
215
215
215

1.2. Grouping by expressions

You can also use expressions in Group By clauses.

Demo 08: A common business need is to group sales by year or by year and month. This also adds in the count function.

```
select extract( year from order_date) as OrdYear , count(*) as NmbOrders
from oe_orderHeaders
group by extract( year from order_date)
order by extract( year from order_date)
; 
```

ORDYEAR	NMBORDERS
-----	-----
2015	57
2016	40

Demo 09: Grouping by year and month. There is no row for 2015 month 5 or 7 since we have no rows in the table for that month.

```
select extract( year from order_date)as OrdYear
, extract( month from order_date)as OrdMonth
, count(*) as NmbOrders
from oe_orderHeaders
group by extract( year from order_date), extract( month from order_date)
order by extract( year from order_date), extract( month from order_date)
; 
```

ORDYEAR	ORDMONTH	NMBORDERS
2015	4	1
2015	6	12
2015	8	8
2015	9	7
2015	10	9
2015	11	12
2015	12	8
2016	1	14
2016	2	8
2016	3	5
2016	4	7
2016	5	6

Using a CTE can help make this easier to think about.

```
With orderdates as (
    select extract (year from order_date) as OrdYear
    ,extract (month from order_date) as OrdMonth
    from oe_orderHeaders
)
select OrdYear, OrdMonth, count(*)
from orderdates
group by OrdYear, OrdMonth
order by OrdYear, OrdMonth ;
```

2. Using Groups and Aggregate Functions

We want to create the sub-tables so that we could get information about the groupings. In this example, we want to know how many employees are in each department and their average salary. Since we want data for **each** department, we group by the department id.

Demo 10: For each department, how many employees are there and what is the average salary for that department?

```
select dept_id
, count(*) as empCount
, avg(salary) as AvgSalary
from emp_employees
group by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
30	8	76599.875
20	1	15000
210	2	67000
215	4	83563.5
35	3	64333.3333
80	3	36000
10	1	100000

Demo 11: What is the average list price and the largest list price for **each category** of product we sell?

```
select catg_id
, avg(prod_list_price) as "Avg List Price"
, max(prod_list_price) as "Max List Price"
from prd_products
group by catg_id
;
```

CATG_I	Avg List Price	Max List Price
PET	123.167273	549.99
HD	23.875	45
MUS	13.8781818	15.95
HW	67.641	149.99
SPG	178.125	349.95
APL	479.986	850
GFD	8.75	12.5

Basic Rule for Grouping and Aggregates:

If you use an aggregate function and a non-aggregated attribute in the SELECT clause, then you must GROUP BY the non-aggregated attribute(s).

If you use a GROUP BY clause, then the SELECT clause can include only aggregate functions and the attributes used for grouping. You are not required to show the grouping column in the output.

Demo 12: Using two groups and aggregate functions; this groups by the different departments and job ids. It then sorts the rows by the avg salary.

```
select dept_id, job_id
, count(*) as NumEmployee
, avg(salary) as AvgSalary
from emp_employees
group by dept_id, job_id
order by avg(salary);
```

DEPT_ID	JOB_ID	NUMEMPLOYEE	AVGSALARY
20	2	1	15000
80	8	2	24500
35	8	1	30000
80	4	1	59000
210	32	1	65000
215	32	1	65000
210	64	1	69000
215	64	2	74627
30	16	4	74863.75
30	32	4	78336
35	16	2	81500
10	1	1	100000
215	16	1	120000

Demo 13: Suppose we wanted to show statistics for average salaries by department- but not identify the department by name (for political reasons!) It is up to the user if this output is useful.

```
select 'not specified' as Department, count(*) as EmployeeCount
, Round(avg(salary),0) as AvgSalary
from emp_employees
group by dept_id
order by avg(salary);
```

DEPARTMENT	EMPLOYEECOUNT	AVGSALARY
not specified	1	15000
not specified	3	36000
not specified	3	64333
not specified	2	67000
not specified	8	76600
not specified	4	83564
not specified	1	100000

Demo 14: Oracle allows a syntax such as this which does the aggregates over the entire table and uses a group by with empty parentheses.

```
select count(*) as NumEmployee
, avg(salary) as AvgSalary
from emp_employees
group by ();
```

NUMEMPLOYEE	AVGSALARY
-----	-----
22	68047.8636

In the previous document we found the most expensive SPG item and we found the most expensive PET item. What if we want to find the most expensive item(s) in **each** category?

Demo 15: We can use this query to find the max price in each category

```
select catg_id, max(prod_list_price) as MaxPrice
from prd_products
group by catg_id
;
```

CATG_I	MAXPRICE
PET	549.99
HD	45
MUS	15.95
HW	149.99
SPG	349.95
APL	850
GFD	12.5

Demo 16: We can use this query as a CTE and join to the product table on the category id and then use a test that price equals the max price for that category. Note that we do get ties.

```
with maxprices as (
    select catg_id, max(prod_list_price) as MaxPrice
    from prd_products
    group by catg_id
)
select p.catg_id, p.prod_id, substr(p.prod_desc, 1, 25) as prod_desc
, p.prod_list_price
from prd_products P
join maxprices MP on p.catg_id = MP.catg_id
                    and p.prod_list_price = MP.maxPrice
;
```

CATG_I	PROD_ID	PROD_DESC	PROD_LIST_PRICE
HW	1090	Gas grill	149.99
SPG	1040	Super Flyer Treadmill	349.95
HW	1160	Stand Mixer with attachme	149.99
PET	4567	Our highest end cat tree-	549.99
PET	4568	Satin four-poster cat bed	549.99
APL	1126	Low Energy Washer Dryer c	850
GFD	5000	Cello bag of mixed finger	12.5
HD	5005	Steel Shingler hammer	45
MUS	2014	Bix Beiderbecke - Tiger R	15.95

3. Using the Having Clause

Sometimes you have grouped your rows but you do not want to see all of the groups displayed. Perhaps you have grouped employees by department and only want to see those departments where the average salary is more than 60,000. In this case you want to filter the returns by the aggregate function return values. **It is not allowed to put an aggregate function in the Where clause.** So we need another clause to do this- the Having clause. The Having clause filters the groups. You cannot use a column alias in a Having clause. In most cases you will use aggregate functions in the Having clause.

Demo 17: This groups by department and uses all rows.

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from emp_employees
group by dept_id
order by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
10	1	100000
20	1	15000
30	8	76599.875
35	3	64333.3333
80	3	36000
210	2	67000
215	4	83563.5

Demo 18: This groups by department and returns groups where the average salary exceed 60000. The test on avg(salary) is in the HAVING clause.

```
select dept_id
, count(*) as EmpCount
, avg(salary)
from emp_employees
group by dept_id
having avg(salary) > 60000
order by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
10	1	100000
30	8	76599.875
35	3	64333.3333
210	2	67000
215	4	83563.5

If you have a choice of filtering in the Where clause or in the Having clause- pick the Where clause. Suppose we want to see the average salary for department where the average is more than 60000 but we only want to consider department 30-40. In that case we should filter on the dept_id in the Where clause and filter on the Ave(salary) in the Having clause. There is no sense creating groups for departments that we do not want to consider.

Demo 19: Groups with Where clause and a Having clause

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
```

```
from emp_employees
where dept_id between 30 and 40
group by dept_id
having avg(salary) > 60000
order by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
30	8	76599.875
35	3	64333.3333

There are several different meanings to finding averages. The following query allows only employees who earn more than 60000 to be put into the groups- and we get a different answer. The issue here is not "which is the right query?" but rather "what do you want to know?" Do you want to know the average salary of the more highly paid employees or the departments with the higher average salaries?

Demo 20: Using a Where clause— this acts before the grouping. Only employees earning more than 60000 get into the groups.

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from emp_employees
where salary > 60000
group by dept_id
order by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
10	1	100000
30	8	76599.875
35	2	81500
210	2	67000
215	4	83563.5

Demo 21: Show statistics only if the department has more than three employees.

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from emp_employees
group by dept_id
having count(*) > 3
order by dept_id ;
```

DEPT_ID	EMPCOUNT	AVGSALARY
30	8	76599.875
215	4	83563.5

Demo 22: What if you want to find out if you have more than one employee with the same last name?

```
select name_last as DuplicateName
, count(*) as NumEmp
from emp_employees
group by name_last
having count(*) > 1;
```

DUPLICATENAME	NUMEMP
King	2
Russ	2

Demo 23: If you want to determine the number of order lines for each order, you can use the order details table and group on the order_id. How many order lines for each order?

```
select order_id
, count(*) as "NumberLineItems"
from oe_orderDetails
group by order_id
order by order id;
```

selected rows	ORDER_ID	NumberLineItems
	105	3
	106	1
	107	1
	110	2
	111	2
	112	1
	115	4

3.1. Grouping on additional columns

But suppose I also wanted to show the customer ID and the shipping mode for each of these orders. You know that each order has a single customer_id, and shipping mode, so it would be logical that you could just add them to the Select clause. In Oracle to display these attributes, you either need to include them in the Group By clause, or wrap them in an aggregate function, such as Max, in order to display them. You need to be certain that the extra grouping attributes are not changing the logic of the query.

Demo 24: You need to include extra group levels for this query.

```
select OH.customer_id
, OH.order_id
, OH.shipping_mode_id
, count(*) AS "NumberLineItems"
from oe_orderHeaders OH
join oe_orderDetails OD on OH.order_id = OD.order_id
group by OH.order_id, customer_id, shipping_mode_id
order by OH.customer_id, OH.order_id;
```

CUSTOMER_ID	ORDER_ID	SHIPPING_MODE_ID	NumberLineItems
400300	378	USPS1	2
401250	106	FEDEX1	1
401250	113	FEDEX2	1
401250	119		1
401250	301	FEDEX2	1
401250	552	FEDEX1	2
401890	112	USPS1	1
401890	519	USPS1	2
. . . rows omitted			

Demo 25: What is the amount due for each order? Note that I am able to use order_date in the group by clause and use order_date within a function in the Select ; the order_date attribute is a datetime value.

```
select CS.customer_id, CS.customer_name_last, OH.order_id
```

```
, to_char(order_date, 'yyyy-mm-dd') as OrderDate
, sum( quantity_ordered * quoted_price) as AmntDue
from cust_customers CS
join oe_orderHeaders OH on CS.customer_id = OH.customer_id
join oe_orderDetails OD on OH.order_id = OD.order_id
group by OH.order_id, CS.customer_id, CS.customer_name_last, OH.order_date
order by CS.customer_id, OH.order_id;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	ORDER_ID	ORDERDATE	AMNTDUE
400300	McGold	378	2015-06-14	4500
401250	Morse	106	2015-10-01	255.95
401250	Morse	113	2015-11-08	22.5
401250	Morse	119	2015-11-28	225
401250	Morse	301	2015-06-04	205
401250	Morse	552	2015-08-12	157.3
401890	Northrep	112	2015-11-08	99.98
401890	Northrep	519	2016-04-04	114.74
...	rows omitted			

Demo 26: It does allow the following which produces one group per order date and displays the year for that group. It is up to you to decide if this output is meaningful.

```
select extract(year from order_date) as OrdYear , count(*)as NmbOrders
from oe_orderHeaders
group by order_date
order by order_date;
```

ORDYEAR	NMBORDERS
2015	1
2015	1
2015	6
2015	2
2015	1
...	

It probably makes more sense to group by the year of the order date.

```
select extract(year from order_date) as OrdYear , count(*)as NmbOrders
from oe_orderHeaders
group by extract(year from order_date);
```

ORDYEAR	NMBORDERS
2015	57
2016	40

Demo 27: This one produces a group for each combination of first and last name and then allows you to use those grouping expressions in the Select within a function.

```
select coalesce(customer_name_first || ' ', '') || customer_name_last as
Customer
, count(*) as NbrOrders
from customer_customers
group by customer_name_last, customer_name_first
order by customer_name_first, customer_name_last;
```

CUSTOMER	NBRORDERS
Abigail Adams	1
Al Williams	1
Alexis Hamilton	1

Arnold McGold	3
Barry Mazur	1
...	

4. Group By and Nulls

Demo 28: The order headers table has several rows where there is no value for the shipping mode. What happens if we group by the column shipping mode?

```
select shipping_mode_id
, count(order_id) as OrderCount
from oe_orderHeaders
group by shipping_mode_id
order by shipping mode_id;
```

SHIPPING_MODE_ID	ORDERCOUNT
FEDEX1	26
FEDEX2	2
UPSEXP	5
UPSGR	15
USPS1	38
USPS2	3
	8

When we group by the shipping mode, we get one group for all of the nulls. We have eight rows in that group.

Demo 29: What if we want to display a message instead of a null for the null group? We can use coalesce. If the column we are displaying is not a string, we would need to cast it to a string.

```
select
coalesce(shipping_mode_id, 'No shipping mode') as ShippingMode
, count(order_id) as OrderCount
from oe_orderHeaders
group by shipping_mode_id
order by shipping mode_id;
```

SHIPPINGMODE	ORDERCOUNT
FEDEX1	26
FEDEX2	2
UPSEXP	5
UPSGR	15
USPS1	38
USPS2	3
No shipping mode	8

5. Sorting

Demo 30: Suppose we take the previous query and try to sort by the order_id. This is not allowed.

```
select
coalesce(shipping_mode_id, 'No shipping mode') as ShippingMode
, count(order_id) as OrderCount
from oe_orderHeaders
group by shipping_mode_id
order by order_id;
```

```
order by order_id
*
ERROR at line 5:
ORA-00979: not a GROUP BY expression
```

Demo 31: This is allowed. The Where clause is simply to reduce the number of rows in the result set. The Order By clause gets the largest (most recent) order date for each customer. If this does not make sense, add the max(order_date) expression to the Select and run that.

```
select customer_id  
from oe_orderHeaders  
where customer_id > 900000  
group by customer_id  
order by max(order_date) desc;
```

6. SQL layout

For a query that uses grouping, the required SQL layout has the Group By clause and the Having clause on new lines

```
select dept_id  
, avg(salary) as avgsalary  
from emp_employees  
group by dept_id  
having count(*) > 1;
```

This is an Oracle feature; this is not found in MySQL nor in SQL Server. Be very careful when you use this. People make a lot of mistakes with this approach (particularly on the final exam).

1. Two Level Nested Aggregates

Suppose we want to find the average amount of our orders. To do this, we need to group each order into a sub-table- so we need a group by order_id. We then need to find the sum of the price * quantity of those groups. Finally we need to find the average of that calculated data. Oracle allows nesting the aggregate functions with some limitations.

Demo 01: Step 1: What is the amount due for each order?

```
select order_id
, sum( quantity_ordered * quoted_price) AS AmntDue
from oe_orderDetails
group by order_id;
```

ORDER_ID	AMNTDUE
105	1205.4
106	255.95
107	49.99
108	22.5
109	149.99
. . . rows omitted	

Demo 02: Step 2 What is the average of those sums? This is nesting the sum function inside the avg function.

```
select avg( sum( quantity_ordered * quoted_price) ) AS "AvgOrderSize"
from oe_orderDetails
group by order_id;
```

AvgOrderSize
904.148723

We can find the highest average salary of any department and we can find the average of the highest salary in each department. Again this is not a matter of which query is correct- the question is what do you want to know.

Demo 03: What is the highest average salary of any department? We get a single row returned.

first what is the average salary by department?

```
select to_char(avg(salary), '999,999.00') as "AvgSalary"
from emp_employees
group by dept_id;
```

AvgSalary
76,599.88
15,000.00
67,000.00
83,563.50
64,333.33
36,000.00
100,000.00

```
select max (avg(salary))
from emp_employees
group by dept_id;
```

MAX(AVG(SALARY))
100000

Demo 04: What is the average of the highest salary of any department? We get a single row returned.

Step 1 highest salary by department

```
select max(salary)
from emp_employees
group by dept_id;
```

```
MAX(SALARY)
```

```
-----
99090
15000
69000
120000
98000
59000
100000
```

```
select avg ( max (salary))
from emp_employees
group by dept_id;
```

```
AVG(MAX(SALARY))
```

```
-----
80012.8571
```

See another version in the demo using a CTE.

Demo 05: If the company fired the highest paid person in each department, what would they save in salary?

```
select sum (max( salary ))
from emp_employees
group by dept_id;
```

```
SUM(MAX(SALARY))
```

```
-----
560090
```

Now consider a demo we had recently which calculated the amount due for each order. This is that query, with fewer columns.

Demo 06: These are customers and the amounts of their orders. We group by the order id so that we can get the amount for each order and add the customer id to the grouping so that we can display that. Each order has only one customer id, so we are not changing the grouping logic.

```
select customer_id, order_id
, to_char( sum( quantity_ordered * quoted_price), '999,999.00') as "AmntDue"
from oe_orderHeaders
join oe_orderDetails using (order_id)
group by order_id, customer_id
order by "AmntDue" desc
;
```

CUSTOMER_ID	ORDER_ID	AmntDue
900300	609	9,630.00
903000	312	9,405.00
400300	378	4,500.00
900300	307	4,500.00
903000	2121	3,800.25
903000	551	3,500.00
403000	395	2,925.00
409150	415	2,879.95
...		

Demo 07: Now we nest the aggregates to find the largest amount due for any of the orders. The group by clause affects the inner aggregate - in this case the sum and the outer aggregate works over that result set.

```
select max( sum( quantity_ordered * quoted_price)) as "LargestAmntDue"
from oe_orderHeaders
join oe_orderDetails using (order_id)
group by order_id, customer_id
order by customer_id, order_id
;

```

LargestAmntDue

9630

Now you want to see the largest order per customer- It would seem to make sense to try to use max(sum(quantity_ordered * quoted_price)) but if you did that, what is the group by clause? We want to group by the order id to get the amt due per order but we want one row per customer so it looks like the group by clause should be just the customer id. Oracle does not allow the following query.

```
select
  customer_id
, order_id
, max(sum(quantity_ordered * quoted_price)) as "AmntDue"
from oe_orderHeaders
join oe_orderDetails Using (order_id)
group by customer_id;
```

The error message is rather clumsy.

Error report:
SQL Error: ORA-00937: not a single-group group function

What you really need to do is say do this aggregate -sum- grouping by the order id and then another aggregate -max- grouping by the customer id. We can do that with a CTE- isolating the first aggregate in the CTE and then using Max in the main query.

Demo 08:

```
with amtDueByOrder as (
  select customer_id, order_id
  ,sum( quantity_ordered * quoted_price) as AmntDue
  from oe_orderHeaders
  Join oe_orderDetails using (order_id)
  group by order_id, customer_id
)
select customer_id, max(amntDue) as "LargestOrderByCustomer"
from amtDueByOrder
group by customer_id
order by customer_id;
```

CUSTOMER_ID	LargestOrderByCustomer
-----	-----
400300	4500
401250	255.95
401890	114.74
402100	2305
403000	2925
403010	1900
...	

Table of Contents

1. What is a cross tab report and who wants one	1
2. Using Aggregates & Case for a Cross Tab.....	3
2.1. Adding a grouping	6
2.1. Doing a count instead of a sum.....	7
2.2. Filtering the entire data source and using a calculated case expression.....	7

This is one of those topics where people tell me they spent/wasted hours (and hours) looking this up on the internet and got more confused. So don't do that yet. One of the problems is that a CrossTab result is a generic report type of result and there is more than one way to do this. Some dbms has developed their own proprietary techniques for this process. Some web pages take you into rollups - which we do later. Some web pages go into correlated subqueries which we have not covered yet.

This document discusses one technique which should work across all dbms commonly used. That makes it valuable for people who work in more than one system. And this technique is required for the assignment. In a job situation (or on the final exam) you should use whichever technique is most efficient for your situation. But for now we are sticking with creating a cross tab report using the case expression and grouping and aggregates.

1. What is a cross tab report and who wants one

If you do a search for "cross tab report" and look at images, you should get an idea that these are very common business reports. This is the way that a lot of people need to see their data organized. These are a few screen shots.

This is a cross table report of sales(?) by country and by product. This report includes a total row and a total column.

A Cross-Tab report displays the information in a compact format, which makes it easier to see the results.

	China	England	France	Japan	USA	Total
Active Outdoors Crochet Glove		12.00	4.00	1.00	240.00	257.00
Active Outdoors Lycra Glove		10.00	6.00		323.00	339.00
InFlux Crochet Glove	3.00	6.00	8.00		132.00	149.00
InFlux Lycra Glove		2.00			143.00	145.00
Triumph Pro Helmet	3.00	1.00	7.00		333.00	344.00
Triumph Vertigo Helmet		3.00	22.00		474.00	499.00
Xtreme Adult Helmet	8.00	8.00	7.00	2.00	251.00	276.00
Xtreme Youth Helmet		1.00			76.00	77.00
Total	14.00	43.00	54.00	3.00	1,972.00	2,086.00

The cross-tab is made up of rows, columns, and summary fields. The summary fields are the intersection of rows and columns. Their values represent a summary (sum, count, and so on) of those records that meet the row and the column criteria.

A Cross-Tab also includes several totals: row totals, column totals, and grand totals. The grand total is the value at the intersection of the row total and the column total.

http://www-01.ibm.com/support/knowledgecenter/SS4JCV_7.5.5/com.businessobjects.integration.eclipse.designer.doc/html/topic299.html

A cross table report of a survey result by age and student status. This report includes a total column. The data points are expressed as percents and the data points are color shaded

What is your age?	Are you a student?			Total
	Yes - Full Time	Yes - Part Time	No	
15 and under	88%	12%	-	8
16 - 18	95%	-	5%	42
19 - 23	68%	12%	20%	205
24 - 29	15%	10%	74%	353
30 - 35	5%	9%	86%	182
36 - 45	4%	8%	88%	165
over 45	1%	7%	92%	129

Cross tab report of expense reports filed by year and department.

Department Expense Reports Filed

	Finance	HR	IT	Manufacturing
1 2008	1	1	4	52
2 2009	1	1	5	55
3 2010	1	1	6	60
4 2011	2	1	5	60

A cross tab reports looks more like a spreadsheet display. In these examples we have three sources of data. In the Expense Reports Filed example we have years(2008,2009, 2010,2011). These come down the first column in the result. We have Departments (Finance, HR, IT, Manufacturing). These go across the first row of the result. The third type of data is the number (count) of reports files by each department for each year. These go into the cells.

A cross tab query normally aggregates data and displays it with the **aggregate** (sum or count or others) values as the cells and two sets of grouping data- one across the top and the other down the left side.

Some of the examples include totals as a right-most column- the first example has a total for all countries as the right most column. We will do that in this discussion. Some of the examples have a total down the columns as the last row in the display. We do that with roll up and we won't discuss that in this unit.

When you look at images of crosstab reports you will also see color shading of data, you may see more than one data value in a cell. These features are generally added by report writing applications.

This is a simpler example that we could do with our vets data. This is a simple grouping query with the total fees by animal type for exams in 2103; this is not a cross tab query. This is not using this semester's data- this is just a sample display.

AN_TYPE	FEETOTAL
cat	899.46
porcupine	299.50
lizard	515.00
hedgehog	110.00
dog	201.00
chelonian	200.00
dormouse	275.00

But we want to break the fees down by quarters.

AN_TYPE	QUARTER	FEETOTAL
cat	2	899.46

chelonian	1	100.00
chelonian	2	100.00
dog	4	201.00
dormouse	4	275.00
hedgehog	4	110.00
lizard	1	125.00
lizard	2	145.00
lizard	3	50.00
lizard	4	195.00
porcupine	1	49.50
porcupine	3	200.50
porcupine	4	49.50

That really doesn't look very good. It is harder to compare quarter by quarter and we do not have any rows for quarters with no fees.

The following is easier to read and it is easy to add a column for the total year and rows for animal types with no exams if we want that. This is a cross tab report. The column at the left shows the an_type values; the header row displays the second grouping condition- the quarter of the year in which the exam took place. It looks more like a standard business report.

The three data sources are: animal type, the exam date(specifically the quarter) and the **total** fees for each quarter for each animal type. We do have a Total (all_2013) column on the right.

AN_TYPE	QRT1_2013	QRT2_2013	QRT3_2013	QRT4_2013	ALL_2013
bird	0.00	0.00	0.00	0.00	0.00
cat	0.00	899.46	0.00	0.00	899.46
chelonian	100.00	100.00	0.00	0.00	200.00
dog	0.00	0.00	0.00	201.00	201.00
dormouse	0.00	0.00	0.00	275.00	275.00
hamster	0.00	0.00	0.00	0.00	0.00
hedgehog	0.00	0.00	0.00	110.00	110.00
lizard	125.00	145.00	50.00	195.00	515.00
porcupine	49.50	0.00	200.50	49.50	299.50
snake	0.00	0.00	0.00	0.00	0.00

We could also transpose this and list the quarters in the first column and the an_type values across the first row. One reason to refer to this as a report, rather than a query result is that the order of the columns and rows is significant to the usability of the report.

2. Using Aggregates & Case for a Cross Tab

We will use the prd_products table and create a cross tab query for the total quantity sold for selected category of item for each order. The category grouping will be the row header and the order_id grouping will be the first column. The three sources of data will be order_id (in the first column), certain product categories as the first row and the total quantity (sum) as the data points.

This is where we are heading in the first set of demos: What is the total quantity sold for each of these product categories?

HD_QntSold	SPG_QntSold	HW_QntSold	GFD_QntSold	APL_QntSold
29	305	200	0	84

We can start by looking at ways to get the total sold for the HD category- hardware as in hammers.

Demo 01: First we can display the current data we have for HD orders. I picked a category with few sales so that you can double check the calculations. This has no aggregates.

```
select order_id as "OrdID", prod_id as "Product"
, OH.order_date as "OrdDate", OD.quantity_ordered as "HD_QntSold"
from oe_orderHeaders OH
join oe_orderDetails OD using(order_id)
join prd_products PR using(prod_id)
where PR.catg_id = 'HD';
```

OrdID	Product	OrdDate	HD_QntSold
400	5008	15-OCT-15	5
400	5005	15-OCT-15	5
400	5004	15-OCT-15	5
400	5002	15-OCT-15	5
401	5002	15-OCT-15	3
402	5002	18-OCT-15	3
407	5008	15-NOV-15	1
407	5005	15-NOV-15	1
407	5005	15-NOV-15	1

Demo 02: If we filter for a value of catg_id, we can get an aggregate across the table and one row returned.

That is the total quantity ordered for hardware items.

I do not need the order headers table for this query.

```
select sum(quantity_ordered) as "HD_QntSold"
from oe_orderDetails OD
join prd_products PR using(prod_id)
where catg_id = 'HD';
```

HD_QntSold

29

Demo 03: We can also write a query for the number of details lines that were for a HD item.

```
select count(order_id) as "HD_Detail_lines"
from oe_orderDetails OD
join prd_products PR using(prod_id)
where catg_id = 'HD';
```

HD_Detail_lines

9

Demo 04: We also can find out how many orders included a HD item. Be certain you understand the difference between this query and the previous one- both in terms of the syntax and in terms of what the task is asking for.

```
select count(distinct order_id) as "HD_Orders"
from oe_orderDetails OD
join prd_products PR using(prod_id)
where catg_id = 'HD';
```

HD_Orders

4

Now we are going to work on getting closer to the cross tab based on the sum of the quantity.

Demo 05: We can also use the case expression to include only the rows for HD in the total. This means we do not need the Where clause filter. If the item is included in the HD category. we include it in the sum and if not, we don't include it.

```
select
    sum(case when catg_id = 'HD' then quantity_ordered else null end) as
    "HD_QntSold"
from oe_orderDetails OD
join prd_products PR using(prod_id);
```

HD_QntSold

29

Think about that query for a few minutes and see the syntax and logic pattern.

We want to get the sum of the values for the quantity_ordered but only if this is a HD item.

We know that the sum function will ignore nulls.

So we use the case expression to return one of two values, the quantity_ordered if this is an HD item and a null if it is not. We then give that result to the sum function.

The net result of that expression is "if this is an HD item, add its quantity_ordered value into the sum.

Demo 06: You create a case expression for **each** column that you want returned. The first column does the sum for the hardware; if the row is for a hardware item(HD), then its quantity is part of the Sum for that column. The second column does the sum for the sporting goods items, etc. All of these column expression follow the same pattern and differ on the category id selects and the column alias.

```
select
    sum(case when catg_id = 'HD' then quantity_ordered else null end) as "HD_QntSold"
    , sum(case when catg_id = 'SPG' then quantity_ordered else null end) as "SPG_QntSold"
    , sum(case when catg_id = 'HW' then quantity_ordered else null end) as "HW_QntSold"
    , sum(case when catg_id = 'GFD' then quantity_ordered else null end) as "GFD_QntSold"
    , sum(case when catg_id = 'APL' then quantity_ordered else null end) as "APL_QntSold"
from oe_orderDetails
join prd_products using(prod_id);
```

HD_QntSold	SPG_QntSold	HW_QntSold	GFD_QntSold	APL_QntSold
29	305	200		84

A few things to note here. (1) Each column case test is set for a specific category and the column alias is hard coded for that category. (2) We have chosen 5 categories only. There are other categories but we are not concerned with them. (3) We got a null for the GFD category.

Why did we get a null for GFD? Take that first query and change it to filter for GFD ; we get no rows returned. We do not have any orders for that category.

Now we need to think about that for awhile. What should we display if there are no orders for a category? This actually is a business rule level decision- but what are our possibilities? We tend to think of NULL as meaning we don't have the data. In this case we know that with our query a Null means there are no orders for that category. We can then say that we do know what that means- the GFD total quantity sold is 0. You need to take a lot of care with a decision like that. (It does not mean that all numeric nulls should be represented as 0. It means that in this situation, that decision would make sense.)

Demo 07: We could handle this by wrapping a coalesce around EACH of the sum expressions. Do not make the mistake of saying only the GFD column needs coalesce. It may be that the next time someone runs this query, there are no SPG sales.

```
select
    coalesce(sum(case when catg_id = 'HD' then quantity_ordered else null end),0) as "HD_QntSold"
    , coalesce(sum(case when catg_id = 'SPG' then quantity_ordered else null end),0) as "SPG_QntSold"
    , coalesce(sum(case when catg_id = 'HW' then quantity_ordered else null end),0) as "HW_QntSold"
    , coalesce(sum(case when catg_id = 'GFD' then quantity_ordered else null end),0) as "GFD_QntSold"
    , coalesce(sum(case when catg_id = 'APL' then quantity_ordered else null end),0) as "APL_QntSold"
from oe_orderDetails
join prd_products using(prod_id);
```

The order of the functions called is Coalesce(Sum(Case

Demo 08: We could also- in this case- rewrite the case expression to return 0 if the category id is not matched.

The Sum aggregate function ignores null, but arithmetically we can add 0 to a running total (SUM) without changing the total. (This will not work the same way with Avg, Max, Min so you are better off just using the null version.)

```
select
    sum(case when catg_id = 'HD' then quantity_ordered else 0 end) as
    "HD_QntSold"
    . . .
from oe_orderDetails
join prd_products using(prod_id)
;
```

2.1. Adding a grouping

So far we did not display a leading column - we have only two data sources- the category and the total quantity. We will add the order id as another data source and display this in the first column. **To do that we need to group on the order id to get one row per order id.**

Demo 09: We want to know how many products of each of these categories are on EACH order so we add a grouping on the order id. Because I do not have a lot of data in the tables, we get a lot of 0 values. I have selected rows that show the HD sales.

```
select
    order_id
    , sum(case when catg_id = 'HD' then quantity_ordered else 0 end) as "HD_QntSold"
    , sum(case when catg_id = 'SPG' then quantity_ordered else 0 end) as "SPG_QntSold"
    , sum(case when catg_id = 'HW' then quantity_ordered else 0 end) as "HW_QntSold"
    , sum(case when catg_id = 'GFD' then quantity_ordered else 0 end) as "GFD_QntSold"
    , sum(case when catg_id = 'APL' then quantity_ordered else 0 end) as "APL_QntSold"
from oe_orderDetails
join prd_products using(prod_id)
group by order_id
order by order_id
;
```

selected rows: 94 rows returned

ORDER_ID	HD_QntSold	SPG_QntSold	HW_QntSold	GFD_QntSold	APL_QntSold
312	0	50	0	0	0
313	0	0	1	0	0
324	0	0	0	0	0
378	0	0	0	0	10
390	0	8	0	0	0
395	0	15	0	0	0
400	20	0	0	0	0
401	3	0	0	0	0
402	3	0	0	0	0
405	0	6	0	0	0
407	3	0	0	0	0
408	0	0	1	0	0
411	0	2	4	0	0
412	0	0	0	0	1
413	0	0	10	0	0
414	0	13	0	0	0

You could write a version that produces spaces instead of 0's- that results in this case in a lot of spaces and make it harder to read across the report. The sql is in the demo.

2.1. Doing a count instead of a sum

Now let's count the number of orders for HD products instead of getting the total quantity.

You might just take a previous demo and change the SUM to COUNT, but that could pose problems.

Demo 10: Suppose you did the following expression for each category (see the demo file for the query)

```
select
  count(case when catg_id = 'HD' then quantity_ordered else 0 end) as "HD_NbrOrders"
, count(case when catg_id = 'SPG' then quantity_ordered else 0 end) as "SPG_NbrOrders "
. . .
```

The result would be the following . WHY? (Hint- the number of rows in the order details table is 184.)

HD_NrbOrders	SPG_NrbOrders	HW_NrbOrders	GFD_NrbOrders	APL_NrbOrders
-----	-----	-----	-----	-----
184	184	184	184	184

And counting the quantity_ordered column should seem wrong. If you want to find out how many Orders- count the order_id. That gives us two more versions. They are in the demo file, but try to figure these out for yourself first. if all you do is look at demos, you do not learn much.

Demo 11: Getting a count

HD_NrbOrders	SPG_NrbOrders	HW_NrbOrders	GFD_NrbOrders	APL_NrbOrders
-----	-----	-----	-----	-----
9	49	54	0	27

Demo 12: Getting a different count

HD_NrbOrders	SPG_NrbOrders	HW_NrbOrders	GFD_NrbOrders	APL_NrbOrders
-----	-----	-----	-----	-----
4	30	39	0	23

If this is still a mystery, look in the demo file for another hint.

2.2. Filtering the entire data source and using a calculated case expression

A lot of business reports want sales data organized by time periods. You can do this by using a date function in the case expression.

Demo 13: How many orders for each customer for each of these three months of last year?

The only table needed is the order headers table. In this case I can count a literal instead of the order id.

This is still the same pattern Aggregate(Case ...). In this situation I do not need coalesce since count will return 0 if there is no data. What does a case expression return if there is no match to the When clause?

```
variable lastyear number;
exec :lastyear := extract(year from sysdate) -1;
select customer_id
, count(case when extract(month from order_date)= 10 then 1 end) as "Oct"
, count(case when extract(month from order_date)= 11 then 1 end) as "Nov"
, count(case when extract(month from order_date)= 12 then 1 end) as "Dec"
from oe_orderHeaders
Where extract(year from order_date)= :lastyear
group by customer_id
order by customer_id;
```

CUSTOMER_ID	Oct	Nov	Dec
400300	0	0	0
401250	1	2	0
401890	0	1	0
402100	0	3	0
403000	3	1	0
403010	0	1	0
403050	1	0	0
403100	3	1	0
.	.	.	.

Suppose we want to display the data by quarter for the year 2015 and include a total column.

First we join the tables, using an outer join to include products with no sales

```
from prd_products PR
left join oe_orderDetails OD on OD.prod_id = PR.prod_id
left join oe_orderHeaders OH on OH.order_id = OD.order_id
```

Then we filter for the year- we want sales for the year 2015. If there are no sales, the order date would be null and we want to include those also.

```
where year(order_date) = 2015 or order_date is null
```

We want to data organized by the category id

```
group by catg_id
```

Now we can come back to the display- the Select. We have 6 columns- the first is the category id, the next column is the total sales for the first quarter, followed by columns for the total sales for the second quarter, third and fourth quarter and the total for all quarters.

We are using the case expression to determine the quarter for each order date; we can use the to_char function with a quarter format

```
case when to_char(order_date, 'Q') = '1'
```

If the sales is in the quarter we want we get the extended cost, otherwise treat it as 0 - that avoids the need to coalesce. We are doing a sum- adding in a 0 does not alter the total.

```
case when to_char(order_date, 'Q') = '1'
then quantity_ordered * quoted_price else 0 end
```

We want to total of all of the sales for that quarter, so we need the sum aggregate function.

We then add a last column that does not test the quarter

Demo 14: Display of sales by quarter for different product categories.

```
select catg_id as "CatgId"
, sum( case when to_char(order_date, 'Q') = '1'
then quantity_ordered * quoted_price else 0 end) as "Qrt1_2015"
, sum( case when to_char(order_date, 'Q') = '2'
then quantity_ordered * quoted_price else 0 end) as "Qrt2_2015"
, sum( case when to_char(order_date, 'Q') = '3'
then quantity_ordered * quoted_price else 0 end) as "Qrt3_2015"
, sum( case when to_char(order_date, 'Q') = '4'
then quantity_ordered * quoted_price else 0 end) as "Qrt4_2015"
, coalesce(sum( quantity_ordered * quoted_price),0) as "All_Qrts"
from prd_products PR
left join oe_orderDetails OD on OD.prod_id = PR.prod_id
left join oe_orderHeaders OH on OH.order_id = OD.order_id
where extract(year from order_date) = 2015 or order_date is null
group by catg_id
order by catg_id
;
```

CatgId	Qrt1_2015	Qrt2_2015	Qrt3_2015	Qrt4_2015	All_Qrts
APL	0.00	10000.00	4005.39	5724.98	19730.37
GFD	0.00	0.00	0.00	0.00	0.00
HD	0.00	0.00	0.00	695.65	695.65
HW	0.00	594.99	1945.90	1309.93	3850.82
MUS	0.00	0.00	314.60	0.00	314.60
PET	0.00	774.25	494.85	1194.12	2463.22
SPG	0.00	14079.95	16550.25	3069.00	33699.20

This gives us 7 rows. But we have 9 different product categories. Why did we not get 9 rows? See the demo file if you cannot figure this out. We want to display a row for every catg_id in the categories table whether or not we have a matching row in the products table.

Demo 15: This might not be considered a cross tab query because it does not have two grouping levels, but it is similar in using the Case technique.

Analyze quantity of items purchased by price . You would need to take care with the tests at each step to avoid missing some data.

```
select
    sum (case when quoted_price between 0.01 and 25
              then quantity_ordered
              else 0 end) as "Price 0.01-25"
  , sum (case when quoted_price between 25.01 and 100
              then quantity_ordered
              else 0 end) as "Price 25.01-100"
  , sum (case when quoted_price between 100.01 and 250
              then quantity_ordered
              else 0 end) as "Price 100.01- 250"
  , sum (case when quoted_price > 250
              then quantity_ordered
              else 0 end) as "Price > 250"
  , sum(quantity_ordered) as "Tot Quant"
from oe_orderDetails;
```

Price 0.01-25	Price 25.01-100	Price 100.01- 250	Price > 250	Tot Quant
506	115	240	89	950

Demo 16: A different layout for this query result. The expressions in the Select and the Group By are identical.

```
select
case
    when quoted_price between 0.01 and 25      then 'Price 0.01 - 25'
    when quoted_price between 25.01 and 100     then 'Price 25.01 - 100'
    when quoted_price between 100.01 and 250    then 'Price 100.01 - 250'
    when quoted_price > 250                   then 'Price over 250'
end as "Price Range"
,
sum(quantity_ordered) AS "Total Quantity"
from oe_orderDetails
Group by case
    when quoted_price between 0.01 and 25      then 'Price 0.01 - 25'
    when quoted_price between 25.01 and 100     then 'Price 25.01 - 100'
    when quoted_price between 100.01 and 250    then 'Price 100.01 - 250'
    when quoted_price > 250                   then 'Price over 250'
end
order by 1;
```

Price Range	Total Quantity
Price 0.01 - 25	506
Price 25.01 - 100	115
Price 100.01 - 250	240
Price over 250	89

Demo 17: You can use a CTE to simplify this a bit; this lets you write that case statements once only making the query easier to maintain.

```
With SalesAnalysis as (
    select
        case
            when quoted_price between 0.01 and 25      then 'Price 0.01 - 25'
            when quoted_price between 25.01 and 100    then 'Price 25.01 - 100'
            when quoted_price between 100.01 and 250   then 'Price 100.01 - 250'
            when quoted_price > 250                  then 'Price over 250'
        end as PriceRange
    , quantity_ordered
    from oe_orderDetails
)
select PriceRange as "Price Range"
, sum (quantity_ordered) as "Total Quantity"
from SalesAnalysis
group by PriceRange
order by PriceRange;
```

Suggestion: for the column aliases in the CTE , use simple aliases without blanks or specail characters. Then you do not need to use quoted aliases- which just make it harder to write the rest of the query.

Table of Contents

1. Scalar Subquery.....	1
2. Using a Scalar subquery	2
3. Putting a subquery in the Select	3
4. Putting a scalar subquery in an Order by clause.....	4

1. Scalar Subquery

A subquery is a Select expression that is placed within a query. The subquery is placed within parentheses.

A scalar subquery returns a single value (one row and one column) . This is still a table, but it is sometimes called a **scalar subquery expression** to emphasize that the result has a single value. The value of a scalar subquery is the value in the Select clause. That means the Select clause in the subquery can have only one expression. It can be used in places where the value of the expression can be used. You can test a scalar subquery with an equality test and you can use a scalar query in places where a single value is allowed. We have seen these starting in Unit 5; we used them as part of a Where clause filter. For example

Demo 01:

```
select emp_id
, name_last as "Employee"
from emp_employees
where dept_id =
  (select dept_id
   from emp_employees
   where emp_id = 162
  );
```

EMP_ID	Employee
162	Holme
200	Whale
207	Russ

Each employee has exactly one dept_id value so, if we have an employee 162 the subquery will return exactly one row and the Select in the subquery has only one column so this subquery returns a single value. That value can then be used by the Where clause in the main query with an equality test.

But what happens if we do not have an employee with id 162? In that case the subquery expression is null and no rows are returned by the main query. The predicate dept_id = null is never true.

If you are using subqueries in this way, it is your responsibility to ensure that the subquery returns a single row and a single column. Suppose you change the query as shown here:

Demo 02:

```
select emp_id
, name_last as "Employee"
from emp_employees
where dept_id =
  (select dept_id
   from emp_employees
  );
```

Now the subquery might return more than one row and in that case we would get an error message

SQL Error: ORA-01427: single-row subquery returns more than one row

Demo 03: Now consider the following query. Will this query run without error?

```
select emp_id, name_last as "Employee"
from emp_employees
where dept_id = (
    select dept_id
    from emp_employees
    where name_last = 'Green'
);
```

If we have no employees with the last name of Green, the query will run; the subquery returns a null and the main query returns no rows. If we have exactly one employee with the last name of Green then the subquery returns the department id for that employee and the main query returns all employees for that department. If we have more than one employee with the last name of Green, then the subquery is no longer a single-row subquery and you get an error message.

It is up to the person developing the query to ensure that in a query such as this that the subquery returns one or no rows. You don't get to say that when you set up the query we had only one employee with the name Green and it is not your fault that we now have two employees with that name. This query (demo3 and also demo 2) are poorly formed and it is your job to not write this type of query.

How can you ensure that a query returns exactly one row? The subquery could use a filter on a primary key; the subquery could return a single table aggregate; you could use where rownum <= 1 - but that might be poor logic. (As you develop more skills, you could create procedures that will handle the possibility of these types of error and do something better than simply fail- but that is for another class- CS 151P)

2. Using a Scalar subquery

The previous examples show scalar subqueries being used in a Where clause predicate. We can use scalar subqueries with tests for equality, inequality, between etc.

This is a demo of using a subquery in a between test; I am calculating the average price of products of a specific category and then finding products in that category within 10% of that average price.

Demo 04: This uses a variable for the category.

```
variable catg varchar2(5);
exec :catg := 'MUS';
```

You can see that this returns a single value. (Actually a very nice repeating decimal! SQL Developer)

```
select avg(prod_list_price)
from prd_products
where catg_id = :catg;
-----
```

AVG(Prod_List_Price)

13.8781818181818181818181818181818182

Demo 05: The main query uses a between test with a subquery for the lower and upper range values.

```
select prod_id, prod_name, prod_list_price
from prd_products
where catg_id = :catg
and prod_list_price between (
    select 0.9 * avg(prod_list_price)
    from prd_products
    where catg_id = :catg)
and (
    select 1.1 * avg(prod_list_price)
    from prd_products
    where catg_id = :catg)
;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
2746	B00000JWCM	14.50
2747	B000002I4Q	14.50

3. Putting a subquery in the Select

Demo 06: We can determine the median salary of all of the employees.

```
select median(salary)
from emp_employees;
```

MEDIAN(SALARY)

65000

Since that is a scalar subquery, we could put it in the Select clause.

```
select emp_id, name_last,
salary
, (
    select median(salary)
    from emp_employees
) as MedSalary
from emp_employees;
```

EMP_ID	NAME_LAST	SALARY	MEDSALARY
100	King	100000	65000
201	Harts	15000	65000
101	Koch	98005	65000
108	Green	62000	65000
205	Higgs	75000	65000
102	D'Haa	60300	65000
103	Hunol	69000	65000
104	Ernst	65000	65000
145	Russ	59000	65000
150	Tuck	20000	65000
155	Hiller	29000	65000
. . . rows omitted			

That is not too interesting since the subquery value is always the same. But we might want to know how far people's salaries are from the median.

Demo 07: Using the scalar subquery in an calculation in the select

```
select emp_id, name_last
, salary
, salary -(
    select median(salary)
    from emp_employees
) as "Over/Under"
from emp_employees;
```

EMP_ID	NAME_LAST	SALARY	Over/Under
100	King	100000	35000
201	Harts	15000	-50000
101	Koch	98005	33005
108	Green	62000	-3000
205	Higgs	75000	10000
102	D'Haa	60300	-4700

103 Hunol	69000	4000
104 Ernst	65000	0
145 Russ	59000	-6000
150 Tuck	20000	-45000
. . . rows omitted		

Do not try to put a non-scalar subquery in a Select clause. The following would produce an error (ORA-01427: single-row subquery returns more than one row).

```
Select emp_id, (select salary from emp_employees) as AllSalaries
from emp_employees;
```

4. Putting a scalar subquery in an Order by clause

This is pushing the idea a bit but we can put a scalar subquery in an Order by clause- not common but it works.

We want to sort employees by how far away they are from the median salary- either above or below (that is what abs does for us. I filtered for dept 30 simply to reduce the output display.

Demo 08:

```
select emp_id
, name_last
, salary
, salary -(select median(salary) from emp_employees) as "Over/Under"
from emp_employees
where dept_id in (30)
order by abs(salary -(select median(salary) from emp_employees))
```

DEPT_ID	EMP_ID	NAME_LAST	SALARY	Over/Under
30	109	Fiet	65000	0
30	203	Mays	64450	-550
30	108	Green	62000	-3000
30	110	Chen	60300	-4700
30	205	Higgs	75000	10000
30	206	Geitz	88954	23954
30	101	Koch	98005	33005
30	204	King	99090	34090

Table of Contents

1. Working up to Correlated Subqueries	1
2. Correlated Subqueries.....	3

1. Working up to Correlated Subqueries

We want to find out how many items (total quantity) are on each order we have. The first of these queries uses a scalar subquery in the Select but the output is not very interesting

Demo 01:

```
select customer_id, order_id
, (
    select sum(quantity_ordered)
    from oe_orderDetails  OD
) as "NumItems"
from oe_orderHeaders  OH
where rownum <=5;
```

CUSTOMER_ID	ORDER_ID	NumItems
409150	218	950
409150	223	950
409160	224	950
409160	225	950
403050	227	950

This seems to say that every order has a total quantity of 950 items; that does not look correct. The scalar subquery calculated the total number of items purchased on **all** orders. Look at the subquery. It says:

```
select sum(quantity_ordered)
from oe_orderDetails  OD
```

What we want for each order is the total number of items **for this order**. We have to tie that subquery calculation to the Order id in the Order headers table. This is done via a correlation- the table in the subquery (order details) is tied to/joined to/correlated with the order header row we are looking at. This is different.

The subquery has a Where clause that correlates the subquery to the main query. OH is an alias for a table in the main query.

```
( select sum(quantity_ordered)
    from oe_orderDetails  OD
    where OH.order_id = OD.order_id)
```

Demo 02: Now our result set makes more sense and is more useful

```
select customer_id, order_id
, ( select sum(quantity_ordered)
    from oe_orderDetails  OD
    where OH.order_id = OD.order_id) as "NumItemsPerOrder"
from oe_orderHeaders  OH
where customer_id IN ( 404100, 903000)
order by customer_id, order_id;
```

CUSTOMER_ID	ORDER_ID	NumItemsPerOrder
404100	303	1
404100	605	43
404100	2503	1
404100	2504	1
404100	2805	29
903000	306	2

903000	312	50
903000	313	1
903000	550	11
903000	551	20
903000	610	2
903000	2120	10
903000	2121	21
903000	3810	2

14 rows selected.

The nested sub query has a Where clause that refers to the outer query. That way we get the total of the quantity for this particular order. **This is a correlated subquery**. The table reference in the subquery is joined to the table references in the outer query.

The main query uses the order headers table and we get one row per order header row. We are not doing a group by clause.

Demo 03: Grouping with an outer join. It is not unusual to have more than one way to do a query.

```
select customer_id, OH.order_id ,sum(quantity_ordered) AS NumPerOrder
from oe_orderHeaders OH
left join oe_order_Details OD on OH.order_id = OD.order_id
where customer_id IN ( 404100, 903000)
group by Customer_id, OH.order_id
order by customer_id, OH.order_id
;
```

Demo 04: We might want to get the number of items per order and the cost of the order. This uses two subqueries.

```
select customer_id, order_id
, ( select SUM (quantity_ordered)
    from oe_orderDetails OD
    where OH.order_id = OD.order_id) as "NumPerOrder"
, ( select SUM (quantity_ordered * quoted_price)
    from oe_orderDetails OD
    where OH.order_id = OD.order_id) as "OrderCost"
from oe_orderHeaders OH
order by customer_id,order_id
;
```

CUSTOMER_ID	ORDER_ID	NumPerOrder	OrderCost
400300	378	10	4500
401250	106	1	255.95
401250	113	1	22.5
401250	119	10	225
401250	301	1	205
401250	552	10	157.3
401250	2506		
401890	112	2	99.98
401890	519	6	114.74
402100	114	5	625
...			

This query returns rows where we have an Order header row with no Detail rows. Why does it do that? How would you change the query to return only Order header row that have Detail rows?

2. Correlated Subqueries

With a non-correlated subquery, the inner query could work on its own. With a correlated subquery, the inner query refers to attributes found in the outer query. That means that the correlated subquery cannot be run independently. Logically, the outer query works on the first row and processes the subquery using the attributes in the first row; then the outer query works on the second row and then processes the subquery using the attributes in the second row; it then continues through the rest of the rows in the outer query reevaluating the subquery repeatedly.

Some of the following are correlated subqueries. Although a correlated subquery may seem inefficient, the efficiency depends on the optimizer for the database engine.

If you have the choice of solving a task with a correlated query or with non-correlated query, you should generally choose the non-correlated version.

Demo 05: This uses an aggregate function to get the average price for all products.

```
select round( Avg ( prod_list_price),2)
from prd_products;
```

AVG(Prod_List_Price)

117.67

Demo 06: This uses a subquery to get products that cost more than the average price for all products.

```
select prod_id, prod_list_price, catg_id
from prd_products
where prod_list_price > (
    select round( Avg ( prod_list_price),2)
    from prd_products
);
```

PROD_ID	PROD_LIST_PRICE	CATG_I
1000	125.00	HW
1010	150.00	SPG
1090	149.99	HW
1040	349.95	SPG
1050	269.95	SPG
1060	255.95	SPG
1160	149.99	HW
4567	549.99	PET
4568	549.99	PET
4569	349.95	APL
1120	549.99	APL
1125	500.00	APL
1126	850.00	APL
1130	149.99	APL

Demo 07: This uses grouping and an aggregate function to get the average price for each product category.

```
select round( Avg ( prod_list_price),2) as AvgPrice, catg_id
from prd_products
group by catg_id;
```

AVGPRICE	CATG_ID
123.17	PET
23.88	HD
13.88	MUS

67.64 HW
178.13 SPG
479.99 APL
8.75 GFD

Demo 08: We can use a **correlated subquery** to get the products that cost more than the average price for the **same type of products**. Notice that prd_products occurs in both the parent and the child query so we need to use table aliases in the join.

```
select prod_id, catg_id, prod_list_price
from prd_products Otr
where prod_list_price > (
    select Round( Avg ( prod_list_price ), 2 )
    from prd_products Inr
    where Otr.catg_id = Inr.catg_id )
order by catg_id, prod_list_price
;
```

PROD_ID	CATG_ID	PROD_LIST_PRICE
1125	APL	500.00
1120	APL	549.99
1126	APL	850.00
5000	GFD	12.50
5005	HD	45.00
1000	HW	125.00
1090	HW	149.99
1160	HW	149.99
2746	MUS	14.50
2747	MUS	14.50
2987	MUS	15.87
2984	MUS	15.87
2337	MUS	15.87
2234	MUS	15.88
2014	MUS	15.95
4567	PET	549.99
4568	PET	549.99
1060	SPG	255.95
1050	SPG	269.95
1040	SPG	349.95

20 rows selected.

Consider the subquery:

```
select AVG(prod_list_price)
from prd_products Inr
where Otr.catg_id = Inr.catg_id
```

This does not include a Group By clause. The subquery looks at only one value for catg_id - the one that matches the value for catg_id in the outer query for the current row being considered.

Since it is looking at only one category id, it will find only one average and so we can use the average in a filter of the type Price > average.

The one thing that should look odd about the subquery is that it refers to a table with an alias Otr that is not part of the subquery. That is where the "correlated" part of the correlated subquery comes in.

So let's go back to the main query. It gets one row from the product table and then tries to compare the price of that row to the average- what average; the average calculated by the subquery- which is the average for the same category id as that on the products table row we are looking at.

So the way to think of this query is

- working one row at a time through the products table
- for each row in the products table (one at a time) calculate the average price for that product id
- if the price for that product row is > average price for that category id, then return it.

This makes it sound like a very inefficient way to do this. Imagine we have a product tables of 50,000 rows of 10 different category ids. If the dbms actually carried the query out as I just described, that would mean calculated the average price 50,000 times. The dbms generally has a more efficient way - internally- to do this type of query. But logically you should think of the query as working with one row from the outer query processed against the subquery.

We want to find orders that are unusually high for a customer. Because we don't have a lot of rows, I defined this as an order that is more than 1.25 times the average order cost for that customer. This uses a CTE to assemble the data being used.

Demo 09: We will need the average order size by customer.

```
With OE_OrdExtTotal as (
    select OH.customer_id
    , order_id
    , sum ( OD.Quantity_ordered * OD.quoted_price) AS ordertotal
    from oe_orderHeaders OH
    join oe_orderDetails OD  using(order_id)
    group by OH.customer_id, order_id
)
select customer_id as "custid"
, to_char(avg(ordertotal), 9999.99) as "AvgPrice"
, to_char(1.25 * AVG(ordertotal), 9999.99) as "Cut_off"
from OE_OrdExtTotal
group by customer_id
order by customer_id;
```

custid	AvgPrice	Cut_off
400300	4500.00	5625.00
401250	173.15	216.44
401890	107.36	134.20
402100	1092.32	1365.40
403000	727.30	909.12
403010	1900.00	2375.00
403050	269.23	336.54
403100	218.84	273.55
. . . rows omitted		

Demo 10: Now we need a **correlated subquery** to compare a particular order with average orders for that customer.

```
With OE_OrdExtTotal as (
    select OH.customer_id
    , order_id
    , SUM ( OD.Quantity_ordered * OD.quoted_price) AS ordertotal
    from oe_orderHeaders OH
    join oe_orderDetails OD  using(order_id)
    group by OH.customer_id, order_id
)
```

```

select customer_id as "custid"
, order_id as "ordid"
, to_char( ordertotal, 9999.99) as "OrderCost"
from OE_OrdExtTotal OTR
where OrderTotal > 1.25 * (
    select AVG (ordertotal)
    from OE_OrdExtTotal INR
    where OTR.Customer_id = INR.Customer_id
    group by customer_id
)
order by customer_id;

```

custid	ordid	OrdTotal
401250	106	255.95
401250	119	225.00
402100	115	2305.00
403000	390	1400.00
403000	528	2629.00
403000	105	1205.40
403000	395	2925.00
403050	527	440.47
. . . rows omitted		

Let's start with the first customer shown here- customer_id 401250. This customer has 5 orders:

CUSTOMER_ID	ORDER_ID	ORDERTOTAL
401250	113	22.50
401250	552	157.30
401250	301	205.00
401250	119	225.00
401250	106	255.95

The cutoff for this customer is 216.44. Our query returns only two of this customer's orders- the ones where the order total is over this customer's cutoff.

401250	106	255.95
401250	119	225.00

Demo 11: To get customers with more than one order, we can use the count function for each customer_id as it occurs in the outer query; we do not need to qualify customer_id with Oe_order_headers in the inner query.

CUSTOMER_ID	CUSTOMER_NAME_LAST
401250	Morse
401890	Northrep
402100	Morise
403000	Williams
403050	Hamilton
403100	Stevenson
. . . rows omitted	

Demo 12: Here the correlated subquery returns a number which is used as a parameter to a case expression. We want to use the number of orders for this customer- not for all customers.

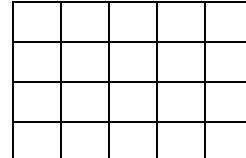
```
select customer_id, customer_name_last
, Case (
    select count(*)
    from oe_orderHeaders OH
    where OH.customer_id = CS.customer_id
)
when 0 then '... No orders'
when 1 then '1 order'
when 2 then '2 orders'
when 3 then '3 orders'
else '4+ orders'
end as NumberOfOrders
from cust_customers CS
;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	NUMBEROFORDERS
401250	Morse	4+ orders
401890	Northrep	2 orders
402100	Morise	3 orders
402110	Coltrane	... No orders
402120	McCoy	... No orders
402500	Jones	... No orders
403000	Williams	4+ orders
403010	Otis	1 order
403050	Hamilton	4+ orders
403100	Stevenson	4+ orders
... rows omitted		

Table of Contents

1.	Use a subquery in a From clause.....	1
2.	Counting in a subquery	2
3.	Joining a regular table and a subquery	3
3.1.	Joining subqueries.....	5
4.	Generating data in the subquery	7

Another shape we can have for a subquery is a multi-column, multi-row table



The subquery returns a virtual table that we can use in places where the query expects a table. We have seen this type of subquery in a CTE and in the Set operations. A Union join two subqueries.

The subquery could be used as a table expression in the From clause. In that case the subquery is sometimes called an in-line view. The subquery does not have to be the only table expression in the From clause; you can join the subquery result to a regular table to do a join.

1. Use a subquery in a From clause

When you embed a subquery in the From clause, it serves as a table expression.

- You should give the derived table an alias. Oracle does not always require a table alias but some other dbms do require this and Oracle allows the table alias.
- Each column needs a name; calculated columns need an alias
- The column names must be unique

Demo 01: This query is not very interesting but it shows using a subquery in the From clause; I am including a table alias- `tbl`.

I do not want to see you writing queries this simplistic. The subquery is providing no value.

```
select *
from (
    select *
    from emp_jobs
) tbl
;
```

This is also an inappropriate use of a subquery. Sometimes people get the idea that using subqueries makes your code more efficient- but that is not always the case.

```
select job_id, job_title, max_salary
from (
    select *
    from emp_jobs
    where max_salary is not null
) tbl
;
```

Demo 02: This is a query we did in unit 6 as a CTE because we could not use a column alias in the Select clause. We can also do this as an Inline view by taking the CTE subquery and using it as the From subquery.

```
With
ClNames as (
    select cl_state
    , cl_name_last || ' ' || cl_name_first as ClientName
    from vt_clients
)
select clientName || ' lives in ' || cl_state as "ClientInfo"
from ClNames
;
```

Using an inline view

```
select clientName || ' lives in ' || cl_state as "ClientInfo"
from (
    select cl_state
    , cl_name_last || ' ' || cl_name_first as ClientName
    from vt_clients
) ClNames
;
```

2. Counting in a subquery

You can use the COUNT (DISTINCT) feature to count the different numbers of shipping modes on all orders.

Demo 03: A standard COUNT DISTINCT- this does not count a null shipping_mode.

```
select COUNT (DISTINCT shipping_mode_id) as "num_diff_ship_modes"
from oe_orderHeaders;
num_diff_ship_modes
-----
6
```

Demo 04: If you wish to count nulls, you can use Coalesce to force a value to be counted.

```
select count (distinct coalesce (shipping_mode_id, 'none'))
as "num_diff_ship_modes"
from oe_orderHeaders;
num_diff_ship_modes
-----
7
```

Demo 05: Another way to do this is to use a subquery in the FROM clause to return the distinct shipping methods- which does return a "null" group, and then use the parent query to count those rows. Oracle does not insist on a table alias; other dbms do require a table alias with a subquery in the From clause

```
select count (*) as "num_diff_ship_modes"
from (
    select distinct shipping_mode_id
    from oe_orderHeaders
) tbl;
num_diff_ship_modes
-----
7
```

Demo 06: Using a CTE (Give your CTE meaningful names)

```
With
ShipModes as (
    select distinct shipping_mode_id
    from oe_orderHeaders
)
select count (*) "num_diff_ship_modes"
from ShipModes;
```

What if you want to count the number of distinct pairs of column values- for example, the number of pairs of shipping and order modes?

Demo 07: This approach yields an error :

```
select count (distinct shipping_mode_id, order_mode) as "num_diff_modes"
from oe_orderHeaders;
ORA-00909: invalid number of arguments
```

Demo 08: You can do this with a subquery.

```
select count (*) as "num_diff_modes"
from (
    select distinct shipping_mode_id, order_mode
    from oe_orderHeaders) tbl;
```

num_diff_ship_modes

13

3. Joining a regular table and a subquery

We want to display the number of employees in each department. We can start this as a simple query.

Demo 09: Since we have departments with no employees, we need an outer join. This is incorrect. Try to figure out the error before you go to the next demo.

```
select D.dept_id, count(*) as Row_Count
from emp_departments D
left join emp_employees E on D.dept_id = E.dept_id
group by D.dept_id;
```

DEPT_ID	ROW_COUNT
10	1
20	1
30	8
35	3
80	3
90	1
95	1
210	2
215	4

Demo 10: The previous query used count(*) and counted the nulled-rows that the outer join generates. So every department row returned a value of at least 1. We want to count employees- not rows.

```
select D.dept_id, count(E.emp_id) as Emp_Count
from emp_departments D
left join emp_employees E on D.dept_id = E.dept_id
group by D.dept_id;
```

DEPT_ID	EMP_COUNT
10	1

20	1
30	8
35	3
80	3
90	0
95	0
210	2
215	4

Demo 11: If we want to see the department names, we have to add dept_name to the group by clause. That is not a big deal since it is only one extra attribute.

```
select D.dept_id, D.dept_name, count(E.emp_id) as Emp_Count
from emp_departments D
left join emp_employees E on D.dept_id = e.dept_id
group by D.dept_id, D.dept_name
order by D.dept_id;
```

DEPT_ID	DEPT_NAME	EMP_COUNT
10	Administration	1
20	Marketing	1
30	Development	8
35	Cloud Computing	3
80	Sales	3
90	Shipping	0
95	Logistics	0
210	IT Support	2
215	IT Support	4

Demo 12: Alternately we could write a query that counts employees grouping by the department ID. This does not give us departments with no employees.

```
select dept_id, count(*) as EmpCount
from emp_employees E
group by dept_id;
```

DEPT_ID	EMPCOUNT
30	8
20	1
210	2
215	4
35	3
80	3
10	1

Demo 13: We could use that query and do an outer join to the department table. The subquery needs to supply an alias for the calculated (count) column because we want to refer to it.

Note that the subquery is enclosed in parentheses and given a table alias. Then the join is done as we usually do joins: on D.dept_id = EC.dept_id

We can use the subquery to contain the details of the aggregation.

```
select D.dept_id, D.dept_name, EC.EmpCount
from emp_departments D
left join (
    select dept_id, count(*) as EmpCount
    from emp_employees
    group by dept_id) EC on D.dept_id = EC.dept_id ;
```

DEPT_ID	DEPT_NAME	EMPCOUNT
10	Administration	1
20	Marketing	1
30	Development	8
35	Cloud Computing	3
80	Sales	3
90	Shipping	
95	Logistics	
210	IT Support	2
215	IT Support	4

Demo 14: You could also write this as a CTE. In general you have a choice between using a CTE or an in-line view. If you need to use the subquery expression more than once in the same query- use the CTE. You can also take a CTE query and rewrite it using an in-line view.

```
With deptEmpCount as (
    select dept_id, count(*) as EmpCount
    from emp_employees
    group by dept_id
)
select D.dept_id, D.dept_name, EC.EmpCount
from emp_departments D
left join deptEmpCount EC on D.dept_id = EC.dept_id;
```

This demo does seem like a bit of over kill to avoid writing group by d.dept_id, d.dept_name. The following query wants to display three extra fields along with the aggregates. And you might just feel that it makes more sense to aggregate on the dept id only.

Demo 15:

```
select D.dept_id, D.dept_name
, l.loc_city, l.loc_state_province
, count(emp_id) as EmpCount
from emp_departments D
join emp_locations L on D.loc_id = l.loc_id
left join emp_employees E on d.dept_id = e.dept_id
group by D.dept_id, D.dept_name, l.loc_city, l.loc_state_province;
```

3.1. Joining subqueries

Demo 16: Using an outer join to a subquery- used as a table expression. And it has one grouping key

```
select D.dept_id, D.dept_name
, l.loc_city, l.loc_state_province
, coalesce(EmpCount,0) as EmpCount
from emp_departments D
join emp_locations L on D.loc_id = l.loc_id
left join (
    select dept_id, count(emp_id) as EmpCount
    from emp_employees E
    group by dept_id) EC on D.dept_id = EC.dept_id;
```

This uses two subqueries as data sources.

Demo 17: How many employees do we have in each department and what percent is that of all employees?

Use each of these as a virtual table in the FROM clause to get the percent of each department over the entire employee table. Notice that we do not need a joining clause since the overall count has only one return row.

```
select
    dept_id
    , dept_count
    , round((dept_count / Count_All),2) AS Percnt
from
    (select dept_id, count(1) AS dept_count /* get count by department */
        from emp_employees
        group by dept_id) vt1, /* get total count for all employees */
    (select count(*) AS Count_All
        from emp_employees) vt2
order by Dept_id;
```

DEPT_ID	DEPT_COUNT	PERCNT
10	1	.05
20	1	.05
30	8	.36
35	3	.14
80	3	.14
210	2	.09
215	4	.18

Demo 18: To get a percent value, multiply by 100 .You can also do some formatting to get a value that looks more like a percent.

```
select
    dept_id
    , dept_count
    , round((1.00 * dept_count / Count_All),2) AS Percnt
    , to_char ((round((dept_count / Count_All),2) * 100), 99.9) || '%' AS Percnt
from
    (select dept_id, count(1) AS dept_count
        from emp_employees
        group by dept_id) vt1,
    (select count(*) AS Count_All
        from emp_employees) vt2
order by Dept_id
; 
```

DEPT_ID	DEPT_COUNT	PERCNT	PERCNT
10	1	.05	5.0%
20	1	.05	5.0%
30	8	.36	36.0%
35	3	.14	14.0%
80	3	.14	14.0%
210	2	.09	9.0%
215	4	.18	18.0%

Demo 19: Using a CTE

```
With DeptCount as
    (select dept_id, count(1) AS dept_count
        from emp_employees
        group by dept_id)
```

```

AllCount as
  (select count(*) AS Count_All
   from emp_employees)
select
  dept_id
, dept_count
, to_char ((Round((dept_count / Count_All),2) * 100), 99.9) || '%' AS Percnt
from DeptCount
cross join AllCount
order by Dept_id
;

```

Demo 20: One way to find customers who bought both an appliance and a houseware item.

(oe_customer_orders is a view you should have created earlier)

The first subquery picks up the appliances

The second subquery picks up the houseware items

The join of the two subqueries checks that we are looking at the same customer id

```

select distinct tblapl.custid
from ( select CustID
       from oe_customer_orders
       where category ='APL' )  tblapl
join ( select CustID
       from oe_customer_orders
       where Category ='HW' )  tblhw
on tblapl.custid = tblhw.custid;

```

4. Generating data in the subquery

Demo 21: We decide we want to display descriptive literals for various salary ranges. We could do this with a case expression. But we can also generate the rating data with a union

```

select 'under paid' as catg, 0 as low, 49999.99 as high from dual
Union all
  select 'medium paid' as catg, 50000.00 as low, 79999.99 as high from dual
Union all
  select 'over paid' as catg, 80000.00 as low, 9999999.99 as high from dual;

```

CATG	LOW	HIGH
under paid	0	49999.99
medium paid	50000	79999.99
over paid	80000	9999999.99

Demo 22: Now join that union to the employees table

```

select emp_id, name_last,salary, catg
from emp_employees E
join (
  select 'under paid' as catg, 0 as low, 49999.99 as high from dual
  Union all
    select 'medium paid' as catg, 50000.00 as low, 79999.99 as high from dual
  Union all
    select 'over paid' as catg, 80000.00 as low, 9999999.99 as high from dual
  ) Ratings on E.salary between Ratings.low and Ratings.high
order by salary;

```

Selected rows

EMP_ID	NAME_LAST	SALARY	CATG
201	Harts	15000	under paid
150	Tuck	20000	under paid
155	Hiller	29000	under paid
207	Russ	30000	under paid
145	Russ	59000	medium paid
110	Chen	60300	medium paid
205	Higgs	75000	medium paid
146	Partne	88954	over paid
206	Geitz	88954	over paid
162	Holme	98000	over paid

Demo 23: Again- you can use a CTE instead of a subquery

```
With ratings as (
  select 'under paid' as catg, 0 as low , 49999.99 as high from dual
  Union all
  select 'medium paid' , 50000.00 , 79999.99 from dual
  Union all
  select 'over paid' , 80000.00 , 9999999.99 from dual )
select emp_id, name_last,salary, catg
from emp_employees E
join ratings R on E.salary between R.low and R.high
order by salary;
```

Demo 24: You could do this with a case

```
select emp_id, name_last,salary
, case
  when salary between 0 and 49999.99 then 'under paid'
  when salary between 50000.00 and 79999.99 then 'medium paid'
  when salary between 80000.00 and 9999999.99 then 'over paid'
  end as catg
from emp_employees E
order by salary;
```

Demo 25: we may want just the aggregates

```
select catg, count(*)as NumEmployees
from emp_employees E
join (
  select 'under paid' as catg, 0 as low, 49999.99 as high from dual
  Union all
  select 'medium paid' as catg, 50000.00 as low, 79999.99 as high from dual
  Union all
  select 'over paid' as catg, 80000.00 as low, 9999999.99 as high from dual
) Ratings on E.salary between Ratings.low and Ratings.high
group by catg;
```

CATG	NUMEMPLOYEES
under paid	4
medium paid	11
over paid	7

The emp_employees table has the attribute salary defined as nullable. Which of the above queries will report back employees with no salary value? How would we get those employees into the reports?

Table of Contents

1. Subqueries versus Joins and Distinct	1
2. OUTER JOINS- what does that null mean?.....	4

This is mostly review; it can help to go back over material several times. Something that did not make sense two weeks ago might click now.

1. Subqueries versus Joins and Distinct

We want to know the names of all of our customers who currently have an order in our order headers table. Since we need the customer name we need the customer table. We only want customers with orders so we use an inner join to the order headers table. We were not asked to check that the order has any detail lines so we do not need the order details table.

Demo 01: Using an Inner join

```
select CS.customer_name_last, CS.customer_name_first
from cust_customers CS
join oe_orderHeaders OH on CS.customer_id = OH.customer_id
order by CS.customer_name_last, CS.customer_name_first
;

-- sample rows-
```

but we do have 97 rows and only 33 customers. It is pretty clear that we have some duplicates here.

CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
Adams	Abigail
Adams	Abigail
Button	D. K.
Button	D. K.
Button	D. K.
Clay	Clem
Day	David
Hamilton	Alexis
Hamilton	Alexis
Hamilton	Alexis
Martin	Jane
. . . .	

What we commonly do when we have duplicate output is add the Distinct modifier.

Demo 02: Using distinct and the inner join

```
select distinct CS.customer_name_last, CS.customer_name_first
from cust_customers CS
join oe_orderHeaders OH on CS.customer_id = OH.customer_id
order by customer_name_last, customer_name_first
;
```

CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
Adams	Abigail
Button	D. K.
Clay	Clem
Day	David
Hamilton	Alexis
Martin	Jane
Martin	Joan

Mazur	Barry
McGold	Arnold
Morise	William
Morris	William
Morse	Samuel
Northrep	William
Olmsted	Frederick
Otis	Elisha
Prince	
Stevenson	James
Williams	Al
Williams	Sally

19 rows selected.

That looks better- but this is hiding some data. We know that we could have customers with the same name.

Demo 03: Adding the customer id to the Select. Using distinct and the inner join; we get 2 more rows. Some customers do not have any orders. If you are going to use Distinct, you might want to include a PK

```
select distinct CS.customer_name_last, CS.customer_name_first, CS.customer_id
from cust_customers CS
join oe_orderHeaders OH on CS.customer_id = OH.customer_id
order by customer_name_last, customer_name_first
;
```

CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST	CUSTOMER_ID
Adams	Abigail	915001
Button	D. K.	004100
Clay	Clem	008770
Day	David	005000
Hamilton	Alexis	003050
Martin	Jane	009160
Martin	Joan	009150
Mazur	Barry	009030
McGold	Arnold	300300
McGold	Arnold	900300
McGold	Arnold	903000
Morise	William	002100
Morris	William	004950
Morse	Samuel	001250
Northrep	William	001890
Olmsted	Frederick	004000
Otis	Elisha	003010
Prince		009190
Stevenson	James	003100
Williams	Al	004900
Williams	Sally	003000

21 rows selected.

Now we get several more rows. In this case we could find them by inspection. We have three customers with the name Arnold. McGold . We do not know if these are three different people or one or two people who registered with us more than one time. The person who develops the queries cannot make that decision.

But we did expose the customer ID- maybe we should not do that. We are to display the name only.

Demo 04: We could hide the query with the custID inside a subquery and use it as the data source.

This will give us the rows we expect with Arnold McGold showing up three times. .

```
select customer_name_last, customer_name_first
from (
    select distinct CS.customer_name_last, CS.customer_name_first
    , CS.customer_id
    from cust_customers CS
    join oe_orderHeaders OH on CS.customer_id = OH.customer_id
) CS
;

```

CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
Stevenson	James
Morise	William
Otis	Elisha
Williams	Al
McGold	Arnold
Morse	Samuel
Day	David
Prince	
Northrep	William
Williams	Sally
Clay	Clem
Martin	Jane
McGold	Arnold
Morris	William
Adams	Abigail
Hamilton	Alexis
Olmsted	Frederick
Button	D. K.
Mazur	Barry
McGold	Arnold
Martin	Joan

21 rows selected.

Demo 05: Another solution would be to have a subquery that gets us the customer ID from the order headers table- that gives us customer with orders and then use that subquery as a filter for the outer query which uses the customers table to get the names.

```
select customer_name_last, customer_name_first
from cust_customers CS
where customer_id in
    (select customer_id
     from oe_orderHeaders )
;
```

Which of these two approaches is better? That depends on several things. Optimizers may make different plans based on the size of the tables, the presence or absence of indexes, and the distribution of data. So having a rule such as technique A is always better than technique B is not generally true.

If your dbms always does a sort if you use Distinct, then the second version might be more efficient. A sort, including a hidden sort, is generally expensive and you should avoid it when possible.

Of course there is another question- what are the business needs of the person who asked for the result (Remember users never want queries- they want results). Maybe the user really wants to see the different names of people who have orders- marketing people do odd types of sales analysis and maybe they are looking for name patterns!

2. OUTER JOINS- what does that null mean?

Suppose we write a query that shows the various locations for our employee departments. We want to include any department that might not have an assigned location.

Demo 06: An outer join from department to locations.

```
select
    dept_id
    , dept_name
    , loc_type as LocationType
from emp_departments D
left join emp_locations L on D.loc_id = L.loc_id
order by dept_id
;

dept_id dept_name          LocationType
----- -----
    10 Administration        office
    20 Marketing             warehouse
    30 Development            warehouse
    35 Cloud Computing
    80 Sales                  warehouse
    90 Shipping
    95 Logistics
    210 IT Support            office
    215 IT Support

(9 row(s) affected)
```

This has several rows with a null for the location type - what can that null mean in terms of the query. Try to think of the possibilities; you may have to look at the table create statements to see how this could happen.

One possibility is that the department does not have an assigned location and the dbms generated a null record due to the outer join.

If you look at the definition of the location table, the loc_type was defined as nullable- so it might be that the department has a location but the location type is null. The result does not differentiate between these two.

If you look at the current data set, you can see that dept 35 has no assigned location; dept 215 has an assigned location, but that location has a null for the location type.

Demo 07: Let's show more columns so that we can work on this.

```
select
    dept_id
    , dept_name
    , l.loc_id
    , loc_type as LocationType
    , loc_city as City
    , loc_state_province as "State/Province"
from emp_departments D
left join emp_locations L on D.loc_id = L.loc_id
order by dept_id
;

dept_id dept_name          loc_id LocationType      City           State/Province
----- -----
    10 Administration        1560   office       San Francisco    California
    20 Marketing             1400   warehouse   Southlake        Texas
    30 Development            1500   warehouse   South San Francisco  California
```

35 Cloud Computing				
80 Sales	1400	warehouse	Southlake	Texas
90 Shipping				
95 Logistics				
210 IT Support	1800	office	Toronto	Ontario
215 IT Support	2700		Munich	Bavaria

9 rows selected.

What we want as a result is the following:

dept_id	dept_name	LocationType	City	State/Province
10	Administration	office	San Francisco	California
20	Marketing	warehouse	Southlake	Texas
30	Development	warehouse	South San Francisco	California
35	Cloud Computing	No Site	No Site	No Site
80	Sales	warehouse	Southlake	Texas
90	Shipping	No Site	No Site	No Site
95	Logistics	No Site	No Site	No Site
210	IT Support	office	Toronto	Ontario
215	IT Support	Unknown type	Munich	Bavaria

Demo 08: We commonly use coalesce in these situations.

```

select
    dept_id
    , dept_name
    , coalesce(loc_type, 'Unknown type') as LocationType
    , coalesce(loc_city, 'No Site') as City
    , coalesce(loc_state_province, 'No Site') as "State/Province"
from emp_departments D
left join emp_locations L on D.loc_id = L.loc_id
order by dept_id
;

```

dept_id	dept_name	LocationType	City	State/Province
10	Administration	office	San Francisco	California
20	Marketing	warehouse	Southlake	Texas
30	Development	warehouse	South San Francisco	California
35	Cloud Computing	Unknown type	No Site	No Site
80	Sales	warehouse	Southlake	Texas
90	Shipping	Unknown type	No Site	No Site
95	Logistics	Unknown type	No Site	No Site
210	IT Support	office	Toronto	Ontario
215	IT Support	Unknown type	Munich	Bavaria

This works for the City and State- these are not null attributes in the location table definition so they only way there are null is if the department does not have a location. But the location type has two ways to be null. We want to handle that circumstance on the location table level. If the field in the location table is null then use the alternate display.

```

select loc_id
    , coalesce(loc_type, 'UnknownType') as loc_type
    , loc_city, loc_state_province
from emp_locations
;

```

If we use this as the data source- rather than using the location table directly, then in the outer query location type can be treated the same as the City and state. So just substitute the above query as a subquery in our outer join.

Demo 09: Using the subquery as a data source

```
select
    dept_id
    , dept_name
    , coalesce(loc_type, 'No Site') as LocationType
    , coalesce(loc_city, 'No Site') as City
    , coalesce(loc_state_province, 'No Site') as "State/Province"
from emp_departments D
left join (
    select loc_id
    , coalesce(loc_type, 'UnknownType') as loc_type
    , loc_city, loc_state_province
    from emp_locations) L on D.loc_id = L.loc_id
order by dept_id
;
```

dept_id	dept_name	LocationType	City	State/Province
10	Administration	office	San Francisco	California
20	Marketing	warehouse	Southlake	Texas
30	Development	warehouse	South San Francisco	California
35	Cloud Computing	No Site	No Site	No Site
80	Sales	warehouse	Southlake	Texas
90	Shipping	No Site	No Site	No Site
95	Logistics	No Site	No Site	No Site
210	IT Support	office	Toronto	Ontario
215	IT Support	UnknownType	Munich	Bavaria

Table of Contents

1. Subqueries with the Exists Predicate	1
2. Correlation and the Exists Predicate	5
3. Subqueries that produce report style output	8

Exists is a predicate that accepts a subquery as an argument and returns either True or False. Exists is a test for existence- does the subquery return any rows. Sometimes you do not care what is in the rows returned by a subquery- only if there are any such rows. For example- does this customer have any orders? We might not care about how many orders the customer might have or what is on those orders- we only want to know **if there are any orders for the customer**. Suppose the customer does have one or more orders- as soon as we find one order, we have the answer and we do not need to look any further. So using Exists can be a more efficient way to return data.

Exists is commonly used with correlated subqueries but we will start with some non-correlated subqueries to see how this predicate works. This use of Exists is only to start with a simpler syntax. **Almost all uses of Exists in realistic queries use correlated subqueries.**

1. Subqueries with the Exists Predicate

This is another version of the zoo table (zoo_ex). The SQL for this is in the posted SQL. I have inserted blank lines between the animal types in the display.

ID	AN_TYPE	AN_PRICE
4	bird	100
5	bird	50
15	bird	80
17	bird	80
8	cat	10
16	cat	
1	dog	80
18	dog	80
19	dog	10
6	fish	10
11	fish	10
13	fish	10
3	lizard	
7	lizard	50
12	lizard	50
9	snake	50
10	snake	
14	snake	25
2	turtle	

EXISTS is an operator that is used with subqueries. If the subquery brings back at least one row, then the Exists operator has the value True. If the subquery does not bring back any rows, then the Exists has the value False. Since we do not need any specific values brought back from the inner query, we can use Select 'X'- or any other literal for efficiency. All we need to know is if the subquery brings back any rows or not. Even if the subquery brings back rows that have null values, those rows still exist. The Exists operator can be negated.

Demo 01: What does Exists do?

```
select 'Found' from dual
where exists (
    select 1
    from zoo_ex);
```

This first query might look strange since we are using the dual table in the outer query- but in Oracle you can use this technique to write a valid query without worrying about the table.. And I want to emphasize what Exists actually does. If you run this query after you have created the table but before you have entered any rows, the result is no rows.

The subquery does not return any rows since the table is empty. Therefore Exists evaluates as False. The outer query is now: select 'Found' From dual Where False. So the outer query returns no rows.

```
---- no rows selected-
```

If you have entered even a single row, then the query returns the following result. Exists lets the query know if there were any rows at all in the subquery's result set.

```
'FOUN
-----
Found

1 row selected.
```

The subquery returns one or more rows. Therefore Exists evaluates as True. The outer query is now: Select 'Found' From dual Where True. So the outer query returns all the rows in its data source.

Demo 02: Now we make the subquery slightly more interesting by adding a filter. We do have at least one row with a snake returned by the subquery.

```
select 'Found'
from dual
where exists (
    select 1
    from zoo_ex
    where an_type='snake')
;
```

```
'FOUN
-----
Found

1 row selected.
```

Demo 03: But if we filter for penguin- we get no rows in the subquery and Exists returns False so the outer query returns no rows.

```
select 'Found'
from dual
where exists (
    select 1
    from zoo_ex
    where an_type='penguin')
;
```

```
---- no rows selected-
```

If we look at the rows for an_type = 'snake' we will see that there is a row where the price is null. Suppose our subquery filters for rows for snakes with a null price and the subquery returns the price attribute. Even though that price is null, we do get a row returned.

Demo 04: Snakes with a null price

```
select an_price
from zoo_ex
where an_type='snake'
and an_price is null ;
```

```
an_price
-----
1 row selected.
```

Demo 05: So if we use that subquery with Exists, the outer query returns its rows

```
select 'Found'
from dual
where exists (
    select an_price
    from zoo_ex
    where an_type = 'snake'
    and an_price is null) ;
```

```
'FOUN
-----
Found
```

We really do not care what attribute is used in the select in the subquery and we commonly use a literal- such as 1 or 'x'. You may see people use Select * in the subquery- and most dbms will rewrite that for you. But using Select an_price or Select * suggests that you want all of the attributes or a specific attribute value and that is not the way that the Exists operator works.

Demo 06: If we run the previous query with a filter for 'bird' we get no rows returned since we do not have any birds with a null price. This is not specific to nulls; you could change the filter to an_price = 80 or to an_price = 1250 to see if we have any rows matching that test. The important thing to remember with exists is that it returns True or False and not the sets of rows from the subquery.

```
select 'Found'
from dual
where exists (
    select an_price
    from zoo_ex
    where an_type = 'bird'
    and an_price is null)
;
```

Now we will start to use the zoo_ex table in the outer query also.

Demo 07: If we have any birds that cost less than 75.00 then display all of the birds.

Since we have at least one row that meets that test, all of the rows for birds are returned by the outer query.

```
select *
from zoo_ex
where an_type='bird'
and exists (
    select 1
    from zoo_ex
    where an_type='bird'
    and an_price < 75
)
;
```

ID	AN_TYPE	AN_PRICE
4	bird	100
5	bird	50
15	bird	80
17	bird	80

Change the filter test to `an_price < 15` and no rows are returned.

This might seem rather impractical but suppose your tables dealt with a manufacturing process for medicines and these was a 'quality' attribute which indicated how well this batch passed its quality tests. You might want to run a query that says if any batch run on a particular day failed its tests then display all of the batches run that day.

You can negate the exists test- but the logic is harder to follow

Demo 08: Negating the Exists; try this and then try changing the filter to `an_price < 15` and run that one also.

```
select *
from zoo_ex
where an_type='bird'
and NOT exists (
    select 1
    from zoo_ex
    where an_type='bird'
    and an_price < 75
);
```

Demo 09: Compare the Exists to an IN test- this returns the birds which cost less than 75 and does not need the subquery approach.

```
select *
from zoo_ex
where an_type='bird'
and id in (
    select id
    from zoo_ex
    where an_type='bird'
    and an_price < 75
);
```

ID	AN_TYPE	AN_PRICE
5	bird	50

1 row selected.

Now take queries 7 and 8 and run them testing for snake- which has a null price. How does this affect the output? It is always a good idea to try to figure out what these will do before you try them. Just running queries is not as helpful. And you can run the subquery part by itself to help understand what happens with nulls.

Demo 10: A: Snakes less than 75 and B: snakes less than 15

```
-- demo 10A:
select *
from zoo_ex
where an_type='snake'
and exists (
    select 1
    from zoo_ex
    where an_type='snake'
    and an_price < 75
);
```

```
-- demo 10B:
select *
from zoo_ex
where an_type='snake'
and exists (
```

```
select 1
from zoo_ex
where an_type='snake'
and an_price < 15
);
```

Demo 11: Negating these : A: Snakes less than 75 and then B: snakes less than 15

```
-- demo 11A:
select *
from zoo_ex
where an_type='snake'
and NOT exists (
    select 1
    from zoo_ex
    where an_type='snake'
    and an_price < 75
);

-- Demo 11B:
select *
from zoo_ex
where an_type='snake'
and NOT exists (
    select 1
    from zoo_ex
    where an_type='snake'
    and an_price < 15
);
```

2. Correlation and the Exists Predicate

When we add correlation then these can be used to solve more problems. The following queries are correlated queries so that we are checking if there are any rows in the subquery that are related to the current row in the parent query. Remember that if you want to find subquery information for **this** customer or for **this** order, you generally will need to correlate.

This is a much more common use of Exists.

We are now using the tables in the altgeld_mart database.

Which customers have an order with a date in October 2015? We don't care how many orders they had or what they bought- we only want a list of the customers with an order in that month.

Demo 12: We can solve this with an Exists and a correlated subquery. We use a correlated subquery because- for each customer we want to know if there are any orders **for this** customer.

```
select customer_id, customer_name_last, customer_name_first
from cust_customers
where EXISTS (
    select 'X'
    from oe_orderHeaders
    where oe_orderHeaders.customer_id = cust_customers.customer_id
    and extract( MONTH from order_date) = 10
    and extract(Year from order_date) = 2015)
; 
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
401250	Morse	Samuel
403000	Williams	Sally
403050	Hamilton	Alexis
403100	Stevenson	James
404950	Morris	William

We can also solve this with an In subquery and with an inner join.

-- demo 12B

```
select customer_id, customer_name_last, customer_name_first
from cust_customers
where customer_id in (
    select customer_id
    from oe_orderHeaders
    where extract( MONTH from order_date) = 10
    and extract(Year from order_date) = 2015);
```

-- demo 12C This query may be less efficient if there is a sort done to handle the distinct.

```
select distinct customer_id, CS.customer_name_last, CS.customer_name_first
from cust_customers CS
join oe_orderHeaders OH using (customer_id)
where extract( MONTH from order_date) = 10
and extract(Year from order_date) = 2015;
```

-- demo 12D This query can return multiple rows for the same customer.

```
select customer_id, CS.customer_name_last, CS.customer_name_first
from cust_customers CS
join oe_orderHeaders OH using (customer_id)
where extract( MONTH from order_date) = 10
and extract(Year from order_date) = 2015;
```

Demo 13: The following requires a correlated query, since we need to check not if there is any order at the discount- but if **this customer** has an order with this discount. Display customers who purchased at item at a discount greater than 10% of the list price .

```
select customer_id
, customer_name_last
, customer_name_first
from cust_customers
where EXISTS (
    select 1
    from oe_orderHeaders OH
    join oe_orderDetails OD using(order_id)
    join prd_products PR using(prod_id)
    where (prod_list_price - quoted_price)/prod_list_price > 0.1
    and cust_customers.customer_id = OH.customer_id)
order by customer_id;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	CUSTOMER_NAME_FIRST
400300	McGold	Arnold
401250	Morse	Samuel
402100	Morise	William
403000	Williams	Sally
403050	Hamilton	Alexis
403100	Stevenson	James
404000	Olmsted	Frederick
... rows omitted		

What do we get when we negate the Exists? Remember that negative logic requires extra thought. Do we want customers who have never gotten the discount or do we want customers who have a purchase which was not at the discount.

Demo 14:

```
select customer_id
, customer_name_last
, customer_name_first
from cust_customers
where NOT EXISTS (
    select 1
    from oe_orderHeaders OH
    join oe_orderDetails OD using(order_id)
    join prd_products PR using(prod_id)
    where (prod_list_price - quoted_price)/prod_list_price > 0.1
    and cust_customers.customer_id = OH.customer_id)
order by customer_id;
```

Demo 15:

```
select customer_id
, customer_name_last
, customer_name_first
from cust_customers
where EXISTS (
    select 1
    from oe_orderHeaders OH
    join oe_orderDetails OD using(order_id)
    join prd_products PR using(prod_id)
    where not((prod_list_price - quoted_price)/prod_list_price > 0.1 )
    and cust_customers.customer_id = OH.customer_id)
; 
```

Demo 16: Display any products for which we have an order with a quantity order of 10 or more. Note that with this query we do not care how many such orders we have only whether or not we have any such orders **for this product**.

```
select prod_id, prod_name
from Prd_products PR
where EXISTS (
    select 'X'
    from oe_orderDetails OD
    where PR.prod_id = OD.prod_id
    and quantity_ordered >= 10)
; 
```

PROD_ID	PROD_NAME
<hr/>	
1010	Weights
1020	Dartboard
1030	Basketball
1040	Treadmill
1050	Stationary bike
1060	Mountain bike
1070	Iron
1140	Bird cage- simple
1150	Cat exerciser
4576	Cosmo cat nip

Demo 17: Which customers bought both an APL product and a HW product? We solved this problem earlier with two IN tests and again with a join.

```
select cust_customers.customer_id
, customer_name_last
from cust_customers
where EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = Cust_customers.customer_id
    and category ='APL')
and EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = Cust_customers.customer_id
    and category ='HW')
;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST
404950	Morris
404100	Button
903000	McGold
409150	Martin
403000	Williams
900300	McGold
402100	Morise
409030	Mazur

How would you use exists to find

- customers who bought either an APL product or a HW product
- customers who bought an APL product but not a HW product?

3. Subqueries that produce report style output

These queries produce tables- that is always the case but these look like very simple reports

Demo 18: This query produces a simple report as to customer purchase in three categories

```
select CS.customer_id
, CS.customer_name_last
, case when EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = CS.customer_id
    and category ='APL')
then 'Appliances '
else ' --- '
End as " "
, case when EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = CS.customer_id
    and category ='HW')
then 'Housewares '
else ' --- '
End as " "
, case when EXISTS (
    select 'X'
```

```

from oe_customer_orders
where oe_customer_orders.custID = CS.customer_id
and category ='PET')
then 'PetSupplies '
else ' --- '
End as " "
from cust_customers CS;
-- Selected rows

```

CUSTOMER_ID	CUSTOMER_NAME_LAST			
401250	Morse	---	Housewares	---
401890	Northrep	---	Housewares	---
402100	Morise	Appliances	Housewares	PetSupplies
402110	Coltrane	---	---	---
402120	McCoy	---	---	---
402500	Jones	---	---	---
403000	Williams	Appliances	Housewares	PetSupplies
403010	Otis	Appliances	---	---
403050	Hamilton	---	Housewares	PetSupplies
403100	Stevenson	---	---	PetSupplies
403500	Stevenson	---	---	---
403750	O'Leary	---	---	---
403760	O'Leary	---	---	---
903000	McGold	Appliances	Housewares	PetSupplies

Demo 19: A minor change in the query gives you a single column of purchase categories.

```

select CS.customer_id
, CS.customer_name_last
, case when EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = CS.customer_id
    and category ='APL')
then 'Appliances '
else ''
End
|| case when EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = CS.customer_id
    and category ='HW')
then 'Housewares '
else ''
End
|| case when EXISTS (
    select 'X'
    from oe_customer_orders
    where oe_customer_orders.custID = CS.customer_id
    and category ='PET')
then 'PetSupplies'
else ''
End as PurchaseAreas
from cust_customers CS;
-- Selected rows

```

CUSTOMER_ID	CUSTOMER_NAME_LAST	PURCHASEAREAS
401250	Morse	Housewares
401890	Northrep	Housewares
402100	Morise	Appliances Housewares PetSupplies
402110	Coltrane	

402120 McCoy			
402500 Jones			
403000 Williams	Appliances	Housewares	PetSupplies
403010 Otis	Appliances		
403050 Hamilton	Housewares	PetSupplies	
403100 Stevenson	PetSupplies		
403500 Stevenson			
403750 O'Leary			
403760 O'Leary			
903000 McGold	Appliances	Housewares	PetSupplies
404000 Olmsted	PetSupplies		

Table of Contents

1. Any and All Operators	1
2. Using the All Operator.....	2
3. Finding the best(?) using the AltgeldMart tables.....	3
4. Using the Any Operator.....	5

1. Any and All Operators

The Any and All operators accept a list as an argument; you can compare the value returned by Any or All using the relational operators =, !=, >, <, >=, <=. The list is provided by a subquery.

Create a view that returns only the rows where the an_price is not null;

```
Create view zoo_ex_notnull as (
    select id, an_type, an_price
    from zoo_ex
    where an_price is not null);
```

For reference, these are the rows in the zoo_ex table

ID	AN_TYPE	AN_PRICE
1	dog	80
2	turtle	
3	lizard	
4	bird	100
5	bird	50
6	fish	10
7	lizard	50
8	cat	10
9	snake	50
10	snake	
11	fish	10
12	lizard	50
13	fish	10
14	snake	25
15	bird	80
16	cat	
17	bird	80
18	dog	80
19	dog	10

For reference, these are the rows in zoo_ex_notnull.

ID	AN_TYPE	AN_PRICE
1	dog	80
4	bird	100
5	bird	50
6	fish	10
7	lizard	50
8	cat	10
9	snake	50
11	fish	10
12	lizard	50
13	fish	10
14	snake	25
15	bird	80
17	bird	80
18	dog	80
19	dog	10

2. Using the All Operator

The All operator is useful for finding the rows with largest value in a table including ties. We will start with a few examples using the view above to avoid issues with nulls.

Demo 01: We might want to find the most expensive animal.

```
select *
from zoo_ex_notnull
where an_price >= ALL(
    select an_price
    from zoo_ex_notnull
);
```

ID	AN_TYPE	AN_PRICE
4	bird	100

If we tried this without the All operator we would get an error that the subquery returns more than one row. But for our query we want all of the rows since we want to find a row with a value for an_price that is larger than or equal to every row in the view.

Demo 02: Try this with > ALL

```
select *
from zoo_ex_notnull
where an_price > ALL(
    select an_price
    from zoo_ex_notnull
);
```

no rows selected

We get the empty set since there is no row where an_price is larger than every row since that would mean that there is a value for an_price that is larger than itself.

Demo 03: Now filter the two parts of the query for dog and when we filter for the most expensive dog, we get back both dog rows that were tied for the first place. This is probably the easiest way to code find the biggest with ties.

```
select *
from zoo_ex_notnull
where an_type = 'dog'
and an_price >= ALL(
    select an_price
    from zoo_ex_notnull
    where an_type = 'dog'
);
```

ID	AN_TYPE	AN_PRICE
1	dog	80
18	dog	80

Demo 04: What if we try the same logic with the table which includes nulls; we get no rows returned because sql does not know if the null/missing prices are greater than 100 - the greatest actual value we have)

```
select *
from zoo_ex
```

```
where an_price >= ALL(
    select an_price
    from zoo_ex
);
no rows selected
```

Demo 05:

```
select *
from zoo_ex
where an_price >= ALL(
    select an_price
    from zoo_ex
    where an_price is not null
);
```

ID	AN_TYPE	AN_PRICE
4	bird	100

Demo 06: We could find the animal type where all of the animals of that type have the same price. For this we will exclude any nulls. This is a correlated subquery

```
select distinct an_type
from zoo_ex p1
where an_price is not null
and an_price = ALL (
    select an_price
    from zoo_ex p2
    where an_price is not null
    and p1.an_type = p2.an_type
) ;
```

AN_TYPE
cat
fish
lizard

If you want to exclude any an_type (such as cat) where there is only one row of that type, then add a group by and a Having clause count(*) > 1.

3. Finding the best(?) using the AltgeldMart tables

Sometimes we need to analyze data and find the item that is- in some sense- the best among the data. For example we could be asked to find the best selling product. The first thing to do is to get a better definition of "best selling". We will get to this in a moment.

I am going to add another order for a sporting goods item so that we will have a tie for this category in terms of orders. I will use order id 1 since that will be easier to delete later.

```
insert into oe_orderHeaders (order_id, order_date, order_mode, customer_id,
shipping_mode_id, order_status, sales_rep_id)
values ( 1, date '2014-06-20', 'DIRECT', 404950, 'FEDEX1', 1, 155);
insert into oe_orderDetails (order_id, line_item_id, prod_id, quoted_price,
quantity_ordered)
values ( 1, 1, 1020, 2200.00, 10);
```

To remove these later use

```
delete from oe_orderDetails where order_id IN(1);
delete from oe_orderHeaders where order_id IN(1);
```

Demo 07: Let's start with a count function; we are interested in sales of products so we should use the order details table and I will limit this to the SPG category to keep the results short.

```
select prod_id, catg_id, count(distinct order_id) as Cnt
from oe_orderDetails OD
join prd_products PR using(prod_id)
where catg_id = 'SPG'
group by prod_id, catg_id
order by Cnt;
```

PROD_ID	CATG_I	CNT
1030	SPG	4
1050	SPG	4
1040	SPG	8
1060	SPG	9
1020	SPG	12
1010	SPG	12

We want to count distinct order id in case some product was ordered twice on the same order. (that is a business decision.)

```
select order_id, line_item_id, prod_id
from oe_orderDetails
where order_id = 312
;
```

ORDER_ID	LINE_ITEM_ID	PROD_ID
312	1	1040
312	2	1050
312	3	1060
312	4	1060

Demo 08: Now we can find the row with the largest value for CntOrders for the SPG category. We will need to consider the possibilities of ties so we cannot just sort and take the last row

```
select prod_id, prod_name, prod_desc
from oe_orderDetails
join prd_products PR using(prod_id)
where catg_id = 'SPG'
group by prod_id, prod_name, prod_desc
having count(distinct order_id) >= All(
    select count(distinct order_id)
    from oe_orderDetails
    join prd_products PR using(prod_id)
    where catg_id = 'SPG'
    group by prod_id)
;
```

PROD_ID	PROD_NAME	PROD_DESC
1020	Dartboard	Cork-backed dartboard with hanger
1010	Weights	Set of 12 barbells 15 pounds

Demo 09: What if our definition of "best selling" should be based on the quantity of items sold?

```
select prod_id, prod_name, prod_desc
from oe_orderDetails
join prd_products PR using(prod_id)
where catg_id = 'SPG'
group by prod_id, prod_name, prod_desc
```

```

having sum(quantity_ordered) >= All(
    select sum(quantity_ordered)
    from oe_orderDetails
    join prd_products PR using(prod_id)
    where catg_id = 'SPG'
    group by prod_id);

```

PROD_ID	PROD_NAME	PROD_DESC
1010	Weights	Set of 12 barbells 15 pounds

Demo 10: What if our definition of "best selling" should be based on the sales amount (total of price * quantity)?

```

select prod_id, prod_name, prod_desc
from oe_orderDetails
join prd_products PR using(prod_id)
where catg_id = 'SPG'
group by prod_id, prod_name, prod_desc
having sum(quantity_ordered * quoted_price) >= All(
    select sum(quantity_ordered*quoted_price)
    from oe_orderDetails
    join prd_products PR using(prod_id)
    where catg_id = 'SPG'
    group by prod_id)
;

```

PROD_ID	PROD_NAME	PROD_DESC
1020	Dartboard	Cork-backed dartboard with hanger

4. Using the Any Operator

The Any operator is similar to All. The words Any and Some are interchangeable. I do not find this operator as useful as the ALL operator. In some cases you can use Any instead of an In list.

Demo 11: This is an ANY test on price. If we ask to see all of the rows with a price greater than any of the prices we get rows returned. This means we want prices greater than any of the other prices- essentially all prices greater than the smallest price in the table(in our case the value 10.); it does not return rows with nulls.

```

select *
from zoo_ex
where an_price > ANY (select an_price from zoo_ex)
order by an_price
;

```

ID	AN_TYPE	AN_PRICE
14	snake	25
12	lizard	50
9	snake	50
5	bird	50
7	lizard	50
15	bird	80
18	dog	80
17	bird	80
1	dog	80
4	bird	100

Demo 12: Which animals cost the same as a bird- any bird?

```
select *
from zoo_ex
where an_price = ANY (
    select an_price
    from zoo_ex
    where an_type = 'bird'
    and an_price is not null
)
order by an_price;
```

ID	AN_TYPE	AN_PRICE
5	bird	50
7	lizard	50
9	snake	50
12	lizard	50
1	dog	80
15	bird	80
17	bird	80
18	dog	80
4	bird	100

Table of Contents

1.	Business Rules and Table Design.....	1
2.	Relationships and Referential Integrity.....	4
2.1.	Type of relationships	4
2.2.	Referential Integrity rule	4
3.	Normalization.....	5
3.1.	Goals of normalization	5
3.2.	Anomalies.....	5
4.	Examples of data that is not normalized.....	6

Before we cover the syntax for creating tables and setting constraints, it is time to talk a bit more about relations and relationships. Relations is the formal term for how data is structured; we visualize relations as tables. Relationships are the way that tables relate to each other and are essential to understanding how a database works. So we will also look at a few more definitions and considerations for relationships.

As we talk about the design and structure of tables we are discussing metadata. Metadata is the data that refers to the structure of the database, such as the names of the tables and their attributes, the data types of the attributes, and the primary keys. It does not refer to the values (such as the last name of a customer is Jones); instead it refers to the attribute that we use to store that value (such as the last name can store up to 25 characters and cannot be null).

1. Business Rules and Table Design

The ultimate design of a database and its tables comes from the business rules of the company. Business rules are statements about the way a company structures its data and controls its operations. For example, in our vets database there is a rule that each animal is the responsibility of a single client. This business rule dictates that each animal row in the animal table is associated with one client row and, therefore, cl_id is an attribute of the animals table.

Many poorly designed databases have poorly designed tables. Most problems arise from having tables that are too complex—that have too many attributes that do not belong to that table. This is often a result of thinking of the data as a manual table or an Excel spreadsheet. Since manual tables for storing data do not have automatic look-up features, we tend to put all of the data into one big table so that we can find it. This same attitude is often found when people create spreadsheets. We might have a spreadsheet where each row for an animal includes the client id but also their name and address. We would not want to bring that design directly into a database table.

Suppose we had the following single table for animals and clients.

An_id	Type	Name	dob	C1_id	Last Name	First Name	Address	City	State	Zip
11015	snake	Kenny	2005-10-23	4534	Montgomery	Wes	POB 345	Dayton	OH	43784
11029	bird		2005-10-01	4534	Montgomery	Wes	POB 345	Dayton	OH	43874
12035	bird	Mr Peanut	1995-02-28	3560	Monk	Theo	345 Post Street	New York	NY	10006
12038	cat	Gutsy	2007-04-29	3560	Monk	Theo	3405 Post Street	New York	NY	10006
15001	turtle	Big Mike	2008-02-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
15341	turtle	Pete	2007-06-02	93	Wilson	Sam	123 Park Place	Big Rock	AR	71601
15002	turtle	George	2008-02-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601

16002	cat	Fritz	2009-05-25	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21002	snake	Edger	2002-10-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21004	snake	Gutsy	2001-05-12	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21005	cat	Koshka	2004-06-06	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21006	snake		1995-11-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
			5686	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112	
			5689	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112	
			5698	Biederbecke	Sue	50 Phelan Ave	San Francisco	CA	94112	

The spreadsheet approach leads to two problems:

- having rows of data with a lot of empty cells
- data redundancy, having the same data value stored more than once.

In the above table we have several rows which have no data values for the animal columns because these clients don't have any animals. But if we are storing our animal and client data in this table we must have rows for every client even if they do not have any animals.

We also are repeating a lot of values which leads to more problems- did you notice the two errors in the data in the table?

Data redundancy leads to problems. One of the first problems we tend to think of is that we are wasting storage space- we do not need to store client 25's name and address 7 times just because he has 7 animals. But storage space is pretty cheap. The more important problems are the logical ones.

- update anomalies where one copy of the data value is changed but other values of the same data item are not changed
- delete anomalies where deleting one data value results in too much data being deleted
- insert anomalies where the user cannot insert the data values they wish to store.

Suppose client 25 moves- how many places do we have to update that data? Are you certain that every application that updates data will update every one of those rows?

If animal 15341 dies and we want to remove its row, we might also remove the only row we have for client 93.

Suppose we need a rule that the animal id and type and dob could not be null. How do we enforce this rule and still allow the table to include clients with no animals?

These are problems that were a major consideration with traditional file processing and database systems were suppose to help solve these problems.

With a DBMS you can split the data into smaller, more tightly focused tables and then reassemble the big collections of data when needed through queries. The general problem in going from a big table to properly designed tables is to create a collection of related tables without losing any of the information contained in the original collection of data. Note that we have to preserve the table information-not just the table data. Table information includes facts imbedded in the table design.

At this point in the semester, these should seem natural- the client table includes the client name and address and each client gets one name and one address. And the animal table gets one row for each animal.

Cl_id	Last Name	First Name	Address	City	State	Zip
25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
93	Wilson	Sam	123 Park Place	Big Rock	AR	71601

3560	Monk	Theo	3405 Post Street	New York	NY	10006
4534	Montgomery	Wes	POB 345	Dayton	OH	43784
5686	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112
5689	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112
5698	Biederbecke	Sue	50 Phelan Ave	San Francisco	CA	94112

An_id	Type	Name	dob	C1_id
11015	snake	Kenny	2005-10-23	4534
11029	bird		2005-10-01	4534
12035	bird	Mr Peanut	1995-02-28	3560
12038	cat	Gutsy	2007-04-29	3560
15001	turtle	Big Mike	2008-02-02	25
15341	turtle	Pete	2007-06-02	93
15002	turtle	George	2008-02-02	25
16002	cat	Fritz	2009-05-25	25
21002	snake	Edger	2002-10-02	25
21004	snake	Gutsy	2001-05-12	25
21005	cat	Koshka	2004-06-06	25
21006	snake		1995-11-02	25

You might notice that often well designed tables have fewer columns.

If it were important to allow more than one address for a client, then we could split the client table into multiple tables. One table for the client name (each client gets one name) and a second related table for the client addresses. But those tables would also each need the client id to link back to the main client table. This requires understanding the business rules and policies reflected in the data collection.

Now some clients can have multiple addresses

C1_id	Last Name	First Name
25	Harris	Eddie
93	Wilson	Sam
3560	Monk	Theo
4534	Montgomery	Wes
5686	Biederbecke	NULL
5689	Biederbecke	NULL
5698	Biederbecke	Sue

C1_id	Address	City	State	Zip
25	2 Marshall Ave	Big Rock	AR	71601
25	2957 Grover Ave	Springfield	IL	61258
93	123 Park Place	Big Rock	AR	71601
93	56 Meadowland Ln	Springfield	NY	10027
3560	3405 Post Street	New York	NY	10006
4534	POB 345	Dayton	OH	43784
5686	50 Phelan Ave	San Francisco	CA	94112
5689	50 Phelan Ave	San Francisco	CA	94112
5698	50 Phelan Ave	San Francisco	CA	94112

2. Relationships and Referential Integrity

The following are rules and considerations that you would need to think about when you set up tables and constraints.

2.1. Type of relationships

There are three types of relationships:

- One to One
- One to Many
- Many to Many

The relationship types indicate how many rows in one table can be associated with a row in another table.

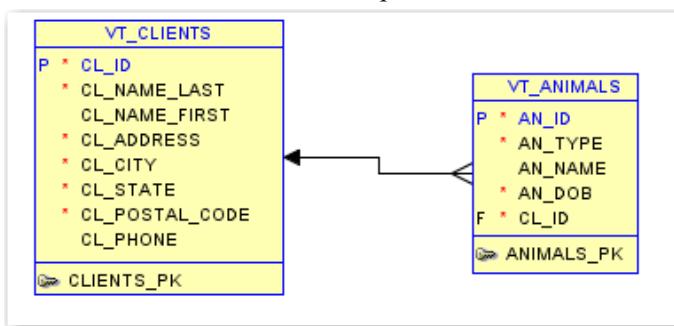
One-to-One- this means that a row in table A is related to at most one row in table B. Suppose we wanted to keep a photo of each member of our staff. Assuming we had a way to store a photo in our database, we might want to set up a second table vt_staffPhoto (staff_id, Photo) since a photo would take a lot of space and we would not access it very often. Keeping that data in another table makes our regular staff table more efficient to access. We might also decide that we want to keep some of the employee data confidential; we could store the confidential data in a second table and restrict access to that table to only a few users. We would include the staff id attribute in the second table to link the two tables in a 1:1 relationship.

One-to-Many- this means that a row in table A can be related to 0, 1, or multiple rows in table B and each row in table B is related to one row in table A. This is the more common relationship. A client row in the clients table can be related to 0, 1, or multiple rows in the animals table and each row in the animals table can be related to 0, 1, or multiple rows in the exams table.

Many-to-Many – this means that a row in table A can be related to 0, 1, or multiple rows in table B and each row in table B is related to 0, 1, or multiple rows in table A. This is a relationship that we can think of as a logical relationship but we seldom see it in a database as most dbms do not support this directly. If we think about medications and exams- a medication can be used on multiple exams and an exam can include multiple medications. One of the purposes of the exam details table is to serve as a bridge to implement this many-to-many relationship.

2.2. Referential Integrity rule

We will look at these in terms of the vets set of tables. This is a diagram showing the clients table and the animals table and the relationship between them.



We have referential integrity set between the Clients table and the Animal table. The pk for the clients table is cl_id and since it is a pk it is not nullable. The pk for the animals table is an_id. The animals table also contains an attribute cl_id which is the fk to the client table. We can navigate through the relationship from the animal table back to the client table to find the name of the client for an animal.

Referential Integrity Rule: The database must not contain any non-null unmatched foreign keys. If there is a value in a foreign key attribute, then that value must match a value in the related attribute in the parent table.

In terms of our tables this means that we cannot add a row to the animals table that has a value for the client id that does not match an existing client row. If we want to add a new animal for a new client, we have to add the client first.

The referential integrity rule does allow for null foreign keys. Your business rules might call for a non-null foreign key.

In our animal table the cl_id field is not nullable. We are not allowed to have an animal row without a related client row. But that is not required by the referential integrity rule. It would be possible to set up the animals table with a nullable cl_id attribute. That would let us add animals with no associated client. This would make it difficult to know who to charge for an exam done for the animal- and the vet probably would not like this. What the referential integrity rule says is that an animal row cannot have a value for cl_id that is not a valid cl_id in the clients table- you can't just make up a client id for an animal.

You will not be able to set referential integrity if there already is data present in the tables that violates the rule, or if the attributes involved do not have the same data types.

3. Normalization

Normalization is the formal process of decomposing tables that have design problems into well-structured relations. Good design is sometimes expressed as

- Storing one fact in one place
- Having all of the attributes in a table depend on the primary key, the whole key and nothing but the key.

3.1. Goals of normalization

Normalization helps you design table to

- preserve the integrity of the data
- identify a unique identifier for each record
- reduce redundancy. Redundancy is storing the same data value in more than one table. Redundancy leads to errors.
- minimize the space needed for the tables which also reduces the volume of data that must be transferred to the client.
- store the data you need without having to create dummy data

No design technique will guarantee that you have well designed tables. Having normalized the tables will help.

3.2. Anomalies

Errors in a table that are a result of a user attempts to modify the data are called anomalies.

Update Anomaly — having inconsistent data because the data was updated in one row but not in all rows. For example, in the spreadsheet style table for the animal and client data, the zip code for client 4534 has two different values.

Insertion Anomaly — the inability to add a new data to a table. For example, we cannot add a new client row to the table unless we have data for an animal for that client (or leave these as null values).

Deletion Anomaly — loss of data because we wanted to remove other data values. For example, if we want to remove the data for animal 15342, we could lose all information for client 93 (unless we leave several attributes with null values).

4. Examples of data that is not normalized

For this we will work with variations on the tables in vets database. The examples will focus on mistakes we might have made in the design and problems these might cause.

Design change 1) In the clients table we have a phone number column. We have now found that our clients have more than one phone number and want to give us both their home number and their office number. If the vet has to keep Fluffy overnight and make decisions about Fluffy's treatment, it might be important to be able to reach the client quickly. So we decide to add a new column to the client table and rename the existing column- we now have a column named home_phone and a column named office_phone. This might have worked in the past when people had one home phone and one job (M_F 9 to 5). This design might still work- but it would be harder to get a list of all of the phone numbers in a single column in the output of a query. And a query that tries to find out which client has a certain phone number has to query two columns. This table is now no longer considered to be good design.

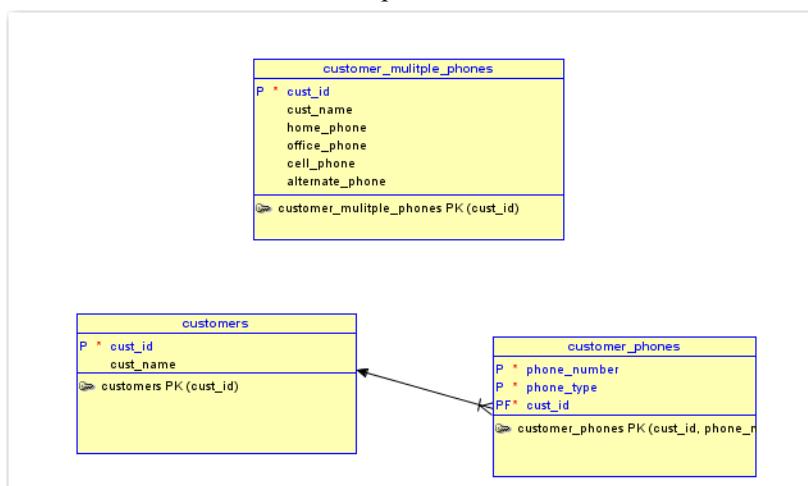
There are several bigger problems: (1) we cannot always distinguish between a home phone and an office phone- what do you do if someone works from home? (2) some people work multiple jobs and have more than one office phone; some people have more than one home phone. (3) how do we classify a cell phone?

Another approach that does not work well is to put all of the phone numbers into a single column separated by commas. Searching is difficult and it is difficult to enforce any format for a legitimate phone number

So if we are going to store multiple phone numbers for a customer, then we need another table- ClientsPhones. This table could have two columns: cl_id and cl_phone. The cl_id relates back to the customers table so we can identify a phone number as belonging to a specific client and we can go from a client to their phone numbers. We might decide to have more columns in that table- phone type- (home, office, cell, fax, weekendHome), maybe a column for the best time to call.

The relationship between clients and phone numbers now follows the same pattern as the relationship between clients and animals. A client could have 0, 1, or more phone numbers. This is a one-to-many relationship. We implement it by making the cl_id in ClientsPhones be a fk to the clients table. We also avoid having a column in the client table which is nullable (The vt.client.cl_phone attribute was a nullable column).

This shows two design possibilities- the first has multiple columns for phone numbers and the second design uses two tables and a relationship.



Design change 2) Suppose we had designed the exam headers and the exam details tables so that the exam date was in the exam details table. Perhaps we knew that we would need to write many queries about services that were done on specific dates. By putting the exam date in the exam details table we could avoid a join. But this is a poor decision. The PK of the exam details table is (ex_id, srv_id) but the exam date depends on the exam id only - it does not depend on the service id (Remember, every piece of data in the table should depend on the

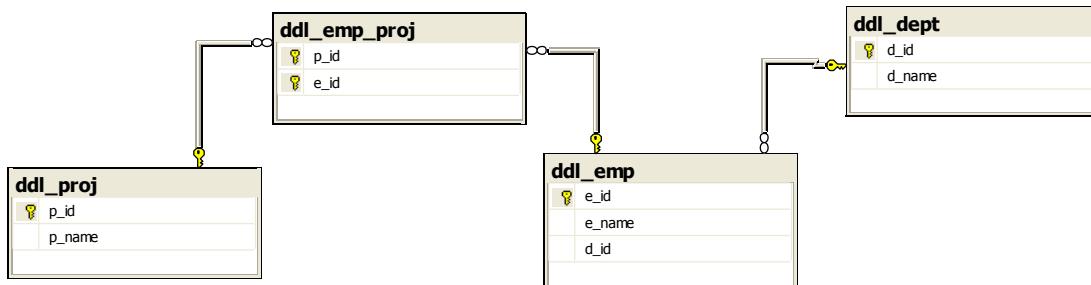
key, the whole key and nothing but the key) From a practical point of view, putting the exam date in the exam detail table would be that we would have to repeat that data value in multiple rows- if an exam involved 5 services we would have to repeat the exam date five times. This is redundant data. There also is a problem that the exam date value might be entered with different values for different rows. This could be due to a data entry error or a change in the system time value while the rows are being entered. If the date needed to be corrected, then the app would need to correct all of the rows consistently. This could be handled at the app level- but it does make this harder.

Design change 3) Suppose we decide that it makes sense to store the staff name in the exam table- so we would have (ex_id, an_id, stf_id, stf_name_last, ex_date). That would save us a join when we wanted to display the details of who was responsible for an exam. But this is a transitive dependency. The pk is ex_id which determines the stf_id which determines the stf_name_last. So in a sense the ex_id determines the stf_name_last but the dependency goes through the stf_id. This design would also cause us to store redundant data and open up the possibility that we spell the person's last name in different ways in different rows. We would also have a problem when a staff person changed their name- we would have to update it in multiple tables.

Table of Contents

1.	Removing a table.....	1
2.	Simple table creation	2
2.1.	Add a single new row using a value list.....	2
3.	Renaming a Table	3
4.	Creating Tables with Integrity	3
4.1.	Primary key, not null, and unique constraints	4
4.2.	Foreign key constraints	5
4.3.	Other constraint types	8
4.4.	Disable and enable constraints	9
4.5.	Drop cascade constraints	9
5.	Data types you can use in tables.....	10
5.1.	Character strings	10
5.2.	Integers	11
5.3.	Fixed precision.....	11
5.4.	Floating point/ Approximate numbers.....	11
5.5.	Temporal data	11
5.1.	XML.....	11
6.	Changing the Table Design.....	11

These queries provide a way to use SQL to create relations, change their design, set relationships, create and drop indexes. These statements are considered Data Definition Language statements. This section shows the queries need to create the relations and relationships shown below.



Some of the demo code will create tables that are used to illustrate a technique. You can delete those tables at the end of the demos for this week. These table names will all start `ddl_`

1. Removing a table

You can drop a table and all of its data with the Drop Table query. The DBMS does not ask you if you are sure that you want to drop the table.

Demo 01: Dropping a table

```

DROP TABLE ddl_emp_proj ;
Table dropped.

```

If you try to drop a table that does not exist you will get an error message. In Oracle this error does not stop a script that is running. So you can have drop table commands at the start of a script that creates tables. It is your responsibility to be certain that you do not drop the wrong tables in error.

```
DROP TABLE ddl_emp_proj ;
DROP TABLE ddl_proj ;
DROP TABLE ddl_emp ;
DROP TABLE ddl_dept ;
```

Sometimes you may find that you cannot drop a table if the table is involved in a relationship as the parent. You cannot drop the parent table if there are rows in the child table. The above set of drops starts with the child table and then works up to the parent tables.

2. Simple table creation

The minimal SQL statement to create a table requires that you give the table a name and that you give each column a name and a data type. The table has to have at least one column defined. This statement does not set a primary key, default value, foreign key references, or other rules about the data- commonly called constraints. All this SQL statement does is create the table structure.

You cannot have two tables with the same name, so if you are experimenting with these statements you might need to drop one or more of these tables. Oracle also enforces rules about dropping tables that are involved in relationships.

Demo 02: Minimal statement to create a table with two columns

```
CREATE TABLE ddl_dept (
    d_id      number(3)
, d_name    varchar2(15));
```

Table created.

When you declare a data type for a column, you limit the values that can be entered in that column. In the above table, the values for d_name are limited to 15 characters. Oracle will not truncate string values to fit the defined data type precisions. The values for the d_id values must be in the range -999 to +999 due to the number (3) designation. Oracle will round decimal values to fit the precisions defined for a numeric type- so this will accept the value 123.5 and round it to 124 but it will not accept the value 1234.

2.1. Add a single new row using a value list

In order to understand table creation you need to test with insert statements. This is a model to insert a single row into the table specifying the column names. This is the preferred technique.

```
insert into my_table (col1_name, col2_name, . . .)
values (value1_name, value2_name, . . .);
```

We will look at variations of the insert statement .

Demo 03: These are examples of Insert statements that may work- or not

Insert statement that works

```
insert into ddl_dept (d_id, d_name)
values (10,'Sales');
```

Insert statement that works; the default is that the columns can accept nulls

```
insert into ddl_dept (d_id, d_name)
Values(null,null);
```

Insert statement that works; we did not set a primary key so we can have two rows with the same value for d_id.

```
insert into ddl_dept (d_id, d_name)
values (10,'Marketing');
```

Insert statement that works; the value for d_id is rounded to 41. This is an implied cast.

```
insert into ddl_dept (d_id, d_name)
values (40.8,'Development');
```

Insert statement that works; We did not restrict negative numbers

```
insert into ddl_dept (d_id, d_name)
values (-44, ''');
```

Insert statement that fails; value for d_id is larger than specified precision allowed for this column

```
insert into ddl_dept (d_id, d_name)
values (1234, 'Research');
```

Insert statement that fails; value for d_name is too large for column (actual: 22, maximum: 15)

```
insert into ddl_dept (d_id, d_name)
values (20, 'Accounting and Payroll');
```

Insert statement that fails; not enough values; we specified two columns in the list but provided only one value.

Note that this does not use a null for the missing value.

```
insert into ddl_dept (d_id, d_name)
values (35);
```

Insert statement that works; this specifies one column in the list and we provide one value. In this case, d_name gets a null entry.

```
insert into ddl_dept (d_id)
values (35);
```

```
select * from ddl_dept order by d_id;
```

D_ID	D_NAME
-44	
10	Sales
10	Marketing
35	
41	Development

6 rows selected

Drop the table ddl_dept. We need a better version of this table.

```
DROP TABLE ddl_dept ;
```

3. Renaming a Table

You can rename a table and Oracle will update constraints and fk relationships relating to the table- but will not update the table name in views, script file etc. This is a fairly dangerous change to make to a working system.

Demo 04: Renaming a table

```
RENAME zoo TO sfzoo;
```

Table renamed.

4. Creating Tables with Integrity

You want your database to have integrity- you want to ensure that as far as possible the values in your data are correct. There are three types of integrity:

- Entity integrity- each row should be uniquely identifiable- enforced by primary key constraints
- Referential integrity- a row in a child table needs to be related to a row in a parent table- enforced by foreign key constraints.
- Domain integrity- a data value meets certain criteria- in its simplest form, this is enforced by the data type; this can be more closely defined by a check constraint.

To create a usable database you would want to set these constraints— rules about what data values can be entered in the tables and how the tables are related to each other. Each constraint is an object and has a name.

Oracle will create constraint names for you if you do not create your own names. You will see many table creation statements that do not create constraint names and that let Oracle create the names. This approach will make it harder to manipulate the constraint later. Constraint names should reflect the table name, the attribute name, and the type of constraint. Constraint names follow the normal Oracle rules for identifiers and are also limited to 30 characters. The pattern I follow of ending constraint names with tags such as _pk, fk etc is common but is not a syntax requirement.

Constraint names must be unique within the schema.

A constraint can be declared on the same line as the column declaration (a column constraint); the constraint may be declared in the create statement after all of the columns are declared (a table constraint). You can also add constraints to an existing table.

At a minimum, when you create a table, you would set the primary key and designate which attributes cannot be left empty.

Constraints that you create when you create the table, or that you add to the table, are enforced by the DBMS. These are called declarative constraints. Other rules about data can be enforced with triggers- these are called procedural integrity constraints. We are not discussing triggers in this class. (see CS151P for triggers.)

4.1. Primary key, not null, and unique constraints

Demo 05: Creates the project table; sets the pk and a unique constraint. Chose a constraint name that reflects its purpose.

```
CREATE TABLE ddl_proj (
    p_id      varchar2(10)      constraint ddl_proj_PK primary key
    , p_name   varchar2(15)      null
                                constraint ddl_proj_P_Name_UN unique
);

```

Demo 06: Creates the department table; sets the pk and a unique constraint If you created this table earlier be sure to drop the previous version.

```
CREATE TABLE ddl_dept(
    d_id      number(3)        constraint ddl_dept_PK primary key
    , d_name   varchar2(15)      not null
                                constraint ddl_dept_D_Name_UN unique
);

```

Primary Key

Setting a primary key constraint also means that the attribute cannot be null and must have unique values with the table.

You can declare a primary key constraint on sets of columns using a table constraint. In that case use the syntax shown here where col1 and col2 are already defined as columns in the table.

```
constraint constraint_name_PK primary key (col1, col2)
```

Null, Not Null

The default setting for nullability in most dbms is that the column can accept null values. Since defaults can be changed, it is a good idea to always specify Null or Not Null for attributes. The NOT NULL constraint is generally not given a constraint name. This constraint can be set only as a column constraint.

Unique

The unique constraint means that no two rows in the table can have the same value for that attribute . Setting a column to be Not Null and Unique means that it is an identifier and is a candidate key for this table. You can have only one primary key defined per table; you can define as many candidate keys as you need for a table.

When you define an attribute to be null and unique, Oracle will let you have multiple rows in the table with a null value for that column. This agrees with the attitude that a null does not equal another null.

You can declare a unique constraint on sets of columns using a table constraint. In that case use the syntax shown here where col1 and col2 are already defined as columns in the table.

```
constraint constraint_name_UN    unique(col1, col2)
```

Demo 07: The Unique constraint on a two column set

```
create table ddl_un (id number(3)
, city varchar2(15), state char(2)
, constraint location_UN unique(city, state)
);
```

The first three inserts are ok

```
Insert into ddl_un (id, city, state) values (1, 'Chicago', 'IL');
Insert into ddl_un (id, city, state) values (2, 'Chicago', 'CA');
Insert into ddl_un (id, city, state) values (3, 'Pekin', 'IL');
```

This one fails because it is a duplicate of (city, state)

```
Insert into ddl_un (id, city, state) values (4, 'Pekin', 'IL');
ORA-00001: unique constraint (ROSE151A.LOCATION_UN) violated
```

What about nulls? These are both allowed; one null is not equal to another null.

```
Insert into ddl_un (id, city, state) values (5, null, null);
Insert into ddl_un (id, city, state) values (6, null, null);
```

ID	CITY	STATE
1	Chicago	IL
2	Chicago	CA
3	Pekin	IL
5		
6		

But the second of this pair of inserts is blocked.

```
Insert into ddl_un (id, city, state) values (7, null, 'NY');
Insert into ddl_un (id, city, state) values (8, null, 'NY');

drop table ddl_un;
```

To satisfy a unique constraint, no two rows in the table can have the same value for the unique key. However, the unique key made up of a single column can contain nulls. To satisfy a composite unique key, no two rows in the table or view can have the same combination of values in the key columns. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.

4.2. Foreign key constraints

The department table and the project table are parent tables and can be created independently. The Employee table is a child of the Department table, therefore you have to make the Department table before the Employee table. The referenced attribute in the parent table must have been defined as the PK or with a Unique constraint. This is so that a child row refers to uniquely one parent row.

Demo 08: Creates the employee table and creates the relationship object to the department table.

```
CREATE TABLE ddl_emp(
    e_id      number(3)      constraint ddl_emp_pk primary key
, e_name   varchar2(15) not null
, d_id     number(3)      not null constraint ddl_emp_dept_fk references ddl_dept
);
```

Demo 09: Creates the employee project table with a composite primary key. The composite pk has to be defined on a separate line as a table constraint. This also creates the relationships to the employee and project tables.

```
CREATE TABLE ddl_emp_proj (
    p_id  varchar2(10) not null
          constraint ddl_emp_proj_proj_fk references ddl_proj( p_id)
, e_id  number(3)  not null
          constraint ddl_emp_proj_emp_fk references ddl_emp(e_id)
, constraint ddl_empproj_pk primary key ( p_id, e_id)
);
```

After running these queries, you will have four tables.

Foreign Key

The foreign key constraint specifies the foreign key attribute(s) in the child table and the parent table and column(s) for the relationship. The referenced column(s) in the parent table must have a Unique or Primary Key constraint.

The foreign key constraint is set in the child table. The child claims the parent.

To set this as a column constraint, you use the following

```
myCol DataType CONSTRAINT cnstrName REFERENCES parentTbl(col_name)
```

If the referenced key is the primary key of the parent table, you do not have to mention it in the constraint.

It is also possible to skip the keyword constraint and the constraint name- but then you get a system generated name.

```
myCol DataType REFERENCES parentTbl
```

To set this as a table constraint, you use the following. Since this is a table constraint you have to include the name of the attribute that is the foreign key.

```
CONSTRAINT cnstrName FOREIGN KEY (myCol) REFERENCES parentTbl(col_name)
```

If the foreign key is made up of multiple columns, create a single constraint listing both columns

```
CONSTRAINT cnstrName FOREIGN KEY (myCol1, myCol2)
    REFERENCES parentTbl(col_name1, col_name2 )
```

You can include a rule as to how to handle deletes with On Delete clause

ON DELETE CASCADE, ON DELETE SET NULL: with the foreign key constraint, you can specify what should happen to the child rows if a related parent row is deleted. The default behavior is that you cannot delete a parent row if it has any child rows.

```
constraint cnstrName references parentTbl on delete cascade
constraint cnstrName references parentTbl on delete set null
```

Demo 10: Creates a parent and a child table with cascade delete. These will be very simple table and I am not naming the constraints.

```
CREATE TABLE ddl_parent (p_id number(3) primary key);
```

```

CREATE TABLE ddl_child (c_id number(3) primary key
                      , fk_id number(3) references ddl_parent
                      on delete cascade);

Insert into ddl_parent(p_id) values (1);
Insert into ddl_parent(p_id) values (2);
Insert into ddl_parent(p_id) values (3);

Insert into ddl_child(c_id, fk_id) values (100,2);
Insert into ddl_child(c_id, fk_id) values (101,2);
Insert into ddl_child(c_id, fk_id) values (103,2);
Insert into ddl_child(c_id, fk_id) values (104,3);

select p_id, c_id, fk_id
from ddl_parent
join ddl_child on ddl_parent.p_id = ddl_child.fk_id;

```

P_ID	C_ID	FK_ID
2	100	2
2	101	2
2	103	2
3	104	3

Demo 11: Deleting rows. The parent row with id 2 is related to three child rows.

```

delete from ddl_parent where p_id = 2;
1 row deleted.

```

If I select the rows in the parent table- one row was removed

```
Select * from ddl_parent;
```

P_ID
1
3

If I select the rows in the child table- three rows were removed; even though the feedback from Oracle had indicated only one deleted row when I did the original delete.

```
Select * from ddl_child;
```

C_ID	FK_ID
104	3

Demo 12: You can demonstrate this with the set null constraint. Clean up the tables and reinsert the rows.

```

drop table ddl_child ;
create table ddl_child (c_id number(3) primary key
                      , fk_id number(3) references ddl_parent
                      on delete set null);

insert into ddl_parent(p_id) values (2);
insert into ddl_child(c_id, fk_id) values (100,2);
insert into ddl_child(c_id, fk_id) values (101,2);
insert into ddl_child(c_id, fk_id) values (103,2);
insert into ddl_child(c_id, fk_id) values (104,3);

```

```
delete from ddl_parent where p_id = 2;
1 row deleted.
```

Now when you look at the rows in the child table the rows are still there but the values for the foreign key that were originally set to 2 are now nulled out. This also requires that the fk_id in the child table was a nullable field.

```
select * from ddl_child;
C_ID      FK_ID
-----  -----
100
101
103
104      3
```

```
drop table ddl_child ;
drop table ddl_parent ;
```

4.3. Other constraint types

DEFAULT: This declaration is used to specify a default value that will be placed in a column.

MyCol DataType Default default_value

Demo 13:

```
create table ddl_default (
    id number(3)
    , d_state char(2) default 'CA'
);
```

You can use the word default in the insert or skip the column to get the default in the column list.

```
insert into ddl_default (id, d_state) values (1, 'PA');
insert into ddl_default (id, d_state) values (2, 'CA');
insert into ddl_default (id, d_state) values (3, default);
insert into ddl_default (id) values (4);
ID D_
-----
1 PA
2 CA
3 CA
4 CA
```

```
drop table ddl_default;
```

CHECK: use this to set other types of tests that a value must meet to be allowed into the table. The CHECK constraint condition must evaluate to TRUE or NULL to be satisfied. It cannot use SYSDATE as part of its test. If you create this as a table constraint, then it can refer to other values in the same column-such as fld1 <= fld2. If you declare this as a column constraint, then it can refer only to this column.

The syntax for a check constraint is

CONSTRAINT MyConstraintName CHECK (test)

The test can use equality tests, In, Between. Use the full test expression even if you are declaring this as a column constraint. (List_Price > 15) The test must be enclosed in parentheses.

Demo 14:

```
create table ddl_check (
    id number(3)
    , d_state char(2) constraint dstated_ck
```

```
        check(d_state in ('CA', 'NV', 'IL'))
);
```

The first two inserts work.

```
insert into ddl_check (id, d_state) values (1, 'CA');
insert into ddl_check (id, d_state) values (2, 'IL');
```

The following insert fails because NY is not in the approved list of values.

```
Insert into ddl_check (id, d_state) values (3, 'NY');
ORA-02290: check constraint (ROSE151A.DSTATED_CK) violated
```

The following inserts succeeds. The rule for tests in inserts is different than the rule in Where clauses. If the value to be inserted does not specifically fail the test then it is allowed.

```
insert into ddl_check (id, d_state) values (4, null);
```

If you really want the value in d_state to be one of those three values you can also set a not null constraint on the column.

```
Drop table ddl_check;
```

You can create the constraints in the same SQL statement as you use to create the table. You can also use a Create table statement to set the column names and data types and then use Alter Table statements to add the constraints. You can add constraints after data has been added to the table if the existing data meets the constraint rule.

Constraint names need to be unique with your schema.

4.4. Disable and enable constraints

You can disable a constraint and then enable it later. This is sometimes done during maintenance operations.

Demo 15:

```
alter table ddl_dept_2
disable constraint DSTATE2_CK;
```

Now I can insert the record from a previous demo that would have been rejected due to the check constraint violation

```
insert into ddl_dept_2 values(77, 'Payroll', 'Bowling Green', 'OH', NULL);
```

But if I try to enable that constraint, I cannot since there is now data in the table that violates the constraint. The dbms takes the integrity rules seriously! (this behavior varies with the dbms)

```
alter table ddl_dept_2
enable constraint DSTATE2_CK;
```

```
ERROR at line 2:
ORA-02293: cannot validate (ROSE151A.DSTATE2_CK) - check constraint violated
```

You might disable a constraint if you are adding a lot of records that are guaranteed to be clean. It takes time for the dbms to validate the data and it can be more efficient to do all of the validations after the inserts are complete.

4.5. Drop cascade constraints

Having relationship created between tables limits your ability to drop tables. You cannot drop a parent table if there is a related child table.

Demo 16:

```
drop table ddl_dept;
```

```
ERROR at line 1:
ORA-02449: unique/primary keys in table referenced by foreign keys
```

Oracle has a cascade constraint options that allows you to drop the parent table.

```
drop table ddl_dept cascade constraints;
```

```
Table dropped.
```

Demo 17: To see how this works, set up a simple pair of parent and child tables.

```
create table z_parent(pr_pk integer constraint parent_pk primary key
, col1 integer );
create table z_child (ch_pk integer constraint child_pk primary key
, col2 integer constraint child_fk references z_parent);
```

Display the constraints for these two tables. (this technique is discussed in the data dictionary document)

```
select table_name, constraint_name, constraint_type, status
from user_constraints
where table_name in( 'Z_PARENT','Z_CHILD');
```

TABLE_NAME	CONSTRAINT_NAME	C STATUS
Z_CHILD	CHILD_PK	P ENABLED
Z_CHILD	CHILD_FK	R ENABLED
Z_PARENT	PARENT_PK	P ENABLED

Now drop the parent with cascade

```
drop table z_parent cascade constraints;
```

Display the constraints again

```
select table_name, constraint_name, constraint_type, status
from user_constraints
where table_name in( 'Z_PARENT','Z_CHILD');
```

TABLE_NAME	CONSTRAINT_NAME	C STATUS
Z_CHILD	CHILD_PK	P ENABLED

We lost the parent_pk table- since we dropped that table. But we also lost the fk constraint on the child table. The child table remains and keeps its primary key- but it loses its foreign key.

5. Data types you can use in tables

For our tables we have been using only a few types of data to define our columns. Oracle supports more data types and you can find information about all of these in the appendix of the Price book. The following is a description of some of the most commonly used types. I am not going to give you a lot of ranges for the types- you can look those up if you need them.

5.1. Character strings

CHAR is used for fixed length strings. We use CHAR(2) to store state abbreviations since they all have a 2 letter. If you are storing data where all of the data has the same number of characters- such as SSN, ISBN13, some product codes, then CHAR is appropriate. If necessary the data will be end-padded with blanks to the stated length.

VARCHAR2 is used for variable length strings. We could use VARCHAR2(25) to store customer last names assuming we won't want to store a name longer than 25 characters.

For both of these - if you are defining a table column with char or varchar and try to insert a value longer than the defined length, you will get an error. Char defaults to a length of 1 if you do not state a length' varchar2 requires a length.

5.2. Integers

We usually use INTEGER or INT or we can use NUMBER(9) specifying a width.

5.3. Fixed precision

Number this lets you set up a type such as number(6,2) which can hold numbers from -9999.99 to +9999.99. The first value is the number of digits and the second the number of digits after the decimal point

For integer you can use a definition such as Number(5) specifying a width or Integer. Number(5) limits the values entered to 5 digits (-99999 to +99999)

5.4. Floating point/ Approximate numbers

Float and real- Use these types for numbers where the number of digits after the decimal is not fixed.

Suppose you wanted a table of different types of animals and their approximate weight. An elephant weights about 6800 kg and an ant about 0.000003 kg. It might make more sense to use a float than a decimal.

```
create table weights (an_type varchar2(15), an_weight_kg float);
insert into weights values ( 'ant', 0.000003);
insert into weights values ( 'elephant', 6800);
select * from weights;
```

AN_TYPE	AN_WEIGHT_KG
ant	0.000003
elephant	6800

5.5. Temporal data

These are for storing date and time values and it helps to see the range of possible values for these

DATE Jan 1 4712 B.C. and December 31 9999 A.D.

The Date type stores both a date and a time component.

5.1. XML

We will get to XML data later

6. Changing the Table Design

You can add and drop columns and constraints, modify columns data types and constraints, and Set Unused columns. If the table contains data, some of these changes are restricted. For example, you cannot add a check constraint to a table if the existing data would violate that constraint.

Demo 18: An alter query to add another column to an existing table

```
Create table ddl_alter (id integer primary key);

desc ddl_alter;
```

Name	Null	Type
ID		NOT NULL NUMBER
alter table ddl_alter		
add d_office varchar2(10) constraint office_un unique;		
desc ddl_alter;		
Name	Null	Type
ID		NOT NULL NUMBER
D_OFFICE		VARCHAR2(10)

Demo 19: Adding more than one column. Delimit the column collection with parentheses.

```
alter table ddl_alter
add ( e_ssn char(11)
, e_namefirst varchar2(20)
, e_salary number(6)
constraint e_salary_ck check (e_salary between 20000 and 100000)
);
```

Name	Null	Type
ID		NOT NULL NUMBER
D_OFFICE		VARCHAR2(10)
E_SSN		CHAR(11)
E_NAMEFIRST		VARCHAR2(20)
E_SALARY		NUMBER(6)

Demo 20: You can drop a column.

```
alter table ddl_alter
drop column e_ssn;
```

Name	Null	Type
ID		NOT NULL NUMBER
D_OFFICE		VARCHAR2(10)
E_NAMEFIRST		VARCHAR2(20)
E_SALARY		NUMBER(6)

Named constraints can be dropped by using an Alter table statement. You can use the system defined constraint names. You can also drop some constraints by naming the constraint type- for example:

Demo 21: Dropping a primary key. This is not ambiguous since a table has only one primary key.

```
alter table ddl_alter
drop primary key;
```

Note that the attribute id was not dropped - the PK constraint was dropped - which then removed the not null constraint.

Name	Null	Type
ID		NUMBER
D_OFFICE		VARCHAR2(10)
E_NAMEFIRST		VARCHAR2(20)
E_SALARY		NUMBER(6)

In some cases you may need to use Cascade; for example, when you are dropping a pk from a table which is referenced by a fk in a child table: ALTER TABLE tblDemo DROP PRIMARY KEY CASCADE;

Demo 22: You can drop a constraint without dropping the column; this does not drop any data.

```
alter table ddl_alter
drop constraint office_un;
```

Demo 23: Increasing the width of a column

```
alter table ddl_alter
modify e_namefirst varchar2(25);
```

Demo 24: Changing a constraint by dropping it and recreating it. This should be done at a time when other people are not using the tables so that no rows are inserted between these two statements..

```
alter table ddl_alter
drop constraint e_salary_ck;

alter table ddl_alter
add constraint e_salary_ck check (e_salary between 2000 and 10000);
```

Demo 25: Show the table description

```
desc ddl_alter;
```

Name	Null?	Type
ID		NUMBER (38)
D_OFFICE		VARCHAR2 (10)
E_NAMEFIRST		VARCHAR2 (25)
E_SALARY		NUMBER (6)

Demo 26: Add a few rows of data to the table ddl_alter

```
Insert into ddl_alter values (1, 'sales', 'Joe', 5000);
Insert into ddl_alter values (2, 'sales', 'Jill', 9000);
```

```
select *
from ddl_alter;
```

ID	D_OFFICE	E_NAMEFIRST	E_SALARY
1	sales	Joe	5000
2	sales	Jill	9000

Demo 27: Set unused columns. If you mark a column as unused, it is no longer accessible

```
alter table ddl_alter
set unused column e_salary;
Table altered.
```

Demo 28: Select * does not see the unused column.

```
select *
from ddl_alter;
```

ID	D_OFFICE	E_NAMEFIRST
1	sales	Joe
2	sales	Jill

Demo 29: describe does not see the unused column.

```
desc ddl_alter;
```

Name	Null?	Type
------	-------	------

ID	NUMBER (38)
D_OFFICE	VARCHAR2 (10)
E_NAMEFIRST	VARCHAR2 (25)

Demo 30: You can later do the actual drop of the columns

```
alter table ddl_alter  
drop unused columns;
```

One reason to do the actual column drop later is that this may involve a lot of work being done by the dbms to recreate the physical files; there may be a time when the system is not as impacted and the drop column can be run at that time.

Once you have marked a column as unused you cannot change your mind and undo that setting

Table of Contents

1.	Data Manipulation Language queries	1
2.	Truncating a table	2
3.	Delete queries	3
4.	Update queries.....	4
5.	Insert into queries	9
5.1.	Add a single new row using a value list.....	9
6.	Append rows to an existing table.....	11
7.	Create table As queries	12

1. Data Manipulation Language queries

After we have created tables, we need to be able to add rows of data to the tables; we also need to be able to change the rows in a table. For this, think of a table as a collection of rows- these changes affect one or more rows in the table. With an insert or a delete statement we appear to be changing on a row-basis but that does affect the entire table. With an update statement, we are actually replacing the current version of the rows with a new version of the rows so that is also changing the table.

DML (Data Manipulation Language) queries are used to change the data in the tables.

- Remove all rows from a table: Truncate
- Remove specified rows from a table: Delete
- Change existing rows in a table: Update
- Append single rows to a table: Insert Into
- Append multiple rows into a table, getting the rows from another table: Insert Into... Select
- Create new tables that have the same design as an existing table: Create Table As
- Do updates, insert, deletes in a single statement: Merge

We will do most of these queries using the following two tables AC_Dept and AC_Emp created below.

Since these statements change the data in tables, the changed data must meet the constraints that were created for the tables. If it does not, then the SQL statement will fail. If you attempt to execute an SQL statement that changes several rows of data and any one of the new rows would be invalid, then the entire statement fails and none of the rows are changed.

Demo 01: Drop the tables if they exist. Note that if you have created a series of related tables, you may have to drop child tables before parent tables.

```
drop table ac_emp;
drop table ac_dept;
```

Demo 02: Create table statements. Read these carefully to see the various constraints on the tables.

```
create table ac_dept (
    d_id          number(3)      constraint ac_dept_pk primary key
  , d_name        varchar2(15)   not null
  , d_budget      number(6)      null
  , d_expenses_to_date number(6)  null
  , d_city        varchar2(15)   not null
  , d_state       char(2)       not null
  , constraint ac_dlocation_un unique(d_name, d_city, d_state)
);

create table ac_emp (
    e_id          number(3)
```

```

, e_name    varchar2(15) not null
, d_id      number(3) null
, salary    number(5)
            default 30000
            constraint ac_esalary_ck1 check(salary between 20000 and 90000)
            null
, hiredate  date
            default trunc(sysdate)
            constraint ac_ehiredd_ck check(hiredate > to_date('2000-01-09',
'yyyy-mm-dd'))
            constraint ac_ehiredd_Midnight check(trunc(hiredate) = hiredate)
            null
, e_status  char(4)
            constraint ac_estatus_ck
            check(e_status in ('PERM', 'TEMP'))
            null
, constraint ac_emp_pk primary key (e_id)
, constraint ac_emp_dept_fk foreign key(d_id) references ac_dept
);

```

Demo 03: Initial inserts- These just set up some rows as I have done in the scripts

```
Insert Into ac_dept (d_id, d_name, d_budget, d_expenses_to_date, d_city, d_state)
Values (301, 'FINANCE', 50000, 15000, 'PEKIN', 'IL');
```

```
Insert Into ac_dept (d_id, d_name, d_budget, d_expenses_to_date, d_city, d_state)
Values (501, 'SALES', 15000, 16000, 'RENO', 'NV');
```

```
Insert Into ac_dept (d_id, d_name, d_budget, d_expenses_to_date, d_city, d_state)
Values (201, 'SALES', 35000, Null, 'CHICAGO', 'IL');
```

```
Insert Into ac_dept (d_id, d_name, d_budget, d_expenses_to_date, d_city, d_state)
Values (401, 'ADMIN', null, null, 'CHICAGO', 'IL');
```

D_ID	D_NAME	D_BUDGET	D_EXPENSES_TO_DATE	D_CITY	D_STATE
201	SALES	35000		CHICAGO	IL
301	FINANCE	50000	15000	PEKIN	IL
401	ADMIN			CHICAGO	IL
501	SALES	15000	16000	RENO	NV

```
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
Values (30, 'HANSON', 201, 40000, '15-MAY-2013', 'PERM');
```

```
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
Values (40, 'IBSEN', 201, 45000, '20-MAY-2013', 'PERM');
```

```
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
Values (50, 'MILES', 401, 25000, '20-JUNE-2013', 'PERM');
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
30	HANSON	201	40000	15-MAY-13	PERM
40	IBSEN	201	45000	20-MAY-13	PERM
50	MILES	401	25000	20-JUN-13	PERM

2. Truncating a table

```
truncate table mytable;
```

This removes all of the rows but does not remove the table structure. This is faster than using Delete but it cannot be rolled back. It does not activate triggers (code that can run when you change data in a table). If there are any fk references to this table, those constraints have to be disabled before truncating the table.

Demo 04: Truncate

You can do the following and the table ac_emp is emptied.

```
Truncate table ac_emp;
Table truncated.
```

```
select * from ac_emp;
no rows selected
```

But if you try this with the table ac_dept, you get an error and the table is not truncated. Note that the table ac_emp is empty so this is not a Cascade issue; this is based on the existence of a FK constraint.

```
Truncate Table ac_dept;
Truncate Table ac_dept
*
ERROR at line 1:
ORA-02266: unique/primary keys in table referenced by enabled foreign keys
```

```
select * from ac_dept;
```

D_ID	D_NAME	D_BUDGET	D_EXPENSES_TO_DATE	D_CITY	D_STATE
301	FINANCE	50000	15000	PEKIN	IL
501	SALES	15000	16000	RENO	NV
201	SALES	35000		CHICAGO	IL
401	ADMIN			CHICAGO	IL

If you deleted the rows in ac_emp, use the insert statements to put those rows back into the table.

3. Delete queries**Remove Existing Rows**

If you do these deletes, put the rows back into the table as each step. You can do this with the inserts in the earlier demo.

```
/* Model to delete row.
```

```
delete
from    tablename
where   condition;
```

Demo 05: This deletes all rows that match the test. (then reinsert those rows)

```
delete
from ac_emp
where d_id = 201;
2 rows deleted
```

Demo 06: This uses a subquery to allow a test for the proper rows to be deleted.

```
delete
from ac_emp
where d_id in (
    select d_id
    from ac_dept
    where d_state = 'NV');
0 rows deleted
```

That department was empty.

Change the query to delete people from the Chicago departments. Then reinsert those rows.

```
delete
from ac_emp
where d_id in (
    select d_id
    from ac_dept
    where d_city = 'CHICAGO');
3 rows deleted
```

If you do not include a Where clause you will delete all of the rows in the table.

A delete statement deletes rows from one table only.

If you are deleting parent rows and do not have cascade delete set for the relationship, then parent rows with existing child rows cannot be deleted. ORA-02292: integrity constraint (*ConstraintName*) violated - child row found.

Suppose we want to delete the two rows with the highest values for salary. We can use rownum but rownum and sort need to be handled separately. The following query will do a descending sort by salary.

```
select e_id
from ac_emp
order by salary desc;
```

If I use that subquery as a table expression, then I can use rownum in the outer query

```
select e_id
from
( select e_id
  from ac_emp
  order by salary desc
)
where rownum <=2;
```

Now I could use that query as the subquery for a delete

```
delete from ac_emp
where e_id in (
    select e_id
    from
    ( select e_id
      from ac_emp
      order by salary desc
    )
    where rownum <=2
);
```

This will delete two rows. If there were a tie on the salary value at the second row then one of those tied rows would be deleted but we are not controlling which one. That does not sound like a good business decision.

4. Update queries

Change Existing Rows

/* Model for an update.

```
update tablename
set columnname = value or expression
where condition;
```

Example: We have a column in the employee table that stores an Employee Status attribute. All employees will be changed to "TEMP" status and then some employees will then be changed to "PERM" status.

Demo 07: A few more employees

```
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (10, 'FREUD', 301, 30000, '06-Jun-2002', 'PERM');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (20, 'MATSON', 201, 60000, '06-Jun-2010', 'PERM');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (30, 'HANSON', 201, 40000, '15-MAY-13', 'PERM');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (40, 'IBSEN', 201, 45000, '20-MAY-13', 'PERM');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (50, 'MILES', 401, 25000, '20-JUN-13', 'PERM');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (60, 'TANG', 401, 25000, '15-JUL-12', 'TEMP');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (70, 'KREMER', 501, 50000, '15-JUL-12', 'TEMP');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (80, 'PAERT', 201, 65000, '15-JUL-12', 'TEMP');
Insert Into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
  Values (90, 'JARRET', 301, 60000, '08-AUGUST-2015', 'TEMP');
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
10	FREUD	301	30000	06-JUN-02	PERM
20	MATSON	201	60000	06-JUN-10	PERM
30	HANSON	201	40000	15-MAY-13	PERM
40	IBSEN	201	45000	20-MAY-13	PERM
50	MILES	401	25000	20-JUN-13	PERM
60	TANG	401	25000	15-JUL-12	TEMP
70	KREMER	501	50000	15-JUL-12	TEMP
80	PAERT	201	65000	15-JUL-12	TEMP
90	JARRET	301	60000	08-AUG-15	TEMP

Demo 08: SET the value for E_Status to TEMP for all rows. Everyone now has TEMP status.

```
update ac_emp
set e_status = 'TEMP';
```

Demo 09: The Where clause allows you to update specified rows. This changes some of the data to PERM.

```
update ac_emp
set e_status = 'PERM'
where hiredate < to_date('01-JUN-2013');
```

Demo 10: You can update multiple attributes. This updates one row since the Where clause selects for a PK.
There is only one SET clause even if we update multiple columns

```
update ac_dept
set d_city = 'SAN FRANCISCO',
    d_state = 'CA'
where d_id = 401;
```

Demo 11: The SET clause can use an expression.

```
update ac_emp
set salary = salary * 1.1
where e_status = 'PERM';
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
10	FREUD	301	33000	06-JUN-02	PERM
20	MATSON	201	66000	06-JUN-10	PERM
30	HANSON	201	44000	15-MAY-13	PERM
40	IBSEN	201	49500	20-MAY-13	PERM
50	MILES	401	25000	20-JUN-13	TEMP
60	TANG	401	27500	15-JUL-12	PERM
70	KREMER	501	55000	15-JUL-12	PERM
80	PAERT	201	71500	15-JUL-12	PERM
90	JARRET	301	60000	08-AUG-15	TEMP

9 rows selected.

Demo 12: Updating rows in a table using a case function.

```
update ac_emp
set salary = salary * ( 1 + case
    when salary < 40000 then 0.25
    when salary < 50000 then 0.10
    else 0.05 end ) ;
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
10	FREUD	301	41250	06-JUN-02	PERM
20	MATSON	201	69300	06-JUN-10	PERM
30	HANSON	201	48400	15-MAY-13	PERM
40	IBSEN	201	54450	20-MAY-13	PERM
50	MILES	401	31250	20-JUN-13	TEMP
60	TANG	401	34375	15-JUL-12	PERM
70	KREMER	501	57750	15-JUL-12	PERM
80	PAERT	201	75075	15-JUL-12	PERM
90	JARRET	301	63000	08-AUG-15	TEMP

Demo 13: This uses a subquery to allow a test for the proper rows to update. The subquery uses one table to provide the filter to update a different table.

```
update ac_emp
set salary = salary * 1.1
where d_id in
    (select d_id
     from ac_dept
     where d_state = 'CA');
```

Demo 14: Another use of a subquery in an update. This uses an inline view and the inline view filters for the rows to be updated.

```
update (
    select salary
    from ac_emp
    join ac_dept using (d_id)
    where d_state = 'CA'
)
set salary = salary * 1.1;
```

Demo 15: You can use variables to do the update. You want to start thinking about variables as ways to get data from an application layer program that delivers values to a query.

```
select d_city, d_state, e_name, salary
from ac_dept
join ac_emp on ac_dept.d_id = ac_emp.d_id
order by d_state, d_city;
```

D_CITY	D_STATE	E_NAME	SALARY
SAN FRANCISCO	CA	TANG	41594
SAN FRANCISCO	CA	MILES	37813
CHICAGO	IL	IBSEN	54450
CHICAGO	IL	HANSON	48400
CHICAGO	IL	MATSON	69300
CHICAGO	IL	PAERT	75075
PEKIN	IL	FREUD	41250
PEKIN	IL	JARRET	63000
RENO	NV	KREMER	57750

```
variable city varchar2(15);
variable state char(2);
```

```

exec :city := 'CHICAGO';
exec :state := 'IL';
variable wageDifferential number
exec :wageDifferential := 1.05

update ac_emp
set salary = salary * :wageDifferential
where d_id in
    (select d_id
     from ac_dept
     where d_state = :state and d_city = :city);

select d_city, d_state, e_name, salary
from ac_dept
join ac_emp on ac_dept.d_id = ac_emp.d_id
order by d_state, d_city ;

```

D_CITY	D_STATE	E_NAME	SALARY
SAN FRANCISCO	CA	TANG	41594
SAN FRANCISCO	CA	MILES	37813
CHICAGO	IL	IBSEN	57173
CHICAGO	IL	HANSON	50820
CHICAGO	IL	MATSON	72765
CHICAGO	IL	PAERT	78829
PEKIN	IL	FREUD	41250
PEKIN	IL	JARRET	63000
RENO	NV	KREMER	57750

Demo 16: This updates data in a row using other data in the same row. It subtracts the expenses amount from the budget and then sets the expenses to null. Of course this creates problems for the business since we end up doing arithmetic with nulls and we let one department go over budget.

```

update ac_dept
set
    d_budget = d_budget - d_expenses_to_date
, d_expenses_to_date = null;

select * from ac_dept;

```

D_ID	D_NAME	D_BUDGET	D_EXPENSES_TO_DATE	D_CITY	D_STATE
301	FINANCE	35000		PEKIN	IL
501	SALES	-1000		RENO	NV
201	SALES			CHICAGO	IL
401	ADMIN			SAN FRANCISCO	CA

Return to the original rows for the ac_dept table

Demo 17: We can use a more complex expression for the new values

```

Update ac_dept
Set d_budget = Case
    When d_budget > COALESCE(d_expenses_to_date, 0)
        Then d_budget - COALESCE(d_expenses_to_date, 0)
    Else 0
    End
, d_expenses_to_date = Case
    When d_budget > COALESCE(d_expenses_to_date, 0)
        Then 0
    End

```

```

        Else COALESCE(d_expenses_to_date, 0) - d_budget
    End

```

Where d budget Is Not Null;

D_ID	D_NAME	D_BUDGET	D_EXPENSES_TO_DATE	D_CITY	D_STATE
301	FINANCE	35000	0	PEKIN	IL
501	SALES	0	1000	RENO	NV
201	SALES	35000	0	CHICAGO	IL
401	ADMIN			CHICAGO	IL

Demo 18: Or we could use a simple expression and just increase the budgets by a random value , letting people with no budget have something.

```

update ac_dept
set d_budget = round(dbms_random.value,2)* coalesce(d_budget, 500);

```

D_ID	D_NAME	D_BUDGET	D_EXPENSES_TO_DATE	D_CITY	D_STATE
301	FINANCE	33950	0	PEKIN	IL
501	SALES	0	1000	RENO	NV
201	SALES	21000	0	CHICAGO	IL
401	ADMIN	350		CHICAGO	IL

Demo 19: Or just give up on budgets

```

update ac_dept
set d_budget = null;

```

Discussion

The keyword Update is used to identify the table to be changed. The keyword Set is used to identify the column(s) to be changed and the new value(s) to be used. The new value supplied overwrites the original value. You can change more than one column's value in the row.

Updates are done 'all at once'. If any of the potential changes do not meet the table's description or constraints, then none of the changes are made.

You can use many of the techniques we have for Select statements to get the proper rows to be updated.

As a reminder that SQL is a set-oriented language; Oracle can swap columns with a single query.

```

select * from demo;

```

COL_1	COL_2
1	101
2	202
3	999

```

update demo
set col_1 = col_2,
    col_2 = col_1

```

3 rows updated.

```

select * from demo;

```

COL_1	COL_2
101	1
202	2
999	3

5. Insert into queries

5.1. Add a single new row using a value list

Model to insert a single row into the table specifying the column names.

```
insert into my_table (col1_name, col2_name, . . .)
values (value1_name, value2_name, . . .);
```

The preferred technique for a single row insert is to list the column names in parentheses after the table name, followed by the key word Values and then the list of values/expressions in parentheses. The order in which you list the column and the order of the values must be consistent. Since I am listing the columns in the column_list, the order of the columns in the insert statement does not have to match the order of the columns in the table definition. I do not need to supply a value for a nullable column.

Demo 20: Inserts a single row into the table specifying the column names.

```
insert into ac_dept (d_name, d_id, d_city, d_state, d_budget, d_expenses_to_date)
values ('FINANCE', 258, 'MORTON', 'IL', 50000, 15000);

insert into ac_dept (d_id, d_budget, d_expenses_to_date, d_name, d_city, d_state)
values (745, 15000, 16000, 'SALES', 'ZENO', 'NV');
```

An alternative syntax for the insert skips the column list; in that case the order of the values must match the order of the columns in the table definition and no columns can be skipped.

Demo 21: Insert a single row into the table; the data values must be in the proper order, matching the order of the columns in the table definition. If you do not want to inset a value for a nullable column, use the word null.

```
insert into ac_dept
values (465, 'SALES', 35000, null, 'PEKIN', 'IL');
```

Demo 22: Listing the column names allows you to skip column names if there are no values for that column.

```
insert into ac_dept (d_name, d_id, d_city, d_state)
values ('ADMIN', 654, 'MORTON', 'IL');
```

Demo 23: When the table has default values defined, **you can use the word DEFAULT instead of a value**, or use the column list that omits that column.

```
insert into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
values (101, 'FREUD', 301, DEFAULT, DEFAULT, 'PERM');
```

Demo 24: When the table has default values defined, you can use the word DEFAULT instead of a value, **or use a column list that omits that column**.

```
insert into ac_emp (e_id, e_name, d_id, e_status)
values (102, 'MATSON', 201, 'PERM');
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
101	FREUD	301	30000	08-NOV-15	PERM
102	MATSON	201	30000	08-NOV-15	PERM

Discussion

The keywords Insert Into are followed by the name of the table. You can list the columns of the table in parentheses after the name of the table. Then you use the keyword Values to indicate that the lists of values will follow. Include the data values inside parentheses after the keyword Values.

If you are going to insert a value into each column of the row and you are going to list the data values in the same order as the column order in the table, then you can use the second version shown and skip the column name list. This is not as safe as the first version since the table structure can be changed to add additional columns. From a logical point of view, the order of the columns in a table cannot be logically significant and this shorthand version relies on the column order.

If you are inserting a row that allows a null in a specific column, you use the word null in the value list to insert the null.

The data value you are trying to insert must fit the column definition. Any constraints you have placed on the table must be met.

Oracle will not truncate a string to fit into a VARCHAR attribute if it is longer than the defined attribute. Character literals must be enclosed in quotes. The case of the data values is maintained in the table.

A numeric value may not be larger in magnitude than the defined column. If there are more digits after the decimal than was defined, Oracle will round the number to fit. So if the column was defined as NUMBER (5,2) you can insert a value 123.4567 but you cannot insert 1234.567

A date value is traditionally written as a string literal in the default format used by the DBMS – such as '06-AUG-2007'. Or you can use the To_Date function with a format string. This is safer in case the default string format is changed. You can also use the ansi standard date format- such as date '2005-05-19'

You can use SYSDATE as a value to insert the current date-and-time into a column.

These rules also apply to updates- you cannot update a row in a table in a way that violates the table constraints. Oracle does not support a multi-row insert you may know from other dbms,

Demo 25: Our tables at this time. ac_dept, ac_emp

D_ID	D_NAME	D_BUDGET	D_EXPENSES_TO_DATE	D_CITY	D_STATE
201	SALES	21000		CHICAGO	IL
258	FINANCE	50000	15000	MORTON	IL
301	FINANCE	33950		PEKIN	IL
401	ADMIN	350		CHICAGO	IL
465	SALES	35000		PEKIN	IL
501	SALES	0	1000	RENO	NV
654	ADMIN			MORTON	IL
745	SALES	15000	16000	ZENO	NV
8 rows selected.					

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
10	FREUD	301	30000	06-JUN-02	PERM
20	MATSON	201	60000	06-JUN-10	PERM
30	HANSON	201	40000	15-MAY-13	PERM
40	IBSEN	201	45000	20-MAY-13	PERM
50	MILES	401	25000	20-JUN-13	PERM
60	TANG	401	25000	15-JUL-12	TEMP
70	KREMER	501	50000	15-JUL-12	TEMP
80	PAERT	201	65000	15-JUL-12	TEMP
90	JARRET	301	60000	08-AUG-15	TEMP
101	FREUD	301	30000	08-NOV-15	PERM
102	MATSON	201	30000	08-NOV-15	PERM
11 rows selected.					

6. Append rows to an existing table

/* Model to insert row from one table into another.

```
insert into my_table (col1_name, col2_name, . . . )
    select value1_expr, value2_expr , . .
        from . . . ;
```

Assumptions: We have a table of potential hires (AC_PotentialHire). We have added rows to this table with nulls for the Hiredate. If we decide to actually hire someone, we put a value into the HireDate column. We want to add anyone from that table with an actual hire date into our employee table and then remove them from the potential hire table.

Demo 26:

```
Create table ac_potentialHire ( e_id number(3)
, e_name varchar2(15) not null
, d_id number(3)
, salary number(5)
, hiredate date
, e_status char (4)
);

insert into ac_potentialHire (e_id, e_name, d_id, salary, hiredate, e_status)
    values( 201, 'ROBYN', 301, 20000, '15-NOV-2015', 'TEMP');
insert into ac_potentialHire (e_id, e_name, d_id, salary, hiredate, e_status)
    values( 202, 'ECHART', 201, 20000, null, 'TEMP');
insert into ac_potentialHire (e_id, e_name, d_id, salary, hiredate, e_status)
    values( 203, 'TATUM', 301, 25000, '18-NOV-2015', 'TEMP');
```

The table AC_PotentialHire

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_ST
201	ROBYN	301	20000	15-NOV-15	TEMP
202	ECHART	201	20000		TEMP
203	TATUM	301	25000	18-NOV-15	TEMP

The next two demos should be considered as a single transaction.

Demo 27: Append rows from sc_potentialhires into ac_emp. This inserts 2 rows

```
insert into ac_emp (e_id, e_name, d_id, salary, hiredate, e_status)
    select e_id, e_name, d_id, salary, hiredate, e_status
        from ac_potentialhire
        where hiredate is not null;
```

This uses the keywords Insert Into ... followed by a subquery that returns the data for the rows. With this syntax for insert, you do not use the keyword values- you simply include the subquery.

Demo 28: Delete these rows from the potential hire table.

```
delete
from ac_potentialhire
where hiredate is not null;
```

```
Select * from ac_emp;
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
30	HANSON	201	40000	15-MAY-13	PERM
40	IBSEN	201	45000	20-MAY-13	PERM
50	MILES	401	25000	20-JUN-13	PERM
10	FREUD	301	30000	06-JUN-02	PERM
20	MATSON	201	60000	06-JUN-10	PERM
60	TANG	401	25000	15-JUL-12	TEMP
70	KREMER	501	50000	15-JUL-12	TEMP
80	PAERT	201	65000	15-JUL-12	TEMP
90	JARRET	301	60000	08-AUG-15	TEMP
101	FREUD	301	30000	08-NOV-15	PERM
102	MATSON	201	30000	08-NOV-15	PERM
201	ROBYN	301	20000	15-NOV-15	TEMP
203	TATUM	301	25000	18-NOV-15	TEMP

```
Select * from ac_potentialhire;
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_ST
202	ECHART	201	20000		TEMP

The use of the column list is optional if the subquery select clause include all of the columns in the correct order. You can also use Select * in the subquery if the select columns are in the correct order. The columns in the subquery can include calculated columns. The subquery could include a join clause.

Note that a query is done “all at once”. If any of the potential new rows do not meet the table's description or constraints, then none of the rows are added.

7. Create table As queries

This creates a new table and inserts rows into the table.

Create A New Table From An Existing Table

```
CREATE TABLE NewTableName AS
  SELECT ColumnExpressions
    FROM ExistingTableName
   WHERE Condition;
```

Demo 29: Creates a new table with rows from an existing table.

```
create table ac_emp_d201 as
  select *
    from ac_emp
   where d_id = 201;
```

```
select * from ac_emp d201;
```

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
30	HANSON	201	40000	15-MAY-13	PERM
40	IBSEN	201	45000	20-MAY-13	PERM
20	MATSON	201	60000	06-JUN-10	PERM
80	PAERT	201	65000	15-JUL-12	TEMP
102	MATSON	201	30000	08-NOV-15	PERM

Demo 30: Creates a new table with specified attributes from the rows in existing tables.

```
create table ac_emp_IL as
  select e_name, salary, d_city, d_state
    from ac_dept
   join ac_emp using (d_id)
   where d_state = 'IL' ;
```

E_NAME	SALARY	D_CITY	D_STATE
HANSON	40000	CHICAGO	IL
IBSEN	45000	CHICAGO	IL
MILES	25000	CHICAGO	IL
FREUD	30000	PEKIN	IL
MATSON	60000	CHICAGO	IL
TANG	25000	CHICAGO	IL
PAERT	65000	CHICAGO	IL
JARRET	60000	PEKIN	IL
FREUD	30000	PEKIN	IL
MATSON	30000	CHICAGO	IL
ROBYN	20000	PEKIN	IL
TATUM	25000	PEKIN	IL

Demo 31: Suppose you did not want all the columns and you wanted to use different column names.
 You could modify the select to include only the columns you want and the expressions you want
 and use the column aliases to set up the new names.

```
create table ac_emp_2 as
  select e_id as EMPID
    , initcap(e_name) as EMPNAME
    , extract( year from hiredate) as EMPHIREYEAR
   from ac_emp
  where d_id = 201;
```

```
select * from ac_emp_2;
```

EMPID	EMPNAME	EMPHIREYEAR
30	Hanson	2013
40	Ibsen	2013
20	Matson	2010
80	Paert	2012
102	Matson	2015

```
Desc ac_emp_2;
```

Name	Null?	Type
EMPID		NUMBER(3)
EMPNAME		VARCHAR2(15)
EMPHIREYEAR		NUMBER

Demo 32: Creates an empty table that has the same columns as an existing table. The Where clause is just to have a False criteria so that no rows are returned into the new table.

```
create table ac_potentialhire_2 as
  select *
    from ac_emp
   where 1=2;
```

Using this technique to create tables does not copy over check constraints, primary or foreign keys, unique keys, column default values. You can use the Alter Table commands to create the needed constraints. It does keep the data types.

If you do desc of ac_emp and of ac_potentialhire_2, they will appear mostly the same.

```
desc ac_emp
```

Name	Null?	Type
E_ID	NOT NULL	NUMBER(3)
E_NAME	NOT NULL	VARCHAR2(15)
D_ID		NUMBER(3)

SALARY		NUMBER (5)
HIREDATE		DATE
E_STATUS		CHAR (4)
desc ac_potentialhire_2;		
Name	Null?	Type
E_ID		NUMBER (3)
E_NAME	NOT NULL	VARCHAR2(15)
D_ID		NUMBER (3)
SALARY		NUMBER (5)
HIREDATE		DATE
E_STATUS		CHAR (4)

But you can insert rows into ac_potentialhire_2 that do not follow the ac_emp constraints. For example:

```
Insert Into ac_potentialhire_2 (e_id, e_name, d_id, salary, hiredate, e_status)
Values (10, 'FREUD', 301, Default, '06-Jun-2002', 'PERM');
```

```
Insert Into ac_potentialhire_2 (e_id, e_name, d_id, salary, hiredate, e_status)
Values (10, 'FREUD', 301, Default, '06-Jun-2002', 'PERM');
```

```
Insert Into ac_potentialhire_2 (e_id, e_name, d_id, salary, hiredate, e_status)
Values (21, 'FREUD', 7, 35.12, '06-Jun-1847', 'unkn');
```

```
Insert Into ac_potentialhire_2 (e_id, e_name, d_id, salary, hiredate, e_status)
Values (10, 'FREUD', 301, Default, '06-Jun-2002', 'PERM');
```

We have duplicate values for e_id, the salary default from ac_emp was not followed; someone was hired in 1847 with a status of unkn and a salary of 35.

E_ID	E_NAME	D_ID	SALARY	HIREDATE	E_STATUS
10	FREUD	301		06-JUN-02	PERM
10	FREUD	301		06-JUN-02	PERM
21	FREUD	7	35	06-JUN-47	unkn
10	FREUD	301		06-JUN-02	PERM

4 rows selected.

Table of Contents

1. Views	1
1.1. Reason for views	1
1.2. Example of creating and using a view	1
1.3. View versus CTE or subqueries	3
2. Some view syntax rules.....	3
3. Updating with Views	4
3.1. Determining which columns in a view can be changed.....	5
3.2. Updatable view using a join	6
4. Views that are more limiting.....	8
4.1. Declaring the view to be read-only	8
4.2. Making some columns not updatable.....	8
4.3. Updates with check constraints	9

1. Views

1.1. Reason for views

Suppose you have a fairly complex query dealing with customer orders that you need to run often. The query joins several tables and uses calculated columns etc. When you run this query, you filter for different order date ranges but the rest of the query is the same. You have asked the dba for help getting the query to be efficient and you do not want to keep writing the query over each time you need to run one of these reports. What you can do is create a database object called a view which encapsulates the complex query but still lets you add a filter for the order dates when you run that query. When you create your query, you can use the view in the From clause and not worry about the details of the join and calculated columns (not worrying about the details is a good thing!). You could also make the view available to programmers who may not have the experience to write complex queries.

A view is essentially a named, saved Select query. The view does not store any data; instead it stores the directions (the query) for getting the data. Once the view is created, you can use it as the table source for other queries. The definition of the view is stored on the server and is retrieved when the view is used. Each time you use the view, it goes to the actual base tables and retrieves the current data. The view metadata is stored in the data dictionary.

There are a few common reasons for using views

- to create a virtual table that presents a simplified set of column for queries. The view code might include joins and filters and calculated columns that the user of the view does not have to deal with directly
- to separate the logical database structure from the physical structure. The physical structure includes the objects create with the Create Table statement. The logical structure includes those tables and also the views. That way different users can have a different logical structure that reflects the user's needs.
- to improve security by letting a user see only certain columns and specified rows of a table.

1.2. Example of creating and using a view

Demo 01: We will use a fairly simple query for our view. When you work on a view, work out the details of the select statement first.

```
create or replace view custReportGoodCredit_01 as
select customer_id as CustomerID
, customer_name_first || ' ' || customer_name_last as CustomerName
from cust_customers
where customer_credit_limit > 3000
with read only;
```

Demo 02: Using the view

```
select CustomerName, CustomerID
from custReportGoodCredit_01
order by CustomerID;
```

CUSTOMERNAME	CUSTOMERID
Arnold McGold	400300
Sally Williams	403000
Elisha Otis	403010
Alexis Hamilton	403050
James Stevenson	403100
... rows omitted	

Note that the person using this view cannot see the credit limit value because it is not exposed by the view.

Data displayed through a view will reflect the current data in the table at the time the query was run; not the data at the time the view was defined. A person using a view as the table expression might not be aware that this is a view and not a base table.

Demo 03: Creating a view that concatenates the customer names and calculates the linetotal

```
create or replace view ordreport_01 as
select
    order_id      as orderid
,   order_date    as orderdate
,   customer_id   as customerid
,   customer_name_first || ' ' || customer_name_last as customername
,   prod_id       as itempurchased
,   prod_name     as itemdescription
,   quoted_price * quantity_ordered as linetotal
from cust_customers
join oe_orderHeaders using (customer_id)
join oe_orderDetails using(order_id)
join prd_products using(prod_id)
where quoted_price > 0 and quantity_ordered > 0 with read only;
```

Demo 04: Describing the view. One thing to check is that the data types of the columns in the view are appropriate. Do not over format the columns you wish to filter; do not output a formatted number column if you might wish to filter on it. Remember that when you format a numeric column with To_Char you are returning a string- not a number.

Desc ordReport_01

Name	Null?	Type
ORDERID	NOT NULL	NUMBER(38)
ORDERDATE	NOT NULL	DATE
CUSTOMERID	NOT NULL	NUMBER(38)
CUSTOMERNAME		VARCHAR2(51)
ITEMPURCHASED	NOT NULL	NUMBER(38)
ITEMDESCRIPTION	NOT NULL	VARCHAR2(25)
LINETOTAL		NUMBER

Demo 05: Using the view this is a lot easier than repeating that join for each query

```
select orderid, orderdate, itempurchased, linetotal
from ordreport_01;
```

ORDERID	ORDERDATE	ITEMPURCHASED	LINETOTAL
105	01-OCT-15	1020	155.4

105 01-OCT-15	1030	300
105 01-OCT-15	1010	750
106 01-OCT-15	1060	255.95
107 02-OCT-15	1110	49.99
. . . rows omitted		

Demo 06: Using the view with a filter

```
Select
    orderid
, orderdate
, itempurchased
From ordreport_01
Where itempurchased in ( 1071, 1072)
order by orderdate;
```

ORDERID	ORDERDATE	ITEMPURCHASED
411	01-AUG-15	1071
411	01-AUG-15	1072
408	20-NOV-15	1071
2122	23-JAN-16	1071
2122	23-JAN-16	1072
4510	01-FEB-16	1071
4510	01-FEB-16	1072

Demo 07: Alternate syntax when creating a view- this defines the column names in the view header rather than in the select

```
create or replace view ordreport_02(orderid, orderdate, customerid, linetotal)
as
select order_id, order_date, customer_id
, quoted_price * quantity_ordered
from oe_orderheaders
join oe_orderdetails using(order_id)
join prd_products using(prod_id)
where quoted_price > 0 and quantity_ordered > 0
order by order_id
with read only;
```

1.3. View versus CTE or subqueries

You may have noticed a similarity between CTE and views. A view is stored in the database as a persistent database object; a CTE exists for the lifetime of the query. So if you have a result set that you want recalculated each time you use it and you would use that result set often or you want other users to use that result set, then a View is a good idea. If you want to break a complex task into pieces to solve one step at a time and the task is unique and it is not sometime you want for other tasks, then a CTE is a good idea.

A production database will often have a large number of views set up for business purposes but you don't want to create a view that is used only once.

2. Some view syntax rules

A view is an object created in a specific database.

- You can use either the Create View syntax or the Create or Replace View syntax. If the name you select for the view is already used by another view, the Create view syntax will report an error; the Create or Replace View syntax will replace the current definition of the view.

- You cannot have a view with the same name as a table.
- You cannot create a view on a table that does not exist yet unless you include the word Force. You cannot use the view until the table is created, but this keyword will let you define the view.
- Since a view is a stored object, you can drop the view with a drop view command.
`drop view my_view;`
- Your user account has to have permissions to create views. Your CCSF Oracle account has that permission.
- You can assign new names for the columns when you define a view or you can use the names in the underlying tables.
- If you have a calculated column in the underlying select, you must provide an alias for that column.
- If you define the columns in the view header, the number of columns in the view definition must match the number of columns in the select used to define the view.
- When you create an alias within a view, the alias should be a single word and not enclosed in quotes
- Do not create a view using Select *; the view will be stored with the * expanded into the column names at the time the view was created and will not be updated if the underlying tables are changed.
- Oracle allows an Order By clause in the definition of a view

3. Updating with Views

Some restrictions for updating (any change) through a view. We will discuss some of these options in this document. If you think about these rules and the need for the dbms to understand exactly what data to change, these rules make more sense.

- If the view is based on all of the columns of a single table and the view was not created with a read-only option or a with check option, then you can update through the view.
- A read only option means that the view cannot be used for updates
- A With Check option means that you cannot do an update that would violate the check option.
- You cannot update a column in a view which is based on a calculated column. If the view has several columns and only some columns are calculated, you can update the other columns
- If you have any calculated columns, then you cannot insert via the view. However you might be able to delete via the view.
- You cannot update views that are defined with aggregate functions or Group By
- You cannot update views that are defined with Distinct or rownum
- You cannot update a view created with a set operation such as Union
- If the view is based on multiple tables, you cannot do an update on a column which maps to a non key-preserved table. (Essentially this means that if the view definition does not contain the pk – as a pk for the view set, then the view cannot be updated. So if we had a parent and child situation and the view definition includes the pk of the parent we would, in general, be able to update the parent. But we could not update the child even with the child table pk since it could occur more than once in the view rowset)

Demo 08: In order to not change the data in the regular table, insert the following rows for testing.

An Insert for the employees table

```
delete from emp_employees where emp_id = 1995;
insert into emp_employees values
(1995, 'Zahn', 'Joe', '111111111', 100, 10, date'2009-09-09', 500, 2);
```

An Insert for the products table

```
delete from prd_products where prod_id = 1995;
insert into prd_products values
(1995, 'book', 'Train your cat book', 29.95, 'PET');
```

3.1. Determining which columns in a view can be changed

Demo 09: What is the structure of emp_employees?

```
desc emp_employees
```

Name	Null?	Type
EMP_ID	NOT NULL	NUMBER (38)
NAME_LAST	NOT NULL	VARCHAR2 (25)
NAME_FIRST		VARCHAR2 (25)
SSN	NOT NULL	CHAR (9)
EMP_MNG		NUMBER (38)
DEPT_ID	NOT NULL	NUMBER (38)
HIRE_DATE	NOT NULL	DATE
SALARY		NUMBER (8,2)
JOB_ID	NOT NULL	NUMBER (38)

Demo 10: Simple, single table view. This view can be used for updates of the employee name and the id

```
CREATE OR REPLACE VIEW vw_Emp AS
select emp_id, name_last, name_first
from emp_employees;
```

Demo 11: Which columns can be changed? Use the data dictionary view user_updatable_columns

```
select column_name, updatable
from user_updatable_columns
where table_name = 'VW_EMP';
```

COLUMN_NAME	UPDATABLE
EMP_ID	YES
NAME_LAST	YES
NAME_FIRST	YES

Discussion: This update works.

```
Update vw_emp set name_first = 'Susan' where emp_id = 1995;
```

This one does not work; the attribute salary is not accessible through the view.

```
Update vw_emp set salary = 12500 where emp_id = 1995;
```

```
ORA-00904: "SALARY": invalid identifier
```

This update does not work; the attribute name_last is accessible through the view, but that attribute is not nullable in the base table. The table constraints must always be met.

```
Update vw_emp set name_last = null where emp_id = 1995;
```

```
ORA-01407: cannot update ("ROSE151A"."EMP_EMPLOYEES"."NAME_LAST") to NULL
```

Can you do an insert via this view? No- there are several attributes in the table which are not nullable and not accessible via the view.

Can you do a delete via this view? Yes, unless there is another constraint that would disallow the delete.

```
Delete from vw_emp where emp_id = 1995;
```

If you have removed or changed that row, reinsert it.

Demo 12: This is a view that shows summary data

```
CREATE OR REPLACE VIEW vw_HighPriceByCategory AS
select Catg_id, MAX(prod_list_price) As HighPrice
from prd_products
group by Catg_id;
```

Which columns can be changed? You cannot change data in a view with a group by.

```
select column_name, updatable
from user_updatable_columns
where table_name = 'VW_HIGHPRICEBYCATEGORY';
```

COLUMN_NAME	UPDATABLE
CATG_ID	NO
HIGHPRICE	NO

If we try to change data through the view , we get an error because the attribute HighPrice does not tie back to a specific row in the underlying table

```
update vw_HighPriceByCategory
set HighPrice = 250
where Catg_id = 'PET';
ORA-01732: data manipulation operation not legal on this view
```

Demo 13: This is a view that includes a calculated column

```
CREATE OR REPLACE VIEW vw_NewPrice AS
select Catg_id, prod_name, Round(prod_list_price * 1.05,2) As NewPrice
from prd_products;
```

Some of these columns can be updated but we cannot update the calculated column.

```
select column_name, updatable
from user_updatable_columns
where table_name = 'VW_NEWPRICE';
```

COLUMN_NAME	UPDATABLE
CATG_ID	YES
PROD_NAME	YES
NEWPRICE	NO

Trying to update the calculated column is blocked.

```
Update vw_NewPrice
set NewPrice = 45
where prod_name = 'Train your cat book';
ORA-01733: virtual column not allowed here
```

Demo 14: This is another view that includes a calculated column

```
CREATE OR REPLACE VIEW vw_NewPrice2 AS
select Catg_id, prod_name, prod_list_price * 1 As NewPrice
from prd_products;
```

If you think about this mathematically, prod_list_price * 1 does not really change the value, but this is still a calculated column and cannot be updated via the view.

```
Update vw_NewPrice2
set NewPrice = 45
where prod_name = 'Train your cat book';
ORA-01733: virtual column not allowed here
```

3.2. Updatable view using a join

Create a view that joins the department table (the parent) and the employee table (the child). Can we update columns in the parent? in the child? There are several different ways to do the join. The next three views use different join techniques

Demo 15: Creating the join with the USING clause.

```
CREATE OR REPLACE VIEW vw_em_dept_1 AS
  select emp_id, name_last
  , dept_id
  , dept_name
  from emp_employees join emp_departments using(dept_id);
```

Demo 16: Creating the join with the ON clause including the dept_id column from the child table.

```
CREATE OR REPLACE VIEW vw_em_dept_2 AS
  select emp_id, name_last
  , emp_employees.dept_id
  , dept_name
  from emp_employees
  join emp_departments
  on emp_employees.dept_id = emp_departments.dept_id;
```

Demo 17: Creating the join with the ON clause including the dept_id column from the parent table.

```
CREATE OR REPLACE VIEW vw_em_dept_3 AS
  select emp_id, name_last
  , emp_departments.dept_id
  , dept_name
  from emp_employees
  join emp_departments
  on emp_employees.dept_id = emp_departments.dept_id;
```

If you do a Select * for any of these three views you get the same result set.

Demo 18: But there is a difference. Which columns are updatable ? (The Break command is used to make the output easier to read)

```
BREAK ON table_name SKIP 1

select table_name, column_name, updatable
from user_updatable_columns
where table_name LIKE 'VW_EM_DEPT %' ;
```

TABLE_NAME	COLUMN_NAME	UPDATABLE
VW_EM_DEPT_1	EMP_ID	YES
	NAME_LAST	YES
	DEPT_ID	NO
	DEPT_NAME	NO
VW_EM_DEPT_2	EMP_ID	YES
	NAME_LAST	YES
	DEPT_ID	YES
	DEPT_NAME	NO
VW_EM_DEPT_3	EMP_ID	YES
	NAME_LAST	YES
	DEPT_ID	NO
	DEPT_NAME	NO

(This is rather complex- I mostly want you to understand that if a view is based on a join of tables, updating gets more complex.)

In all of these we can update the Emp_id and name_last which come from the child table. For each Emp_id or Emp_name, that value ties back to a single definable row in the employees table. There is only one row in the employees table for each row in the view results table. (the term 'key-preserved" is used for a base table that

has a 1-1 relationship with the rows in the views- therefore a row in the view can be unambiguously tied to a row in that base table.) We cannot update the dept_name which comes from the parent table; there are multiple rows in the result set for a row in the departments table.

The dept_id can be updated in the second view- where that attribute in the view comes uniquely from the child table. The dept_id cannot be updated in the third view- where that attribute in the view comes from the parent table and is not unique. The dept_id cannot be updated in the first view- where that attribute in the view comes from the new generated column and not from either table.

Demo 19: An attempt to update the dept_id using the third view. This fails.

```
update vw_em_dept_3
set dept_id = 80
where emp_id = 1995;
set      dept_id = 80
*
ERROR at line 2:
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

Discussion: We can do the update via the second view. In that case we are changing the dept id for child table- (the employee table). So this changes a single row. If we try to do the change via the third view, we would be changing the dept_id in the department table- so presumably we would be changing dept_id 10 to 80 in the department table- that would result in two rows in the department table with ID 80. Allowing this type of change would create problems. (Again, if you did the update, do a rollback.)

4. Views that are more limiting

There are some options that can be used in the creation of a view that provide protection of the data in the base table.

4.1. Declaring the view to be read-only

Demo 20: Declaring that the view is read only. None of the attributes are updatable.

```
CREATE OR REPLACE VIEW ProductList AS
  select prod_id, prod_list_price, prod_name, prod_desc
  from prd_products
WITH READ ONLY;
```

4.2. Making some columns not updatable

Demo 21: The list price and the description can be updated but not the id or name. because these are calculated columns

```
CREATE OR REPLACE VIEW ProductList2 AS
  select  prod_id + 0 as prod_id
        , prod_list_price as prod_list_price
        , prod_name || '' as prod_name
        , prod_desc
  from    prd_products;

  select column_name, updatable
  from user_updatable_columns
  where table_name = 'PRODUCTLIST2';
```

COLUMN_NAME	UPD
PROD_ID	NO
PROD_LIST_PRICE	YES

PROD_NAME	NO
PROD_DESC	YES

4.3. Updates with check constraints

The next few SQL statements show the purpose of the With Check Option clause. Suppose I create a view that has a WHERE clause. I can use the view to update the data in such a way that the data no longer satisfies that WHERE clause. The With Check Option clause restricts this behavior.

Demo 22: Creating a view with a check constraint

```
CREATE OR REPLACE VIEW EmpDept80 AS
    select emp_id, name_last
    , dept_id
  from emp_employees
 where dept_id = 80
WITH CHECK OPTION;
```

Which columns can be changed?

COLUMN_NAME	UPD
EMP_ID	YES
NAME_LAST	YES
DEPT_ID	YES

Demo 23: It is OK to change the last name value through this view. (Be certain that you have employee 1995 in dept 80 before you run this one.)

```
update EmpDept80
set name_last = 'Hiller'
where emp_id = 1995;
```

Demo 24: It is not OK to change the department to a value that does not meet the check constraint.

```
update EmpDept80
set dept_id = 20
where emp_id = 1995;
```

ORA-01402: view WITH CHECK OPTION WHERE -clause violation

Demo 25: Creating a view with a check constraint

```
CREATE OR REPLACE VIEW EmpHighSalary AS
    select emp_id, name_last, salary
    , dept_id
  from emp_employees
 where salary > 70000
WITH CHECK OPTION;

-- the record set
select *
from EmpHighSalary;
```

EMP_ID	NAME_LAST	SALARY	DEPT_ID
100	King	100000	10
101	Koch	98005	30
205	Higgs	75000	30
162	Holme	98000	35
146	Partne	88954	215
161	Dewal	120000	215

206 Geitz	88954	30
204 King	99090	30

8 rows selected.

Demo 26: We can update employee 1995 salary via the employees table and then that employee will appear via the view.

```
update emp_employees
set name_last = 'Prince',
    salary = 75000
where emp_id = 1995;
```

```
select *
from EmpHighSalary ;
```

EMP_ID	NAME_LAST	SALARY	DEPT_ID
100	King	100000	10
101	Koch	98005	30
205	Higgs	75000	30
162	Holme	98000	35
146	Partne	88954	215
161	Dewal	120000	215
206	Geitz	88954	30
204	King	99090	30
1995	Prince	75000	10

9 rows selected.

Demo 27: Updating through the view. This will do the update.

```
update EmpHighSalary
set name_last = 'Prince',
    salary = 85000
where emp_id = 1995;
```

1 row updated.

Demo 28: Updating through the view. This update will fail due to the check constraint

```
update EmpHighSalary
set name_last = 'Koch',
    salary = 10000
where emp_id = 1995;
```

ORA-01402: view WITH CHECK OPTION where-clause violation

Similarly you cannot do an Insert via a view with a check constraint if that insert would not be displayed with the view- i.e. if the insert data does not meet the Where clause criterion.

It would be a good idea to reload the Altgeld tables at this time.

Increasingly people are expecting answers to more complex questions based on the data in their databases. This includes tasks such as ranking, calculating moving averages and rolling up data. The queries at first might seem difficult but these are commonly needed business tasks.

Some of the analytical techniques we have used so far include

- the set functions- Count, Max, Min, Sum, Avg - which work on a collection of rows as a unit.
- Group By and Having- which also produce or work with collections of rows as a unit

To actually analyze our data we need more analytical techniques.

First some examples. Suppose we have a table of student grades for each assignment. Each student has 10 assignment scores and we have three students. SQL for this is in the demo, but not listed here. Here I just want you to see what type of results we might get from the data and the general name for that type of query. I do not expect you to figure out how that code works- yet.

We can list the data sorted by the student ID and the assignment numbers. OK that is all the data but it is not information in terms of looking at the data at a somewhat higher level.

STU	ASGN	SCORE
101	1	50
101	2	50
101	3	40
101	4	45
101	5	40
101	6	47
101	7	45
101	8	30
101	9	45
101	10	48
201	1	0
201	2	50
201	3	0
201	4	0
201	5	5
201	6	30
201	7	35
201	8	30
201	9	37
201	10	30
301	1	50
301	2	50
301	3	45
301	4	45
301	5	40
301	6	40
301	7	42
301	8	30
301	9	30
301	10	30

30 rows selected.

(Ranking) Suppose we want to show the top 3 assignment scores for each student. Note the scores for student 301. This student has 2 scores at 50 and then 2 at 45; since the 3rd and 4th score are tied for the third place, both of these are shown.

STU	SCORE	ASGN
101	50	1

101	50	1
101	50	2
101	48	10
201	50	2
201	37	9
201	35	7
301	50	1
301	50	2
301	45	3
301	45	4

(Rollup) We could show students and their total and average scores.

STU	TOTSCORE	AVGSCORE
101	440	44
201	217	21.7
301	402	40.2

But we can also include the aggregates for all of the students using a rollup. The total scores for all students is not terrible useful so I hide it.

STUDENT	TOTSCORE	AVGSCORE
101	440	44.00
201	217	21.70
301	402	40.20
All students		35.30

We could add the details to that report

STUDENT	ASGN	SCORE
101	1	50
101	2	50
101	3	40
101	4	45
101	5	40
101	6	47
101	7	45
101	8	30
101	9	45
101	10	48
101	AVG	44
201	1	0
201	2	50
201	3	0
201	4	0
201	5	5
201	6	30
201	7	35
201	8	30
201	9	37
201	10	30
201	AVG	22
301	1	50
301	2	50

301	3	45
301	4	45
301	5	40
301	6	40
301	7	42
301	8	30
301	9	30
301	10	30
301	AVG	40
All students	AVG	35

STUDENT	ASGN	Score/Avg
101	1	50
101	2	50
101	3	40
101	4	45
101	5	40
101	6	47
101	7	45
101	8	30
101	9	45
101	10	48
101	AVG	44.00
201	1	0
201	2	50
201	3	0
201	4	0
201	5	5
201	6	30
201	7	35
201	8	30
201	9	37
201	10	30
201	AVG	21.70
301	1	50
301	2	50
301	3	45
301	4	45
301	5	40
301	6	40
301	7	42
301	8	30
301	9	30
301	10	30
301	AVG	40.20
All students	AVG	35.30

34 rows selected.

(Running Total) Maybe you want to see the scores for a student but also see the running total of their scores. This is just student 101.

ASGN	SCORE	RUNNINGTOTAL
101	50	50

1	50	50
2	50	100
3	40	140
4	45	185
5	40	225
6	47	272
7	45	317
8	30	347
9	45	392
10	48	440

(Running Total) Maybe you want to see the scores for each student but also see the running total of their scores.. We need to start the running total over for each student.

StID	Asgn#	Score	RunningTotal
101	1	50	50
101	2	50	100
101	3	40	140
101	4	45	185
101	5	40	225
101	6	47	272
101	7	45	317
101	8	30	347
101	9	45	392
101	10	48	440
201	1	0	0
201	2	50	50
201	3	0	50
201	4	0	50
201	5	5	55
201	6	30	85
201	7	35	120
201	8	30	150
201	9	37	187
201	10	30	217
301	1	50	50
301	2	50	100
301	3	45	145
301	4	45	190
301	5	40	230
301	6	40	270
301	7	42	312
301	8	30	342
301	9	30	372
301	10	30	402

Sometimes these problems were solved by using a programming approach or very complex SQL. The various dbms have been adding more features and functions to SQL to help solve these problems. When these techniques are built into the dbms, they can be optimized by the optimizer and using these functions is generally more efficient than other approaches

Many of these tasks involve the use of analytical functions which can be as simple as the Avg. Sum and Count functions we have been using. We can use these functions to calculate aggregate values over a group of rows.

Many of these techniques use a windowing clause and ranking. Windowing uses an Order By clause and might also partition the data into groups based on a criteria such as department id or by year hired. These subsets of data are called window partitions.

Once the data is sorted and possibly partitioned you can use ranking functions to number the rows in the partitions.

Oracle has added a number of functions that do fairly complex calculations; we will use some of these.

One thing that we can do with these techniques is display both detail and aggregated data in the same query. You have done some of this already with subqueries..

These results of these functions are more meaningful with large amounts of data, but to keep the examples simple, we will use small tables. Although these queries might seem complex at first, they are important for analyzing large amounts of data. This unit's material may seem overwhelming but I think you can handle working out the tasks for the assignment.

The demo file for this document also includes the sql for two tables you need to create and populate.

adv_employees

adv_sales The adv_sales table which will have one row for each day's sales for a range of dates.

Table of Contents

1.	RowNum	2
1.1.	Using a subquery	3
2.	Row_Number.....	4
2.1.	Partition By	6
3.	Rank & Dense Rank.....	7
4.	Top N	8
5.	Using a grouping.....	10
6.	Partition By	11

Ranking functions are used to rank rows of data according to some criteria. This is one of the simplest analytical techniques to understand. You probably rank many things. We might want to rank employees by salary or we might want to rank employees by salary within each department. Ranking functions have to consider ties. The ranking functions discussed here are:

- Row_Number
- Rank
- Dense_Rank

These analytic functions **can be used only in the Select list or the Order By clause**. Other parts of the query (join, Where, Group by, Having) are carried out before the analytic functions.

We will review RowNum to avoid confusion. RowNum is **not** a ranking function

I will generally show only a sampling of the rows for each result set. For reference, these are the rows in the table adv_emp.(the sql for these is in demo_01)

EMP_ID	NAME_LAST	DEPT_ID	YEAR_HIRED	SALARY
301	Green	10	2010	15000
302	Hancock	20	2010	14000
303	Quebec	20	2014	27000
304	Mobley	30	2010	28000
305	Coltrane	10	2012	27000
306	Cohen	30	2010	28000
307	Tatum	30	2012	13500
308	Evans	30	2013	15000
309	Beiderbecke	10	2014	30000
310	Wabich	10	2012	25000
311	Brubeck	10	2012	28000
312	Ellington	20	2010	28000
313	Davis	30	2012	11000
314	Turrentine	30	2013	30000
315	Battaglia	20	2013	12000
316	Monk	30	2013	26000
317	Wasliewski	30	2014	25000
318	Shorter	30	2014	11500
319	Redman	10	2014	30000
320	Jarrett	10	2012	25000
321	Rollins	10	2014	30000
322	Wabich	10	2012	25000
323	Montgomery	15	2012	25000

23 rows selected.

1. RowNum

Suppose we run the following three queries:

Demo 01: Displaying RowNum. The rows display the RowNum values in numeric order

```
select emp_id, salary, dept_id, RowNum
from adv_emp;
```

EMP_ID	SALARY	DEPT_ID	ROWNUM
301	15000	10	1
302	14000	20	2
303	27000	20	3
304	28000	30	4
305	27000	10	5
306	28000	30	6
307	13500	30	7
308	15000	30	8
309	30000	10	9
310	25000	10	10
311	28000	10	11
312	28000	20	12
313	11000	30	13
314	30000	30	14
315	12000	20	15
316	26000	30	16
317	25000	30	17
318	11500	30	18
319	30000	10	19
320	25000	10	20
321	30000	10	21
322	25000	10	22
323	25000	15	23

Demo 02: Now add a sorting clause. We still get the RowNum values but they do not appear in numeric order since the rows are sorted by the salary.

```
select emp_id, salary, dept_id, RowNum
from adv_emp
order by salary ;
```

EMP_ID	SALARY	DEPT_ID	ROWNUM
313	11000	30	13
318	11500	30	18
315	12000	20	15
307	13500	30	7
302	14000	20	2
301	15000	10	1
308	15000	30	8
322	25000	10	22
323	25000	15	23
320	25000	10	20
317	25000	30	17
310	25000	10	10
316	26000	30	16
305	27000	10	5
303	27000	20	3

306	28000	30	6
312	28000	20	12
311	28000	10	11
304	28000	30	4
314	30000	30	14
321	30000	10	21
319	30000	10	19
309	30000	10	9

The RowNum values are determined before the sort is applied- so we do not see the RowNum values in row number order. RowNum values reflect something about the physical order of the rows in secondary storage and how they are retrieved from storage and we know that we should never write code based on the physical characteristics of the rows in storage. RowNum is a pseudo column that is generated as the rows are retrieved from secondary storage and we have little control over that. RowNum is Oracle specific.

1.1. Using a subquery

As a first attempt to solve this we could try a subquery; the subquery does the sort and then delivers the sorted rows to the parent query which applies the RowNum.

Demo 03: Getting the data in a subquery in the From clause and using RowNum in the parent query

```
select emp_id, dept_id, salary, rownum
from ( select emp_id, dept_id, salary
       from adv_emp
       order by salary);
```

This does give us the rows ranked ok- except that employees can have the same salary but different row num (such as salary 25000). If you were going to give raises to the ten lowest paid employees- this would be a problem. (Think of this as an automatic raise- not a situation where a human looks at the output and notices the tie!)

EMP_ID	DEPT_ID	SALARY	ROWNUM
313	30	11000	1
318	30	11500	2
315	20	12000	3
307	30	13500	4
302	20	14000	5
301	10	15000	6
308	30	15000	7
322	10	25000	8
323	15	25000	9
320	10	25000	10
317	30	25000	11
310	10	25000	12
316	30	26000	13
305	10	27000	14
303	20	27000	15
306	30	28000	16
312	20	28000	17
311	10	28000	18
304	30	28000	19
314	30	30000	20
321	10	30000	21
319	10	30000	22
309	10	30000	23

There have been various ways to work around these kinds of problems and there is a need to have a uniform way of dealing with this task.

2. Row_Number

This example uses the row_number function and a simple windowing clause. That function produces a new number for each row in the result set. Notice that the Over clause is using to supply the sorting rule. The rows in a table do not have a natural ordering- we need to supply one.

You cannot use the row_number function without also supplying an Over clause. Row_Number is an ansii standard technique.

Demo 04: Using the Row_Number() function

```
Select emp_id, dept_id, salary
, row_number () Over (order by salary ) as col_Order
from adv.emp;
```

EMP_I	DEPT_ID	SALARY	COL_ORDER
313	30	11000	1
318	30	11500	2
315	20	12000	3
307	30	13500	4
302	20	14000	5
301	10	15000	6
308	30	15000	7
322	10	25000	8
323	15	25000	9
320	10	25000	10
317	30	25000	11
310	10	25000	12
316	30	26000	13
305	10	27000	14
303	20	27000	15
306	30	28000	16
312	20	28000	17
311	10	28000	18
304	30	28000	19
314	30	30000	20
321	10	30000	21
319	10	30000	22
309	10	30000	23

23 rows selected.

The query gets data from the adv_emp view and returns each row; there is no Where clause to filter the rows. The row_number function supplies a number for each row returned to the result set.

The row_number function has a different syntax than we are used to for functions; it has a required Over () clause attached to it. This is a "window specification"; it is also referred to as a partition by clause. This clause can contain any of a number of things- here we have an ordering. It is the order by clause in the parentheses following the keyword Over which says to supply the row_numbers in salary order.

The windowing clause lets us calculate moving averages.

We can reset aggregates or ranks when a department changes (a control break report)

We can use multiple functions within a single query.

The Over clause supports three different techniques

- Order the rows by some attribute: Over (order by salary)
- Partition the rows by some attribute: Over (partition by Dept_id order by salary)
- Define a moving window frame: Over (order by day rows between 2 preceding and 1 following)

Change the SQL to sort in descending order and the rows are ranked in descending order.

Demo 05: Row_Number with a descending sort

```
Select emp_id, dept_id, salary
, row_number () Over (order by salary desc nulls last) as col_Order
from adv_emp;
```

Demo 06: You can specify Nulls Last or Null First in the ordering

```
Select emp_id, dept_id, salary
, row_number () Over (order by salary desc nulls first) as col_Order
from adv_emp;
```

Now add a regular order by clause as the last clause in the query. The final order by clause controls the order in which the rows are displayed. It does not affect the value returned by the row_number function. If you have a lot of employees and you want to find their row_number rank, sorting by the employee id can be useful.

Demo 07: Row_Number and sorting the result

```
select emp_id, dept_id, salary
, row_number () Over (order by salary desc ) as col_Order
from adv_emp
order by emp_id;
```

EMP_ID	DEPT_ID	SALARY	COL_ORDER
301	10	15000	17
302	20	14000	19
303	20	27000	10
304	30	28000	6
305	10	27000	9
306	30	28000	8
307	30	13500	20
308	30	15000	18
309	10	30000	3
310	10	25000	12
. . . rows omitted.			

But most of the time you would want to do a final sort by the same attribute as the "over" clause to emphasize the ranking order. Note that we have not solved the problem with the ties yet.

Demo 08: You can include more than one sort key in the window specification. The row_number values still go from 1 to 23- one for each row.

```
select emp_id, dept_id, salary
, row_number () Over (order by dept_id, salary desc ) as col_Order
from adv_emp
order by dept_id, salary desc;
```

EMP_ID	DEPT_ID	SALARY	COL_ORDER
309	10	30000	1
321	10	30000	2
319	10	30000	3
311	10	28000	4
305	10	27000	5
320	10	25000	6
322	10	25000	7
310	10	25000	8
301	10	15000	9
323	15	25000	10
312	20	28000	11
303	20	27000	12
302	20	14000	13
315	20	12000	14
314	30	30000	15
304	30	28000	16
306	30	28000	17
316	30	26000	18
317	30	25000	19
308	30	15000	20
307	30	13500	21
318	30	11500	22
313	30	11000	23

23 rows selected.

2.1. Partition By

There is another option you can use which does a partition by an attribute.

Demo 09: Row number with a partition. Here we are partitioning the data by the department id; for each new department the row_number restarts at 1

```
select emp_id, dept_id, salary
, row_number () Over (partition by dept_id order by salary ) as col_order
from adv_emp
order by dept_id, salary ;
```

EMP_ID	DEPT_ID	SALARY	COL_ORDER
301	10	15000	1
320	10	25000	2
310	10	25000	3
322	10	25000	4
305	10	27000	5
311	10	28000	6
319	10	30000	7
321	10	30000	8
309	10	30000	9
323	15	25000	1
315	20	12000	1
302	20	14000	2
303	20	27000	3
312	20	28000	4

313	30	11000	1
318	30	11500	2
307	30	13500	3
308	30	15000	4
317	30	25000	5
316	30	26000	6
306	30	28000	7
304	30	28000	8
314	30	30000	9

23 rows selected.

3. Rank & Dense Rank

Change the query to use the dense_rank function and then the rank function instead of the row_number function. The windowing clause is ordering by salary. The Rank functions determines the rank of data relative to a group of values. The windowing clause is not changed.

Demo 10: Using the DenseRank() function

```
Select emp_id, dept_id, salary
, dense_rank () Over (order by salary desc nulls last ) as col_Order
from adv_emp;
```

Note the values for the rank column go 1,1,1,1 , 2,2,2,2, 3,3, 4,5,5 because tied rows get the same number, but dense_rank does not skip numbers for ties.

EMP_ID	DEPT_ID	SALARY	COL_ORDER
314	30	30000	1
321	10	30000	1
309	10	30000	1
319	10	30000	1
311	10	28000	2
304	30	28000	2
312	20	28000	2
306	30	28000	2
305	10	27000	3
303	20	27000	3
316	30	26000	4
310	10	25000	5
322	10	25000	5
323	15	25000	5
320	10	25000	5
317	30	25000	5
301	10	15000	6
308	30	15000	6
302	20	14000	7
307	30	13500	8
315	20	12000	9
318	30	11500	10
313	30	11000	11

23 rows selected.

Demo 11: Using the Rank() function

```
Select emp_id, dept_id, salary
, rank () Over (order by salary desc nulls last ) as col_Order
from adv_emp;
```

Note the values for the rank column go 1,1,1,1 5,5,5,5 ,... because there are four rows tied for first place at salary 30000; then it skips 2,3,4, etc The next set of ties all get value 5 and then the count skips to 9.

EMP_ID	DEPT_ID	SALARY	COL_ORDER
314	30	30000	1
321	10	30000	1
309	10	30000	1
319	10	30000	1
311	10	28000	5
304	30	28000	5
312	20	28000	5
306	30	28000	5
305	10	27000	9
303	20	27000	9
316	30	26000	11
310	10	25000	12
322	10	25000	12
323	15	25000	12
320	10	25000	12
317	30	25000	12
301	10	15000	17
308	30	15000	17
302	20	14000	19
307	30	13500	20
315	20	12000	21
318	30	11500	22
313	30	11000	23

You can imagine that people could not agree on which was the "right" way to handle ranks with ties- so they gave us both ways. You need to use the version that makes the most sense to the business situation.

4. Top N

Now we want to get the top six employees in terms of salary. The first problem is to find out what the user wants to do if there are ties. We will assume that the user wants all ties at the last selected position- so we might return 6 rows or more than 6 if there are ties at the last position.

We might try putting the column alias in a Where clause- but that is not valid syntax. Neither can we put the rank or dense_rank function in the Where clause. These functions are allowed in the Select list.

So we go back to a subquery that produces the ranks and a parent query that gets everyone of rank 6 or less. Note that we are using rank, not dense_rank. How many rows would you get with dense_rank?

Demo 12: Top 6 salaries

```
select emp_id, dept_id, salary, col_Order
from (
    select emp_id, dept_id, salary
```

```

        , rank() over (order by salary desc nulls last ) as col_Order
        from adv_emp
    ) tbl
where col_Order <= 6;

```

EMP_ID	DEPT_ID	SALARY	COL_ORDER
309	10	30000	1
321	10	30000	1
319	10	30000	1
314	30	30000	1
304	30	28000	5
312	20	28000	5
311	10	28000	5
306	30	28000	5

8 rows selected.

Demo 13: You can also use the With clause instead of a subquery. I have changed the sort to ascending to get the bottom six salaries

```

with rankings as (
    select emp_Id, dept_id, salary
    , rank() over (order by salary asc nulls last ) as col_Order
    from adv_emp
)
select emp_Id, dept_id, salary, col_Order
from rankings
where col_Order <= 6;

```

EMP_ID	DEPT_ID	SALARY	COL_ORDER
313	30	11000	1
318	30	11500	2
315	20	12000	3
307	30	13500	4
302	20	14000	5
301	10	15000	6
308	30	15000	6

7 rows selected.

Although we think of an SQL statement as being done "all-at-once" there is an ordering in which the various parts of a select query is processed. If we have a query with a Join and an Order by, the From clause is the first part done- so the Joins are done first, then the Where clause is applied, then the Select and finally the Order By. If we have an analytical function added to the query, it is processed after any Join, Where, Group By or Having clause. It is evaluated just before the final Order By clause.

If we run the following query, only the rows for Dept 30 and 20 will be given a rank.

Demo 14: Rank() within selected departments

```

select emp_Id, Dept_id, salary
, rank() Over (order by salary desc ) as col_Order
from adv_emp
where dept_id IN (30, 20);

```

5. Using a grouping

In this query we group by the department ID. We want to show the average salary and we rank over the avg(salary). I truncated the averages to make the output easier to read.

Demo 15: Rank() and Grouping

```
select dept_id
, trunc(avg(salary)) as "AvgSalary"
, rank() Over (order by avg(salary) desc ) as col_Order
from adv_emp
group by dept_id
order by dept_id
;

DEPT_ID  AvgSalary  COL_ORDER
-----  -----
10        26111      1
15        25000      2
20        20250      4
30        20888      3
```

We do not have to show the avg salary in order to use it in the rank function. Maybe we want to keep the salary amounts secret.

Demo 16: Rank() and Grouping

```
select dept_id
, rank() Over (order by avg(salary) desc ) as col_Order
from adv_emp
group by dept_id
;

DEPT_ID  COL_ORDER
-----  -----
10        1
15        2
30        3
20        4
```

We can use more than one of the ranking functions in a single query.

Demo 17: Using two Rank() expressions

```
Select dept_id
, Min(salary) as MinSalary
, rank() Over (order by min(salary) ) as MinSalaryRank
, max(salary) as MaxSalary
, rank() Over (order by max(salary) ) as MaxSalaryRank
from adv_emp
group by dept_id
order by dept_id
;
```

DEPT_ID	MINSALARY	MINSALARYRANK	MAXSALARY	MAXSALARYRANK
10	15000	3	30000	3
15	25000	4	25000	1
20	12000	2	28000	2
30	11000	1	30000	3

6. Partition By

The Partition by clause with a ranking function creates groups and restarts the ranking numbers for each group. Here we rank dept 10 first and then dept 15 and then dept 20 and then dept 30.

Demo 18: Rank() with a Partition

```
Select dept_id, emp_id, year_hired, salary
, rank () Over (partition by Dept_id order by salary ) as col_Order
from adv_emp;
```

DEPT_ID	EMP_I	YEAR_HIRED	SALARY	COL_ORDER
10	301	2010	15000	1
10	320	2012	25000	2
10	310	2012	25000	2
10	322	2012	25000	2
10	305	2012	27000	5
10	311	2012	28000	6
10	319	2014	30000	7
10	321	2014	30000	7
10	309	2014	30000	7
15	323	2012	25000	1
20	315	2013	12000	1
20	302	2010	14000	2
20	303	2014	27000	3
20	312	2010	28000	4
30	313	2012	11000	1
30	318	2014	11500	2
30	307	2012	13500	3
30	308	2013	15000	4
30	317	2014	25000	5
30	316	2013	26000	6
30	306	2010	28000	7
30	304	2010	28000	7
30	314	2013	30000	9

23 rows selected.

You can have more than one partition attribute. The following restarts the numbering for each department and for each year within the department.

Demo 19: Partition by two attributes

```
select dept_id, year_hired, emp_id, salary
, rank()Over(partition by Dept_id, year_hired order by salary ) as col_Order
from adv_emp;
```

DEPT_ID	YEAR_HIRED	EMP_I	SALARY	COL_ORDER
10	2010	301	15000	1
10	2012	320	25000	1
10	2012	322	25000	1
10	2012	310	25000	1
10	2012	305	27000	4
10	2012	311	28000	5
10	2014	319	30000	1

10	2014	309	30000	1
10	2014	321	30000	1
15	2012	323	25000	1
20	2010	302	14000	1
20	2010	312	28000	2
20	2013	315	12000	1
20	2014	303	27000	1
30	2010	304	28000	1
30	2010	306	28000	1
30	2012	313	11000	1
30	2012	307	13500	2
30	2013	308	15000	1
30	2013	316	26000	2
30	2013	314	30000	3
30	2014	318	11500	1
30	2014	317	25000	2

23 rows selected.

Table of Contents

1.	Aggregate () over ().....	1
1.1.	Partition	2
1.2.	ListAgg	4
2.	Running Totals	5
3.	Windowing clause.....	6
3.1.	The Range phrase	7
3.2.	Moving Windowing Clause	8
3.3.	Logical Windowing.....	11

1. Aggregate () over ()

Suppose you want to display each person's salary and how much their salary is over the average salary for all employees.

We'll start by looking just at department 20. We have four employees. The sum of their salaries is 81000 ; the average salary (rounded to an integer) is 20250 . We could do this with a subquery in the Select.

```
variable dpt number
exec :dpt := 20;
```

Demo 01: Using a subquery and the avg function

```
select emp_id, salary
, Round(salary -
        (select Avg(salary)
         from adv_emp
         where dept_id = :dpt),2) as Over_under_avg
from adv_emp
where dept_id = :dpt
;

EMP_ID      SALARY OVER_UNDER_AVG
-----  -----
302          14000      -6250
303          27000      6750
312          28000      7750
315          12000      -8250
```

Demo 02: You could also use a CTE and a Cross join. Be sure you understand why a cross join will work here

```
with avgSal as (
    select Avg(salary * 1.0) as AvgDept100
    from adv_emp
    where dept_id = :dpt
)
select emp_id, salary
, Round(salary - AvgDept100,2) as Over_under_avg
from adv_emp
cross join avgSal
where dept_id = :dpt;
```

Demo 03: Using avg() Over()

We can also do this with an Avg() Over function to get the same result. This is a much simpler syntax than the subquery. If we just do the *avg(salary) over()* for the second column, you can see that the average is calculated over dept_id 20 for each row.

So we do the subtraction to get the offset of each employee's salary compared to the average.

```
select emp_id, salary
, Round(salary - ( Avg(salary) Over() ) ,2 ) as Over_under_avg
from adv_emp
where dept_id = :dpt;
```

In this case the Over() clause has no argument; this is referred to as a Null Over clause and it means that the function applies to the entire dataset- since the function is calculated after the Where clause, this means to the rows for dept_id 20 only. In the subquery version we had to do the filter in both the subquery and the parent query to get the correct result.

1.1. Partition

We might want to look at all the employees and check their over_under_avg based on their dept_id only. This means we want to group the employees by dept_id and calculate the average for each group separately. That is a partition. This query will give that result. I have sorted by Dept ID and salary to make the output easier to read.

Demo 04: Using avg() Over() with a partition

```
select dept_id, salary
, Round(salary - ( Avg(salary) Over( Partition by dept_id) ), 0 )
      as Over_under_avg
, emp_id
from adv_emp
order by dept_id, emp_id
;
```

DEPT_ID	SALARY	OVER_UNDER_AVG	EMP_ID
10	15000	-11111	301
10	27000	889	305
10	30000	3889	309
10	25000	-1111	310
10	28000	1889	311
10	30000	3889	319
10	25000	-1111	320
10	30000	3889	321
10	25000	-1111	322
15	25000	0	323
20	14000	-6250	302
20	27000	6750	303
20	28000	7750	312
20	12000	-8250	315
30	28000	7111	304
30	28000	7111	306
30	13500	-7389	307
30	15000	-5889	308
30	11000	-9889	313
30	30000	9111	314
30	26000	5111	316
30	25000	4111	317
30	11500	-9389	318

23 rows selected.

Demo 05: Who earned more than the average salary for their department?

```
with Aggs as (
    select emp_id, dept_id, salary
    , Avg(salary) Over( Partition by dept_id) as dept_avg
    from adv_emp
)
select emp_id, dept_id, salary
from aggs
where salary > dept_avg
order by dept_id, emp_id;
```

EMP_ID	DEPT_ID	SALARY
305	10	27000
309	10	30000
311	10	28000
319	10	30000
321	10	30000
303	20	27000
312	20	28000
304	30	28000
306	30	28000
314	30	30000
316	30	26000
317	30	25000

12 rows selected.

Demo 06: Calculating Percent of total: Now we want to know what each employee's salary is as a percent of the total salary for that department.

```
select dept_id, emp_id, salary
, round(salary / (sum(salary) over (partition by dept_id)) * 100, 2)
as percent_dept_salary
from adv_emp
order by dept_id, salary;
```

DEPT_ID	EMP_ID	SALARY	PERCENT_DEPT_SALARY
10	301	15000	6.38
10	320	25000	10.64
10	310	25000	10.64
10	322	25000	10.64
10	305	27000	11.49
10	311	28000	11.91
10	319	30000	12.77
10	321	30000	12.77
10	309	30000	12.77
15	323	25000	100
20	315	12000	14.81
20	302	14000	17.28
20	303	27000	33.33
20	312	28000	34.57
30	313	11000	5.85
30	318	11500	6.12
30	307	13500	7.18
30	308	15000	7.98
30	317	25000	13.3
30	316	26000	13.83
30	306	28000	14.89
30	304	28000	14.89
30	314	30000	15.96

23 rows selected.

Start by looking at the results for dept 15. There is one employee, with a salary of 25000. This row reports as 100% of the department salary total. Then look at the results for dept 20. There are four employees. The total salary for dept 210 is 28000. Employee 315 has a salary of 12000 which is about 15% of the department total salary.

You could get the same result with the **ratio_to_report** function:

```
ratio_to_report ( salary ) over ( partition by dept_id )
```

Demo 07: Using Ratio_to_Report

```
select dept_id, emp_id, salary
, round(ratio_to_report ( salary ) over ( partition by dept_id ) * 100,2)
  as percent_dept_salary
from adv_emp
order by dept_id, salary;
```

1.2. ListAgg

This is another aggregate function you can use with the analytical techniques. ListAgg returns a concatenated set of values.

Demo 08: Simple aggregate of the names with a semicolon followed by a space as the delimiter, We get a single row returned with the employees last names.

```
select listagg(name_last, '; ')
      within group (order by name_last) as "Employee List"
from adv_emp;
Employee List
-----
Battaglia; Beiderbecke; Brubeck; Cohen; Coltrane; Davis; Ellington; Evans; Green; Hancock;
Jarrett; Mobley; Monk; Montgomery; Quebec; Redman; Rollins; Shorter; Tatum; Turrentine;
Wabich; Wabich; Wasliewski
```

Demo 09: We can order the values by salary in descending order

```
select listagg(name_last, '; ')
      within group (order by salary desc) as "Employee List"
from adv_emp;
Employee List
-----
Beiderbecke; Redman; Rollins; Turrentine; Brubeck; Cohen; Ellington; Mobley; Coltrane;
Quebec; Monk; Jarrett; Montgomery; Wabich; Wabich; Wasliewski; Evans; Green; Hancock;
Tatum; Battaglia; Shorter; Davis
```

Demo 10: We can add a regular Group by clause and listagg returns aggregates for each group

```
select dept_id as "Dept",
listagg(name_last, '; ')
      within group (order by year_hired) as "Employees by Year Hired"
from adv_emp
group by dept_id;
Dept Employees by Year Hired
-----
10 Green; Brubeck; Coltrane; Jarrett; Wabich; Wabich; Beiderbecke; Redman; Rollins
15 Montgomery
20 Ellington; Hancock; Battaglia; Quebec
```

```
30 Cohen; Mobley; Davis; Tatum; Evans; Monk; Turrentine; Shorter; Wasliewski 10 King
4 rows selected.
```

You can also put the dept_id into a partition by clause in the function. Also try this without the Distinct keyword.

```
Select Distinct dept_id As "Dept"
, listagg(name_last, ',' )
  WITHIN GROUP
    (ORDER BY year_hired)
  OVER (PARTITION BY dept_id) as  "Employees by Year Hired"
from adv.emp;
Dept Employees by Year Hired
-----
15 Montgomery
30 Cohen; Mobley; Davis; Tatum; Turrentine; Evans; Monk; Wasliewski; Shorter
20 Hancock; Ellington; Battaglia; Quebec
10 Green; Coltrane; Brubeck; Jarrett; Wabich; Redman; Rollins; Beiderbecke
4 rows selected.
```

2. Running Totals

We can calculate the total of the salaries and get one value for the rows in the table.

Demo 11: Simple aggregate functions over the table

```
select sum(salary), count(*)
  from adv.emp;
SUM(SALARY)      COUNT(*)
-----
529000           23
```

Suppose we want a running total instead; we want the first row to show the total of the first salary; the second row to show the total of row 1 to row 2; the third row to show the total of row 1 to row 3, etc.

The following query does that; it sums the salary over a range consisting of all of the rows from the start (**unbounded preceding**) to the **current** row. We want the rows in the running total to be in emp_id order.

Demo 12: Running total over the table using Range Between

```
select emp_id, salary
, sum(salary) over(
  order by emp_id
  range between unbounded preceding and current row)
  as run_tot
from adv.emp
order by emp_id
;
EMP_ID SALARY      RUN_TOT
-----
301     15000       15000
302     14000       29000
303     27000       56000
304     28000       84000
305     27000       111000
```

306	28000	139000
307	13500	152500
308	15000	167500
309	30000	197500
310	25000	222500
311	28000	250500
312	28000	278500
313	11000	289500
314	30000	319500
315	12000	331500
316	26000	357500
317	25000	382500
318	11500	394000
319	30000	424000
320	25000	449000
321	30000	479000
322	25000	504000
323	25000	529000

23 rows selected.

Be certain to have the two Order By clauses to sort on the same columns for the output to be understandable.

3. Windowing clause

By adding **Partition BY department_id**, we have told SQL to do a running total for each department, restarting the total when the department changes. We are ordering by dept_id, salary.

This is an example of a windowing frame- a set of rows creating a windows partition. The windowing frame is a moving frame. In this example the windowing frame consists of all rows from the last dept_id change (the partition by argument) up to and including the current row.

Demo 13: Running total by department, adding a partition clause.

```
select dept_id
,   emp_id
,   salary
,   sum(salary) over(
    PARTITION BY DEPT_ID
    order by dept_id, salary
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    as run_tot
from adv_emp
order by dept_id, salary
;
```

I artificially added blank lines for each dept id break

DEPT_ID	EMP_ID	SALARY	RUN_TOT
10	301	15000	15000
10	320	25000	90000
10	310	25000	90000
10	322	25000	90000
10	305	27000	117000
10	311	28000	145000
10	319	30000	235000
10	321	30000	235000
10	309	30000	235000
15	323	25000	25000

20	315	12000	12000
20	302	14000	26000
20	303	27000	53000
20	312	28000	81000
30	313	11000	11000
30	318	11500	22500
30	307	13500	36000
30	308	15000	51000
30	317	25000	76000
30	316	26000	102000
30	306	28000	158000
30	304	28000	158000
30	314	30000	188000

3.1. The Range phrase

The Range phrase is called the Windowing-clause and it defaults to range between unbounded preceding and current row.

Demo 14: This query uses the default and gives us the same output as above.

```
select dept_id, emp_id, salary
, sum(salary) over(
    PARTITION BY DEPT_ID
    order by dept_id, salary
)
as run_tot
from adv_emp
order by dept_id, salary;
```

Demo 15: Using the default range between unbounded preceding and current row and ordering by employee id

```
select emp_id, salary
, sum(salary) over(
    order by emp_id
    -- range between unbounded preceding and current row
)
as run_tot
from adv_emp
order by emp_id;
```

EMP_ID	SALARY	RUN_TOT
301	15000	15000
302	14000	29000
303	27000	56000
304	28000	84000
305	27000	111000
306	28000	139000
307	13500	152500
308	15000	167500
309	30000	197500
310	25000	222500
311	28000	250500
312	28000	278500
313	11000	289500
314	30000	319500
315	12000	331500
316	26000	357500

317	25000	382500
318	11500	394000
319	30000	424000
320	25000	449000
321	30000	479000
322	25000	504000
323	25000	529000

3.2. Moving Windowing Clause

The windowing-clause allows you to set a moving window of rows over which the function should be applied.

Suppose we want the current row and two rows before and after the current row. In the picture below the current row is highlighted in yellow and the window includes the blue and yellow highlighted rows

22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000
33	300	2008	80000

This shows the moving window frame. Note that at the start or the end of the dataset, the window will include less than 5 rows.

22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000

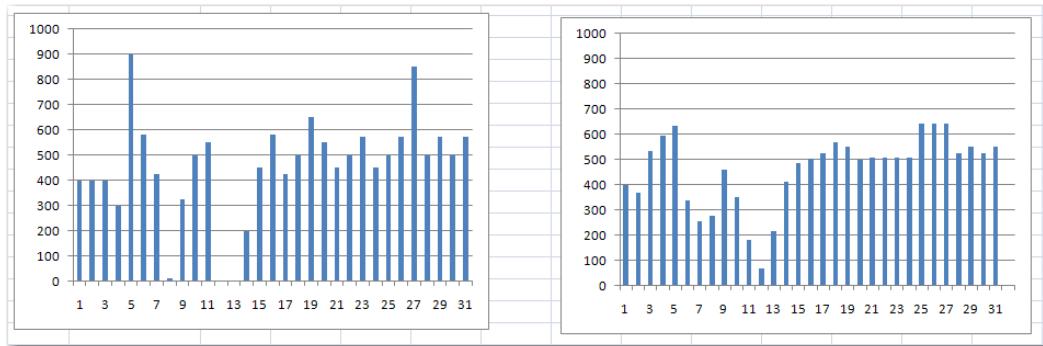
22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000

22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000

We can use several ways to express the range that compose a window. We can use a physical number of rows to be included or a term such as unbounded preceding. The windowing clause is often used when we have a series of values and we want to find a moving average. A moving average can be used with a time dependent series of data and we want to smooth out the data by looking at a three day average. That way if we have a few days that are somewhat out of the normal range they will be blended into the moving average. An example will help with this.

Suppose you were tracking sales over a period of several days. The sales values are apt to fluctuate each day. We could produce the graph of the sales shown on the left which shows each day's sales and we see some pretty big changes. We could also graph a three day average as shown on the right which smoothes out those one day changes. We still see variation in data but a single value does not show up as being that significant.

This is not a case of which graph is correct- but rather what do we want to see.



The test table is `adv_sales` with the first column being a number from 1-20 representing the day of the time period and the second column will be the sales for that day. The inserts are provided on the SQL included with these notes(demo 01). We can refer to this as a three-day average if we have a row, in the proper order, for each day from the first to the last - with no gaps or duplicates.

```
Create table adv_sales (
    sales_day number(2) primary key
    , sales number (5) check (sales >= 0)
);
```

The window we will use include the current row and the 2 preceding rows.

Demo 16: Here the window is the current row and the 2 preceding rows. The Column statement is an SQL*Plus command to format the third column; it does not work in SQL Developer.

```
Column Three_day_avg format "9999.99"
```

```
select sales_day
, sales
, Avg(sales) OVER (
    ORDER BY sales_day
    ROWS BETWEEN
    2 PRECEDING AND CURRENT ROW
) AS three_day_avg
from adv_sales
order by sales_day;
```

SALES_DAY	SALES	THREE_DAY_AVG
25-APR-15	400	400.00
26-APR-15	400	400.00
27-APR-15	400	400.00
28-APR-15	300	366.67
29-APR-15	900	533.33
30-APR-15	580	593.33
01-MAY-15	425	635.00
02-MAY-15	10	338.33
03-MAY-15	325	253.33
04-MAY-15	500	278.33
05-MAY-15	550	458.33
06-MAY-15	0	350.00
07-MAY-15	0	183.33
08-MAY-15	200	66.67
09-MAY-15	450	216.67
10-MAY-15	580	410.00
11-MAY-15	425	485.00
12-MAY-15	475	493.33
13-MAY-15	375	425.00
14-MAY-15	500	450.00
15-MAY-15	650	508.33

16-MAY-15	550	566.67
17-MAY-15	450	550.00
18-MAY-15	500	500.00
19-MAY-15	575	508.33
20-MAY-15	450	508.33
21-MAY-15	500	508.33
22-MAY-15	575	508.33
23-MAY-15	850	641.67
24-MAY-15	500	641.67
25-MAY-15	575	641.67
26-MAY-15	500	525.00
27-MAY-15	575	550.00
28-MAY-15	500	525.00
29-MAY-15	575	550.00
30-MAY-15	575	550.00
31-MAY-15	575	575.00
01-JUN-15	425	525.00
02-JUN-15	500	500.00
03-JUN-15	455	460.00
04-JUN-15	0	318.33
05-JUN-15	0	151.67
06-JUN-15	0	0.00
07-JUN-15	0	0.00
08-JUN-15	900	300.00
09-JUN-15	450	450.00
10-JUN-15	780	710.00
11-JUN-15	475	568.33
12-JUN-15	875	710.00
13-JUN-15	375	575.00
14-JUN-15	800	683.33

We can see that the last column has less variation since each value (except for the end points depending on row preceding or rows following) represents the average of three data points. The low number and the high number do affect the three_Day_avg, but their effect is not as much as in the sales column.

The first two rows are not three row averages, since we do not yet have three rows of sale. What Oracle does is treat the missing rows as Nulls. We would need to consider if this is relevant to the purpose for which we are running the query.

It may be that our company sells more merchandise at the start or the end of the month. In that case, the three-day-average could be misleading. If we were storing data about weather and the data stored the high temperature for each day, we might not think that that is influenced by the start or end of the month. Queries like this are generally done with large amounts of data and often the end points can be ignored or removed.

However you always need to check these assumptions. Maybe some factory in the area does larger runs on the last two days of the month and this could influence the local temperature due to atmospheric conditions. Assumptions in statistics can hide the data you are trying to find.

Demo 17: Here the window extends over four rows, the current, two days preceding and one day following. It also limits the display to some of the rows in the table.

```
Column Four_day_avg format "9999.99"

select sales_day
, sales
, round(avg(sales) over (
    order by sales_day
    rows between 2 preceding and 1 following ), 2) as four_day_avg
from adv_sales
```

```
where extract (month from sales_day) = 5
and   extract (year from sales_day) = 2015
order by sales day;
```

SALES_DAY	SALES	FOUR_DAY_AVG
01-MAY-15	425	217.50
02-MAY-15	10	253.33
03-MAY-15	325	315.00
04-MAY-15	500	346.25
05-MAY-15	550	343.75
06-MAY-15	0	262.50
07-MAY-15	0	187.50
08-MAY-15	200	162.50
09-MAY-15	450	307.50
10-MAY-15	580	413.75
11-MAY-15	425	482.50
12-MAY-15	475	463.75
13-MAY-15	375	443.75
14-MAY-15	500	500.00
15-MAY-15	650	518.75
16-MAY-15	550	537.50
17-MAY-15	450	537.50
18-MAY-15	500	518.75
19-MAY-15	575	493.75
20-MAY-15	450	506.25
21-MAY-15	500	525.00
22-MAY-15	575	593.75
23-MAY-15	850	606.25
24-MAY-15	500	625.00
25-MAY-15	575	606.25
26-MAY-15	500	537.50
27-MAY-15	575	537.50
28-MAY-15	500	537.50
29-MAY-15	575	556.25
30-MAY-15	575	556.25
31-MAY-15	575	575.00

31 rows selected.

3.3. Logical Windowing

The previous windowing clauses have used a certain number of rows. We can also have Oracle calculate which rows fit into our window groups. This is often done with date values where we might want the average of the previous week's sales. In that case the range would be

| Range between Interval '7' day preceding and current row

This syntax requires a datetime field (year, month, day, hour, minute, second)

Demo 18: We want a total of the quantity ordered for this month and the previous month. This uses the oe tables

```
with CTE as (
  select order_date, sum(quantity_ordered) As AMount
  from oe_orderheaders
  join oe_orderdetails using (order_id)
  where extract(year from order_date)= 2016
  group by order_date
)
select order_date, amount
, sum(amount) over (
  order by order_date range between
```

```

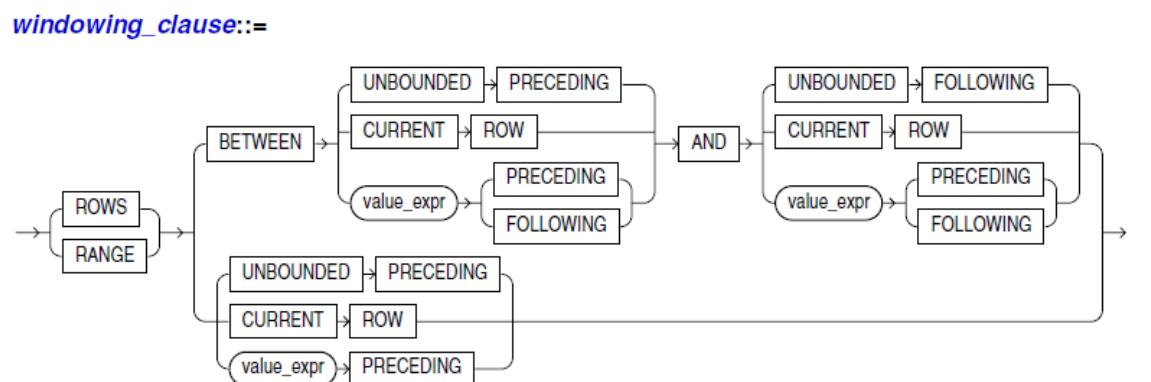
        interval '1' month preceding and current row
    ) as MonthSum
  from CTE
  ;
  ORDER_DATE      AMOUNT  MONTHSUM
  -----  -----
02-JAN-16          10      10
03-JAN-16          21      31
04-JAN-16           1      32
05-JAN-16          29      61
07-JAN-16           1      62
11-JAN-16           1      63
12-JAN-16           4      67
15-JAN-16          10      77
23-JAN-16           5      82
26-JAN-16           3      85
27-JAN-16           7      92
31-JAN-16          24     116
01-FEB-16          16     132
02-FEB-16           2     134
03-FEB-16           6     130
12-FEB-16           7      84
01-MAR-16          53      84
05-MAR-16          54     114
07-MAR-16          20     134
08-MAR-16          14     148
09-MAR-16           5     153
04-APR-16          12     105
05-APR-16           4     109
06-APR-16           1      56
07-APR-16           3      59
08-APR-16           4      43
01-MAY-16          86     110
09-MAY-16           4      90
12-MAY-16          15     105

 29 rows selected

```

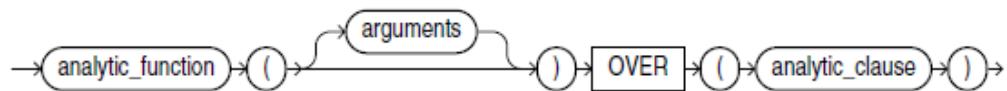
Again the first few rows of data may not be as valuable, since we do not have a full month of prior data. This is more practical and meaningful if we have a lot of data.

The windowing clause can be quite flexible; this is the model for that clause from the Oracle documentation. Windows can be based on a number of rows or a logical interval (often based on time).



The analytic functions can occur only in the Select list or the Order By clause. Other parts of the query (join, Where, Group by, Having) are carried out before the analytic functions.

analytic_function ::=



analytic_clause ::=



Table of Contents

1.	RollUp	1
1.1.	Two level group	2
2.	Cube.....	4
3.	Grouping.....	5
4.	Grouping sets.....	8

1. RollUp

The RollUp function lets you calculate multiple levels of subtotals and a grand total. The Rollup function is added to the Group By clause. You give the Rollup function one or more grouping columns for the subtotals.

Demo 01: Simple group by with a single attribute; this is not using rollup

```
select dept_id, sum(salary)
from adv_emp
group by dept_id
order by dept_id;
```

DEPT_ID	SUM(SALARY)
10	235000
15	25000
20	81000
30	188000

Demo 02: The rollup function gives you department totals and a grand total (529000) with a Null for the dept id since that row does not belong to a department.

```
select dept_id, sum(salary)
from adv_emp
group by rollup( dept_id)
order by dept_id;
```

DEPT_ID	SUM(SALARY)
10	235000
15	25000
20	81000
30	188000
	529000

Demo 03: Replacing the null with a label

```
select
    coalesce(to_char(dept_id, '99999'), 'Total') as Dept
    , sum(salary) as SalaryTotal
from adv_emp
group by rollup( dept_id)
order by dept_id;
```

DEPT	SALARYTOTAL
10	235000
15	25000
20	81000
30	188000
Total	529000

1.1. Two level group

With a two level group we get subtotals by year_hired within each department and the department total and a grand total.

Demo 04: Simple two level group

```
select dept_id, year_hired, sum(salary)
from adv_emp
group by rollup( dept_id, year_hired)
;
```

DEPT_ID	YEAR_HIRED	SUM(SALARY)
10	2010	15000
10	2012	130000
10	2014	90000
10		235000
15	2012	25000
15		25000
20	2010	42000
20	2013	12000
20	2014	27000
20		81000
30	2010	56000
30	2012	24500
30	2013	71000
30	2014	36500
30		188000
		529000

Demo 05: We can group by year_hired and department within the year_hired

```
select year_hired, dept_id, sum(salary)
from adv_emp
group by rollup( year_hired, dept_id )
;
```

YEAR_HIRED	DEPT_ID	SUM(SALARY)
2010	10	15000
2010	20	42000
2010	30	56000
2010		113000
2012	10	130000
2012	15	25000
2012	30	24500
2012		179500
2013	20	12000
2013	30	71000
2013		83000
2014	10	90000
2014	20	27000
2014	30	36500
2014		153500
		529000

Demo 06: roll up on one grouping only; no grand total since we are not rolling up by department

```
select dept_id, year_hired, sum(salary), Count(*)
from adv_emp
group by ( dept_id ), rollup( year_hired );
```

DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT(*)
10	2010	15000	1
10	2012	130000	5
10	2014	90000	3
		235000	9
15	2012	25000	1
15		25000	1
20	2010	42000	2
20	2013	12000	1
20	2014	27000	1
20		81000	4
30	2010	56000	2
30	2012	24500	2
30	2013	71000	3
30	2014	36500	2
		188000	9

Demo 07: With additional formatting for labeling the result set

```
select
  case when dept_id is null then ' Grand total'
        else to_char(dept_id, '99999')
        end as "Department"
  ,
  case when year_hired is null and dept_id is null then ''
        when year_hired is null then ' Dept total'
        else to_char(year_hired, '9999')
        end as "YearHired"
  , sum(salary) as "SalaryTotal"
  , count (*) as "EmpCount"
from adv_emp
group by Rollup( dept id, year hired );
```

Department	YearHired	SalaryTotal	EmpCount
10	2010	15000	1
10	2012	130000	5
10	2014	90000	3
10	Dept total	235000	9
15	2012	25000	1
15	Dept total	25000	1
20	2010	42000	2
20	2013	12000	1
20	2014	27000	1
20	Dept total	81000	4
30	2010	56000	2
30	2012	24500	2
30	2013	71000	3
30	2014	36500	2
30	Dept total	188000	9
Grand total		529000	23

2. Cube

Demo 08: This is one of the Rollups we did earlier

```
select dept_id, year_hired, sum(salary), count (*)
from adv_emp
group by Rollup( dept_id, year_hired )
;
```

DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT (*)
10	2010	15000	1
10	2012	130000	5
10	2014	90000	3
		235000	9
15	2012	25000	1
15		25000	1
20	2010	42000	2
20	2013	12000	1
20	2014	27000	1
20		81000	4
30	2010	56000	2
30	2012	24500	2
30	2013	71000	3
30	2014	36500	2
30		188000	9
		529000	23

Demo 09: Then change the Rollup to Cube and we get more rows of data. In addition to the Grand total and the total by department, we also get a total by year_hired. The Cube function aggregates for all possible combinations of the specified columns. With two columns we have four possible combinations: dept_id, year_hired, both dept_id and year_hired, all- the grand total.

First we get the GrandTotal, then the Year totals, and department totals and Dept + year totals

```
select dept_id
, year_hired
, sum(salary)
, count (*)
from adv_emp
group by CUBE( dept_id , year_hired )
;
```

DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT (*)
		529000	23
	2010	113000	5
	2012	179500	8
	2013	83000	4
	2014	153500	6
10		235000	9
10	2010	15000	1
10	2012	130000	5
10	2014	90000	3
15		25000	1
15	2012	25000	1
20		81000	4
20	2010	42000	2

20	2013	12000	1
20	2014	27000	1
30		188000	9
30	2010	56000	2
30	2012	24500	2
30	2013	71000	3
30	2014	36500	2

3. Grouping

One of the features of the use of RollUp and Cube is that we do not get labels that indicate which rows are totals- we simply get blanks. To remedy this, we use the GROUPING function with either Decode or Case to display labels. The Grouping function takes a single argument and returns either 0 (if this is not a totaling row) or 1 (if this is a totaling row).

The next query shows how the Grouping function works. You would not normally display the value of Grouping.

The first two return rows are grouping row for both dept and year hired- the two grouping functions return a 1.

The next row is a grouping row for dept but not for year hired. The grp_dept column has a 0 and the grp_year column has a 1

The last row is not a grouping row for either dept or year hired- both grouping functions return 1.

Demo 10: The GROUPING function returns a 0 or a 1

```
select
    grouping(dept_id) as grp_dept
    , grouping(year_hired) as grp_year
    , dept_id, year_hired
    , sum(salary)
    , count(*)
from adv_emp
group by CUBE( dept_id , year_hired )
order by dept_id, year_hired;
```

GRP_DEPT	GRP_YEAR	DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT(*)
0	0	10	2010	15000	1
0	0	10	2012	130000	5
0	0	10	2014	90000	3
0	1	10		235000	9
0	0	15	2012	25000	1
0	1	15		25000	1
0	0	20	2010	42000	2
0	0	20	2013	12000	1
0	0	20	2014	27000	1
0	1	20		81000	4
0	0	30	2010	56000	2
0	0	30	2012	24500	2
0	0	30	2013	71000	3
0	0	30	2014	36500	2
0	1	30		188000	9
1	0		2010	113000	5
1	0		2012	179500	8
1	0		2013	83000	4
1	0		2014	153500	6
1	1			529000	23

20 rows selected.

This is the order we want for the rows and the labels to be displayed; we want the labels in the first column, replacing the two Grouping columns. This is sample data only.

GRP_DEPT	GRP_YEAR	DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT (*)
0	0	100	2000	99999	9
0	0	100	2007	9999	9
0	1	100		91999	99
					Dept total
0	0	200	2008	99999	9
0	1	200		99999	99
					Dept total
0	0	300	2007	999999	99
0	1	300		999999	99
					Dept total
1	0		2000	99999	99
1	0		2007	999999	99
1	0		2008	99999	99
					Year total
1	1			999999	999
					Grand total

11 rows selected

The pattern is that if both grouping are 1 then this is the **grand total line**; if the dept grouping is 0 and the year grouping is 1 then this is a **Dept total line**; if the dept grouping is 1 and the year grouping is 0 then this is a **Year total line**; if they are both 0, then we do not have a label. We can use a case statement for this. I added an Rpad to control the width of the first column.

Demo 11:

```
Column Descr format A20
select
    Rpad( CASE
        when GROUPING(dept_ID) = 1 and Grouping (year_hired) = 1
            then 'Grand Total'
        when GROUPING(dept_ID) = 1 and Grouping (year_hired) = 0
            then ' Year Total'
        when GROUPING(dept_ID) = 0 and Grouping (year_hired) = 1
            then '    Dept Total'
        when GROUPING(dept_ID) = 0 and Grouping (year_hired) = 0
            then ''
        end, 20) AS Descr
, dept_id, year_hired
, sum(salary), count ('emp')
from adv_emp
group by Cube( dept_id , year_hired )
order by dept_id , year_hired
;
```

DESCR	DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT ('EMP')
	10	2010	15000	1
	10	2012	130000	5
	10	2014	90000	3

Dept Total	10	235000	9
	15	25000	1
Dept Total	15	25000	1
	20	42000	2
	20	12000	1
	20	27000	1
Dept Total	20	81000	4
	30	56000	2
	30	24500	2
	30	71000	3
	30	36500	2
Dept Total	30	188000	9
Year Total		113000	5
Year Total		179500	8
Year Total		83000	4
Year Total		153500	6
Grand Total		529000	23

Demo 12: An alternative

Column Department format A20

Column year_hired format A20

```

select
    lpad(case when Grouping(dept_id) = 1 then '--- Year Total ---'
            else To_Char(dept_id) end, 20) as Department
    , lpad(case when Grouping(year_hired) = 1 then '--- Dept Total---'
            else To_Char(year_hired) end, 20) as Year_Hired
    , sum(salary)
    , count ('emp')
from adv_emp
group by cube( dept_id , year_hired )
order by dept_id , year_hired
;

```

DEPARTMENT	YEAR_HIRED	SUM(SALARY)	COUNT ('EMP')
10	2010	15000	1
10	2012	130000	5
10	2014	90000	3
10	--- Dept Total---	235000	9
15	2012	25000	1
15	--- Dept Total---	25000	1
20	2010	42000	2
20	2013	12000	1
20	2014	27000	1
20	--- Dept Total---	81000	4
30	2010	56000	2
30	2012	24500	2
30	2013	71000	3
30	2014	36500	2
30	--- Dept Total---	188000	9
--- Year Total ---	2010	113000	5
--- Year Total ---	2012	179500	8
--- Year Total ---	2013	83000	4
--- Year Total ---	2014	153500	6
--- Year Total ---	--- Dept Total---	529000	23

4. Grouping sets

The Grouping Sets expression lets you specify which groupings you want for the aggregates.

Demo 13: Here I have a group by department, a group by year_hired, a group by both attributes and a group by all – indicated by ()

Column year_hired format 9999

```
Select dept_id, year_hired, sum(salary), count ('emp')
from adv_emp
group by grouping sets (
    dept_id
    , year_hired
    , (dept_id, year_hired)
    , ()
);
```

DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT ('EMP')
10	2010	15000	1
20	2010	42000	2
30	2010	56000	2
10	2012	130000	5
15	2012	25000	1
30	2012	24500	2
20	2013	12000	1
30	2013	71000	3
10	2014	90000	3
20	2014	27000	1
30	2014	36500	2
	2010	113000	5
	2012	179500	8
	2013	83000	4
	2014	153500	6
10		235000	9
15		25000	1
20		81000	4
30		188000	9
		529000	23

Demo 14:

```
select dept_id
, year_hired
, sum(salary)
, count ('emp')
from adv_emp
group by grouping sets (dept_id, year_hired, ())
order by coalesce(dept_id, 999999), coalesce(year_hired, 9999)
;
```

DEPT_ID	YEAR_HIRED	SUM(SALARY)	COUNT ('EMP')
10		235000	9
15		25000	1
20		81000	4
30		188000	9

2010	113000	5
2012	179500	8
2013	83000	4
2014	153500	6
	529000	23

Table of Contents

1.	Using Cube to generate a rows table	1
2.	Setting up a table with a set number of rows	1
3.	Generating numbers.....	3
4.	Building a calendar	5
5.	Using a calendar	7

1. Using Cube to generate a rows table

There are times when it is helpful to have a table which has a certain number of rows. It seems simple but there are times that this comes in handy.

You could just define the table and then do a lot of inserts- but that seems boring particularly if you need a lot of rows. This discusses one way to generate the rows for that table. It is not the most efficient but it does give us some more work with analytical techniques.

We will start with a simple demo to show you how you might use a number table. Assume we have a table MyNbrs which contains the numbers from 1 to 50. The code is in the demo. It hard codes the numbers.

Now we want a set of dates for the month of November 2012. We can start with the last day of October and add 1 then 2, then 3 etc to build dates. We want to stop with 30 dates

Demo 01: Using a numbers table to build a calendar. We could insert these values into a table or just use them in a result set.

```
select date '2012-10-31' + col
from myNbrs
where col <= 30;
```

DATE	'2012-10-31'

01-NOV-12	
02-NOV-12	
03-NOV-12	
04-NOV-12	
05-NOV-12	
06-NOV-12	
. . . rows skipped	
28-NOV-12	
29-NOV-12	
30-NOV-12	
30 rows selected.	

But setting up a table of 365 values or 10,000 values would be tedious if we coded each insert. We want to build that table. We don't have to worry too much about efficiency since we could just create the table once and then use it whenever we need a series of numbers.

2. Setting up a table with a set number of rows

Demo 02: Create the table and insert the first value- 1

```
create table starter ( col integer);
insert into starter values (1);
```

Demo 03: If we add a group by cube we get two rows- they both have the value 1, but we have two numbers

```
select 1 as Num1  
from starter  
group by cube(col);
```

Num1

1
1
2 rows selected

Demo 04: Now we group by cube(col,col) and we get 4 rows.

```
select 1 as Num1  
from starter  
group by cube(col, col);
```

Num1

1
1
1
1
4 rows selected

Demo 05: With a three way group we get 8 rows; with 4 we would get 16, with 5 we would get 32- binary magic!

```
select 1 as Num1  
from starter  
group by cube(col, col, col);
```

Num1

1
1
1
1
1
1
1
8 rows selected

Demo 06: We always get the value 1; the more important part is how many rows we get. If we use the generating query as a table expression in the from clause, we can just get the number of rows.

```
select 1 as Num1  
from starter  
group by cube(col, col, col, col);  
  
select count(*) as NumberRows  
from (  
    select 1 as Num1  
    from starter  
    group by cube(col, col, col, col)  
)tbl;
```

```
NumberRows
-----
```

```
16
```

```
select 1 as Num1
from starter
group by cube(col, col, col, col, col);
select count(*)as NumberRows
from (
    select 1 as Num1
    from starter
    group by cube(col, col, col, col, col)
)tbl;
```

```
NumberRows
-----
```

```
32
```

I hope you see the pattern; with each new value in the cube function we double the number of rows.

With 8 arguments to cube, you get 256 rows; with 12 you get 4096 which you should recognize as the binary base numbers. This is one of those patterns that you can use without fully understanding it.

We can use that query to do inserts into a table.

Demo 07: With these statements you get 32 rows into the table MyRows. Each row has a column with a value of 1

```
create table MyRows ( col integer);
insert into MyRows
    select 1 as Num1
    from starter
    group by cube(col, col, col, col, col);
select * from myRows ;
```

Do we really need that table starter? It only serves to provide the value to use in the generated result set. What about a subquery?

Demo 08: A result set with 64 rows, all with the value 13

```
select 13 as num
from (
    select 1 AS num2 from dual
) tb12
group by cube(num2, num2, num2, num2, num2, num2);
```

3. Generating numbers

Demo 09: This probably still looks like a trick; we need something useful. How about generating numbers from 101 to 116?

```
select num + ROW_NUMBER()over (order by num) as the_count
from (
    select 100 as num
    from (
        select 1 AS num2 from dual
    ) tb12
    group by cube(num2, num2, num2, num2)
) tb1;
```

```
the_count
-----
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116

16 rows selected
```

The following query takes a while to run, but we can get a set of numbers in a certain range. Adjust the group by cube to generate enough numbers and then use a filter to set the upper range.

Demo 10: This query generates the numbers from 101 to 3000; this is 2900 rows and took 15 seconds on my system. The CTE generates 4096 rows based on the Group by cube. That is the part of the query that takes time.

```
with numbers as (
  select num + ROW_NUMBER() over (order by num) as the_count
  from
    (
      select 100 as num
      from
        (
          select 1 AS num2 from dual
          ) tb12
        group by cube(num2, num2, num2, num2, num2, num2, num2, num2, num2,
num2, num2)
        ) tb1
    )
  select the_count
  from numbers
  where the_count <= 3000;
```

This is not a great approach for generating a lot of numbers if you want to use this as part of a query. You could insert the values into a number table so that you take the hit once for generating the numbers. But if you need only a few rows, using this technique is sensible.

4. Building a calendar

We have 18 weeks in a semester. We need to generate at least 18 rows; 16 is not enough so we need 32 rows—that is 5 parameters to group by cube.

Demo 11:

```
with numbers as (
  select num + ROW_NUMBER() over (order by num) as the_count
  from
    (
      select 0 as num
      from
        (
          select 1 AS num2 from dual
        ) tbl2
      group by cube(num2, num2, num2, num2, num2)
    ) tbl
)
select the_count, 'Unit: ' || the_count as Units
from numbers
where the_count <= 18 ;
```

THE_COUNT	UNITS
1	Unit: 1
2	Unit: 2
3	Unit: 3
4	Unit: 4
5	Unit: 5
6	Unit: 6
7	Unit: 7
8	Unit: 8
9	Unit: 9
10	Unit: 10
11	Unit: 11
12	Unit: 12
13	Unit: 13
14	Unit: 14
15	Unit: 15
16	Unit: 16
17	Unit: 17
18	Unit: 18

18 rows selected

Demo 12: What were the weekend days in the spring 2016 semester? (You remember weekends!)

```
Variable startdtm  varchar2(10);
Exec :startdtm  := '2016-01-18';

Variable stopdtm  varchar2(10);
Exec :stopdtm  := '2016-05-24';

With weekends as(
  select  dtm
  from (
```

```
select to_date(:startdtm, 'yyyy-mm-dd') + ROW_NUMBER() over (order by num)
as dtm
from
( select 1 as num
  from (select 1 as n from dual) tbl0
  group by cube(n, n, n, n, n, n, n, n)
) tbl
) tbl2
where dtm <= to_date(:stopdtm, 'yyyy-mm-dd') and to_char(dtm, 'fmday') in
('sunday', 'saturday')
)
select dtm
from weekends ;
```

```
DTM
-----
23-JAN-16
24-JAN-16
30-JAN-16
31-JAN-16
06-FEB-16
07-FEB-16
13-FEB-16
14-FEB-16
20-FEB-16
21-FEB-16
27-FEB-16
28-FEB-16
05-MAR-16
06-MAR-16
12-MAR-16
13-MAR-16
19-MAR-16
20-MAR-16
26-MAR-16
27-MAR-16
02-APR-16
03-APR-16
09-APR-16
10-APR-16
16-APR-16
17-APR-16
23-APR-16
24-APR-16
30-APR-16
01-MAY-16
07-MAY-16
08-MAY-16
14-MAY-16
15-MAY-16
21-MAY-16
22-MAY-16
```

36 rows selected.

Demo 13: What happens with this month? (<http://en.wikipedia.org/wiki/1582>) But read the output first!

```

Variable startdtm  varchar2(10);
Exec :startdtm  := '1582-10-01';

Variable stopdtm  varchar2(10);
Exec :stopdtm  := '1582-10-31';

With c as(
select  dtm
from (
    select to_date(:startdtm, 'yyyy-mm-dd') - 1 + ROW_NUMBER()over (order by
num)  as dtm
from
    ( select 1 as num
      from (select 1 as n from dual) tb10
      group by cube(n, n, n, n, n, n, n, n)
    ) tb1
) tb12
where  dtm <= to_date(:stopdtm, 'yyyy-mm-dd')
)
select to_char(dtm, 'yyyy-mm-dd') as OCT1582
from c ;

```

5. Using a calendar

Suppose we want to get the number of orders for each day in June 2015. We are using the oe_order_headers table for this. If we look at just the order date in the oe_order headers table for June 2015, we get 12 rows. That's means we have a lot of days with no orders.

```

select order_id, order_date
from oe_orderheaders
where to_char(order_date, 'yyyy-mm') = '2015-06';

```

ORD_ID	ORD_DATE
540	02-JUN-15
390	04-JUN-15
395	04-JUN-15
301	04-JUN-15
302	04-JUN-15
303	10-JUN-15
306	04-JUN-15
307	04-JUN-15
312	07-JUN-15
313	07-JUN-15
324	11-JUN-15
378	14-JUN-15

12 rows selected.

Demo 14: This gives us the counts only for days where we have orders.

```

select order_date , count(*) from oe_orderheaders
  where to_char(order_date, 'YYYY-mm') = '2015-06'
group by order_date;

```

ORD_DATE	COUNT(*)
04-JUN-15	6

11-JUN-15	1
10-JUN-15	1
02-JUN-15	1
07-JUN-15	2
14-JUN-15	1
6 rows selected.	

So how do we get a count- 0 for each day with no orders.

Demo 15: We build a June 2015 calendar and left join it to the order headers table. We can use a CTE to build the calendar and another CTE to get the counts for each day with orders and then join the two CTEs.

```

variable startdtm  varchar2(10);
exec :startdtm  := '2015-06-01';

variable stopdtm  varchar2(10);
exec :stopdtm   := '2015-06-30';

With calJune2015 as(
    select dtm
    from (
        select to_date(:startdtm, 'yyyy-mm-dd') - 1 + ROW_NUMBER() over (order by num)
        as dtm
    from(
        select 1 as num
        from (select 1 as n from dual) tbl0
        group by cube(n, n, n, n, n, n, n, n)
        )tbl
    )tbl2
where dtm <= to_date(:stopdtm, 'yyyy-mm-dd')
)
,
ordersJune2015 as (
    select order_date , count(*)  OrderCount
    from oe_orderheaders
    where to_char(order_date, 'YYYY-mm') = '2015-06'
    group by order_date
)
select to_char(dtm, 'yyyy-mm-dd') as JuneDates, OrderCount
from calJune2015
left join ordersJune2015 on calJune2015.dtm = ordersJune2015.order_date
order by JuneDates;

```

JUNEDATES	ORDERCOUNT
2015-06-01	
2015-06-02	1
2015-06-03	
2015-06-04	6
2015-06-05	
2015-06-06	
2015-06-07	2
2015-06-08	
2015-06-09	
2015-06-10	1
2015-06-11	1
2015-06-12	
2015-06-13	
2015-06-14	1
2015-06-15	
2015-06-16	

2015-06-17
2015-06-18
2015-06-19
2015-06-20
2015-06-21
2015-06-22
2015-06-23
2015-06-24
2015-06-25
2015-06-26
2015-06-27
2015-06-28
2015-06-29
2015-06-30

30 rows selected.

Table of Contents

1.	Ntile	1
2.	Islands.....	3
3.	Lead and Lag.....	6
4.	Gaps.....	7
5.	Percent_Rank and Cume_Dist	7

Ranking functions are used to rank rows of data according to some criteria. . We might want to rank employees by salary or we might want to rank employees by salary within each department. Ranking functions have to consider ties. The ranking functions are

- NTile
- Islands and Gaps
- Lead and Lag
- Percent_rank (optional)
- Cume_Dist (optional)

Oracle seldom develops functions that are not useful in business.

1. Ntile

Ntile works by dividing the data into percentile groupings, called buckets. Suppose you want the salaries divided into 2 groups (the top 50% and the bottom 50%). The number of buckets is an argument to NTILE

Demo 01: NTILE() based on salary

```
Select NTile(5) Over (order by salary ) as Bucket
, emp_id, dept_id, year_hired, salary
from adv_emp;
```

We have 23 rows with non null salaries and we get one group with 12 rows and one with 11.

BUCKET	EMP_ID	DEPT_ID	YEAR_HIRED	SALARY
1	313	30	2012	11000
1	318	30	2014	11500
1	315	20	2013	12000
1	307	30	2012	13500
1	302	20	2010	14000
2	301	10	2010	15000
2	308	30	2013	15000
2	322	10	2012	25000
2	323	15	2012	25000
2	320	10	2012	25000
3	317	30	2014	25000
3	310	10	2012	25000
3	316	30	2013	26000
3	305	10	2012	27000
3	303	20	2014	27000
4	306	30	2010	28000
4	312	20	2010	28000
4	311	10	2012	28000
4	304	30	2010	28000
5	314	30	2013	30000
5	321	10	2014	30000
5	319	10	2014	30000
5	309	10	2014	30000

Note the employees who earn 25000 ; there are 5 of these employees- three in bucket 2 and 2 in the next bucket. We could add a second sort key to decide on the group if that make business sense. Perhaps we want people with earlier hire date to be in the lower numbered group. You should not make up a distinction rule that has no

business rule to support it. But if we were giving out bonuses based on the employee's bucket- we would have to deal with this appropriately.

Demo 02: Suppose we want to have a different set of buckets for each department. We can partition by the dept_id. This starts a new bucketing for each department. I also use a variable for the bucket count.

```
variable d number;
exec :d := 2;

select
    NTile (:d) Over (partition by dept_id order by salary, year_hired ) as Bucket
    , dept_id, year_hired, salary, emp_id
from adv_emp
;
```

BUCKET	DEPT_ID	YEAR_HIRED	SALARY	EMP_ID
1	10	2010	15000	301
1	10	2012	25000	320
1	10	2012	25000	310
1	10	2012	25000	322
1	10	2012	27000	305
2	10	2012	28000	311
2	10	2014	30000	319
2	10	2014	30000	321
2	10	2014	30000	309
1	15	2012	25000	323
1	20	2013	12000	315
1	20	2010	14000	302
2	20	2014	27000	303
2	20	2010	28000	312
1	30	2012	11000	313
1	30	2014	11500	318
1	30	2012	13500	307
1	30	2013	15000	308
1	30	2014	25000	317
2	30	2013	26000	316
2	30	2010	28000	306
2	30	2010	28000	304
2	30	2013	30000	314

Demo 03: Perhaps we don't want to make buckets for small departments and we want to improve the output a bit. One CTE puts together the bucket list and the second CTE finds the department with more than 8 employees. These are joined in the main query.

```
variable NumBkts number;
exec :NumBkts := 2;
variable MinDept number;
exec :MinDept := 8;

With
Bucket_list as(
    Select
        NTile (:NumBkts) Over (partition by dept_id order by salary, year_hired )
    as nt
    , emp_id, name_last, dept_id
    from adv_emp
```

```

)
Groups as (
    select dept_id, count(*) as deptCount
    from adv_emp
    group by dept_id
    having count(*) >= :MinDept
)
select ' Group:' || cast(groups.dept_id as varchar(5)) || '-' || cast(nt as
varchar(2)) as StudyGroup
, emp_id, name_last
from Groups
join Bucket_list on Groups.dept_id = Bucket_list.dept_id
;

```

STUDYGROUP	EMP_ID	NAME_LAST
Group:10-1	301	Green
Group:10-1	320	Jarrett
Group:10-1	310	Wabich
Group:10-1	322	Wabich
Group:10-1	305	Coltrane
Group:10-2	311	Brubeck
Group:10-2	319	Redman
Group:10-2	321	Rollins
Group:10-2	309	Beiderbecke
Group:30-1	313	Davis
Group:30-1	318	Shorter
Group:30-1	307	Tatum
Group:30-1	308	Evans
Group:30-1	317	Wasliewski
Group:30-2	316	Monk
Group:30-2	306	Cohen
Group:30-2	304	Mobley
Group:30-2	314	Turrentine

2. Islands

Islands are sequences of values with no gaps. For example; these are the order id values between 400 and 510 from the order headers table. I have color coded each of the islands

ORD_ID
400
401
402
405
407
408
411
412
413
414
415

Demo 04: This is the code to get the first value in an island and the last value and the count.

```
with dataset as (
  select order_id, order_id - dense_rank() over (order by order_id) as grp
  from oe_orderheaders
  where order_id between 400 and 510
)
select MIN(order_id) as start_of_range
, MAX(order_id) as end_of_range
, COUNT(order_id) as number_in_range
from dataset
group by grp
order by 1;
```

START_OF_RANGE	END_OF_RANGE	NUMBER_IN_RANGE
400	402	3
405	405	1
407	408	2
411	415	5

How does that work? Take a look at the CTE; I have added another column for the rank. Note that for each island the difference between the ord_id and the rank is the same value. And for different islands, that column has a different value. So we can group by the difference value to put all of the rows in the island in the same group.

```
select order_id
, dense_rank() over (order by order_id) as DRank
, order_id - dense_rank() over (order by order_id) as Diff
from oe_orderheaders
where order_id between 400 and 510;
```

ORD_ID	DRANK	DIFF
400	1	399
401	2	399
402	3	399
405	4	401
407	5	402
408	6	402
411	7	404
412	8	404
413	9	404
414	10	404
415	11	404

Demo 05: Suppose we want to find data about our customers and how consistently they place orders over months. I am going to use the books table and orders for the year 2014

For the demo I limited this to three selected customers: 224038, 227105, 272787. First, display the order dates for those customers. I have inserted spaces between each customer.

```
select cust_id, order_date
from bk_order_headers
where extract (YEAR from order_date) = 2015
and cust_id in (224038, 227105, 272787)
order by cust_id, order date;
```

CUST_ID	ORDER_DATE
---------	------------

```
-----  
CUST_ID ORDER_DATE  
-----  
224038 02-MAY-15  
224038 26-MAY-15  
227105 12-FEB-15  
227105 12-FEB-15  
227105 12-JUN-15  
227105 30-JUL-15  
227105 03-AUG-15  
227105 12-AUG-15  
227105 12-AUG-15  
227105 12-AUG-15  
227105 12-SEP-15  
227105 19-SEP-15  
227105 18-NOV-15  
227105 06-DEC-15  
227105 12-DEC-15  
227105 12-DEC-15  
227105 31-DEC-15  
272787 15-FEB-15  
272787 02-MAR-15  
272787 12-MAR-15  
272787 13-MAR-15  
272787 08-APR-15  
272787 15-JUN-15  
272787 16-JUN-15  
272787 02-JUL-15  
272787 22-SEP-15  
272787 22-SEP-15  
272787 30-SEP-15  
272787 02-NOV-15  
272787 02-NOV-15  
272787 06-NOV-15  
272787 12-NOV-15  
272787 12-NOV-15  
272787 02-DEC-15
```

35 rows selected.

This is the query to display the islands for those customers.

```
with dataset as (  
    select cust_id  
    , extract( month from order_Date) as mn  
    , extract( month from order_Date) - dense_rank()  
    over (partition by cust_id order by extract( month from order_Date))as grp  
from bk_order_headers  
where extract( YEAR from order_Date) = 2015  
and cust_id in (224038, 227105, 272787)  
)  
select cust_id  
, MIN(mn) as month_start  
, MAX(mn) as month_end  
from dataset  
group by cust_id, grp  
order by cust_id, grp;
```

CUST_ID	MONTH_START	MONTH_END
224038	5	5
227105	2	2
227105	6	9
227105	11	12
272787	2	4
272787	6	7
272787	9	9
272787	11	12

8 rows selected.

If you are going to work with actual date values, then you will need to use date arithmetic function to get the difference. Work with the CTE expression to get the same difference value for each island.

3. Lead and Lag

It might be useful to see the data for one day and also the previous day's data on the same line. For this we can use the Lag function. The Lag function also uses an Over clause and the default is one value previous. There is also a Lead function that shows the next value. If there is no previous (or next) value then a Null is returned.

Demo 06: Lag(attrb) and Lead(attrb)

```
select sales_day
, sales
, Lag(sales) Over (order by sales_day) as PrevDay
, Lead(sales) Over (order by sales_day) as NextDay
from adv_sales
order by sales_day;
```

DAY	SALES	PREVDAY	NEXTDAY
25-APR-15	400		400
26-APR-15	400	400	400
27-APR-15	400	400	300
28-APR-15	300	400	900
29-APR-15	900	300	580
30-APR-15	580	900	425
01-MAY-15	425	580	10
02-MAY-15	10	425	325
03-MAY-15	325	10	500
04-MAY-15	500	325	550
. . . rows omitted			

You can specify how many days Lag you want. The nulls here show where the missing data would be.

Demo 07: Lag(attr, n)

```
select sales_day
, sales
, Lag(sales, 3) Over (order by sales_day) as "3_DaysAgo"
, Lag(sales, 2) Over (order by sales_day) as "2_DaysAgo"
, Lag(sales, 1) Over (order by sales_day) as "1_DayAgo"
, Lag(sales, 0) Over (order by sales_day) as "ThisDay"
from adv_sales
order by sales_day;
```

DAY	SALES	3_DaysAgo	2_DaysAgo	1_DayAgo	ThisDay
25-APR-15	400				400
26-APR-15	400			400	400
27-APR-15	400		400	400	400
28-APR-15	300	400	400	400	300
29-APR-15	900	400	400	300	900
30-APR-15	580	400	300	900	580
01-MAY-15	425	300	900	580	425
02-MAY-15	10	900	580	425	10
03-MAY-15	325	580	425	10	325
04-MAY-15	500	425	10	325	500
05-MAY-15	550	10	325	500	550
. . .					

There is a third argument to supply a value for the nulls that occur as a return from the Lag and Lead. This value is not used for a null that occurs in the table data.

4. Gaps

The missing values between the islands are called gaps. You can use Lead to find the gaps. This is using the order id demo from above.

Demo 08:

```
with dataset as (
    select order_id as TheCurrentID
    , lead(order_id) over ( order by order_id) as TheNextID
    from oe_orderheaders
    where order_id between 400 and 520
)
select TheCurrentID + 1 as startOfGap
, TheNextID - 1 as EndOfGap
from dataset
where TheNextID - TheCurrentID >1;
```

STARTOFGAP	ENDOFGAP
403	404
406	406
409	410
416	518

4 rows selected.

5. Percent_Rank and Cume_Dist

These are two more ranking functions

Demo 09: The row filters are to reduce the output to make this easier to understand

```
Select emp_id, salary
, Rank() Over (order by salary) as rank
, round(PERCENT_RANK() Over (order by salary), 2 ) as Percentrank
, round(CUME_DIST() Over (order by salary), 2 ) as CumeDist
from adv_emp
```

```
where salary is not null
and dept_id in (30)
;
```

EMP_ID	SALARY	RANK	PERCENTRANK	CUMEDIST
313	11000	1	0	.11
318	11500	2	.13	.22
307	13500	3	.25	.33
308	15000	4	.38	.44
317	25000	5	.5	.56
316	26000	6	.63	.67
306	28000	7	.75	.89
304	28000	7	.75	.89
314	30000	9	1	1

These two functions are similar in that the ranking values increase as the rank increases.

The Percent_rank starts at 0 for the first row and then calculates its value as

$$(\text{rank} - 1) / (\text{number_of_rows} - 1)$$

The lowest ranked item is 0% and the highest is 100%. If there are ties for the highest position then we do not achieve 100%. If we have ties for the lowest position, then those tied rows are all 0%

Cume_Dist returns a value between 0 and 1 which is the number of rows ranked lower than or equal to the current row, including the current row, divided by the number of rows in the partition. The demo above does not do a partition so the number of rows is based on the table.

I know these can be pretty intimidating (and look a lot like statistics) but I hope you can see that these are important techniques that businesses need to use. And if you take these slowly, you can work them out.

Table of Contents

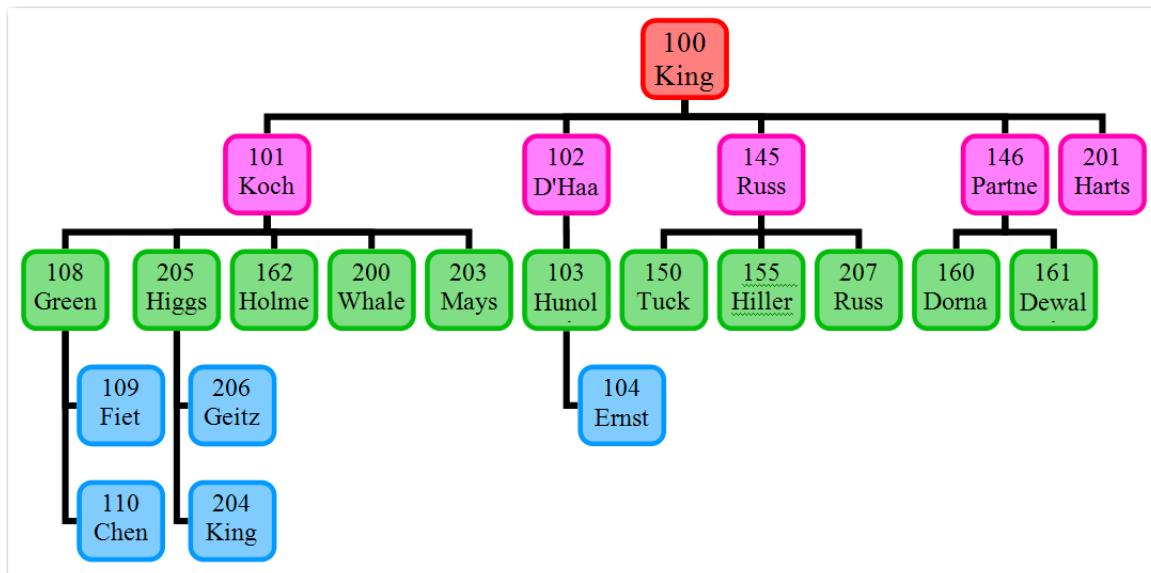
1.	Hierarchical Data	1
1.1.	Flattening hierachal data for a table	1
2.	Start With & Connect By Prior	2
3.	connect_by_isleaf.....	5
4.	SYS_CONNECT_BY_PATH.....	6
5.	Order Siblings By.....	7

This is one of those techniques that looks interesting, has a fairly complication syntax, and it quite powerful if you have data that that this type of a hierarchical organization. Remember this is optional.

1. Hierarchical Data

We often use hierarchical organization when think about or organizing data. The hierarchy chart here makes it pretty obvious which employee reports to whom. This is commonly called a tree structure and we use terms such as node, leaf, branch, and root; we also refer to this organization using the terms parent and child. The basic rules for a tree structure are that there is one root node and that each node other than the root has one parent. There are no circular branches.

This is a tree showing our employees and their managers. Employee 101 reports to employee 100 and employee 205 reports to employee 101 and employees 206 and 204 report to employee 205.

**1.1. Flattening hierachal data for a table**

When we store this data in a relational table we have to flatten it; we do this by placing the parent id into the child row. The root row commonly has its parent left as a null. This is the view listing. The view definition is in the demo.

```
select emp_id, name_last, mng, salary
from hier_emp;
```

EMP_ID	NAME_LAST	MNG
100	King	
201	Harts	100
101	Koch	100
108	Green	101
205	Higgs	101

102 D'Haa	100
103 Hunol	102
104 Ernst	103
145 Russ	100
150 Tuck	145
155 Hiller	145
162 Holme	101
200 Whale	101
207 Russ	145
203 Mays	101
146 Partne	100
109 Fiet	108
160 Dorna	146
161 Dewal	146
110 Chen	108
206 Geitz	205
204 King	205

There are a number of functions and expressions which Oracle supplies for working with this type of data.

2. Start With & Connect By Prior

For the basic query to follow the tree you can use the Start With clause to specify at what point in the tree you want to start. And you add the Connect by Prior to show the relationships- here we want to say that the prior element of the employee id is the manager id—i.e. that we travel up the tree from an employee to their manager.

Demo 01: Using start with and connect by prior

```
select emp_id, name_last, mng
from hier_emp
start with mng is null
connect by prior emp_id = mng
order by mng nulls first, emp_id
;
```

This just shows us the same table display as before.

Demo 02: Now add a level pseudo column

```
select level, emp_id, name_last, mng
from hier_emp
start with mng is null
connect by prior emp_id = mng
order by mng nulls first, emp_id
;
```

LEVEL	EMP_ID	NAME_LAST	MNG
1	100	King	
2	101	Koch	100
2	102	D'Haa	100
2	145	Russ	100
2	146	Partne	100
2	201	Harts	100
3	108	Green	101
3	162	Holme	101
3	200	Whale	101
3	203	Mays	101
3	205	Higgs	101

3	103 Hunol	102
4	104 Ernst	103
4	109 Fiet	108
4	110 Chen	108
3	150 Tuck	145
3	155 Hiller	145
3	207 Russ	145
3	160 Dorna	146
3	161 Dewal	146
4	204 King	205
4	206 Geitz	205

Although this does not seem very interesting, the level column does tell us how far down the tree an employee is. Employee 103 Huno is on the third level while employee 206 Geitz is at the fourth level. That is new information.

Demo 03: Change the Start with column. Now our tree starts with manager 101 and employees who report to him directly or indirectly

```
select level, emp_id, name_last, mng
from hier_emp
START WITH mng = 101
connect by prior emp_id = mng
order by mng nulls first, emp_id;
```

LEVEL	EMP_ID	NAME_LAST	MNG
1	108 Green	101	
1	162 Holme	101	
1	200 Whale	101	
1	203 Mays	101	
1	205 Higgs	101	
2	109 Fiet	108	
2	110 Chen	108	
2	204 King	205	
2	206 Gietz	205	

Demo 04: If we start further down the tree we get fewer rows returned.

```
select level, emp_id, name_last, mng
from hier_emp
START WITH mng = 205
connect by prior emp_id = mng
order by mng nulls first, emp_id;
```

LEVEL	EMP_ID	NAME_LAST	MNG
1	204 King	205	
1	206 Gietz	205	

Demo 05: How many levels do we have?

```
select count(distinct level)
from hier_emp
start with mng is null
connect by prior emp_id = mng
;
```

COUNT(DISTINCT LEVEL)

4

Demo 06: How many employees at each level?

```
select level
, count(emp_id)
from hier_emp
start with mng is null
connect by prior emp_id = mng
group by level
order by level
;
```

LEVEL	COUNT(EMP_ID)
1	1
2	5
4	5
3	11

Demo 07: Variations- here we have two trees . One start with emp 101 and the other starts with emp 146

```
select level, emp_id, name_last, mng
from hier_emp
start with mng in (101, 146)
connect by prior emp_id = mng
;
```

LEVEL	EMP_ID	NAME_LAST	MNG
1	108	Green	101
2	109	Fiet	108
2	110	Chen	108
1	162	Holme	101
1	200	Whale	101
1	203	Mays	101
1	205	Higgs	101
2	204	King	205
2	206	Gietz	205
1	160	Dorna	146
1	161	Dewal	146

Demo 08: Is employee 204 on the branch that starts with employee 101?

```
Select name_last, emp_id
from hier_emp
where emp_id = 204
start with emp_id = 101
connect by mng = prior emp_id;
```

NAME_LAST	EMP_ID
King	204

Demo 09: Is employee 207 on the branch that starts with employee 101?

```
Select name_last, emp_id
from hier_emp
where emp_id = 207
start with emp_id = 101
connect by mng = prior emp_id
;
```

no rows returned

Demo 10: How many people are on the branch starting at employee 146

```
select count(*)
from hier_emp E1
start with emp_id = 146
connect by mng = prior emp_id
;

```

COUNT(*)

3

3. connect_by_isleaf

This function returns a 1 if the row is a leaf and 0 if it is not.

Demo 11: Which employees are not managers?

```
select level, emp_id, name_last
from hier_emp
where connect_by_isleaf = 1
start with mng is null
connect by prior emp_id = mng
order by mng nulls first, emp_id;
```

LEVEL	EMP_ID	NAME_LAST
2	201	Harts
3	162	Holme
3	200	Whale
3	203	Mays
4	104	Ernst
4	109	Fiet
4	110	Chen
3	150	Tuck
3	155	Hiller
3	207	Russ
3	160	Dorna
3	161	Dewal
4	204	King
4	206	Gietz

Demo 12: Which employees are managers?

```
select level, emp_id, name_last
from hier_emp
where connect_by_isleaf = 0
start with mng is null
connect by prior emp_id = mng
order by mng nulls first, emp_id;
```

LEVEL	EMP_ID	NAME_LAST
1	100	King
2	101	Koch
2	102	D'Haa
2	145	Russ
2	146	Partne
3	108	Green
3	205	Higgs
3	103	Huno

4. SYS_CONNECT_BY_PATH

SYS_CONNECT_BY_PATH returns the path of a column value from root to node, with column values separated by a string literal for each row returned by CONNECT BY condition.

Demo 13:

```
select SYS_CONNECT_BY_PATH(emp_id || ' ' || name_last, ' -- ')
       as "Path"
  from hier_emp
 start with emp_id = 101
 connect by prior emp_id = mng
;

```

Path
-- 101 Koch
-- 101 Koch -- 108 Green
-- 101 Koch -- 108 Green -- 109 Fiet
-- 101 Koch -- 108 Green -- 110 Chen
-- 101 Koch -- 162 Holme
-- 101 Koch -- 200 Whale
-- 101 Koch -- 203 Mays
-- 101 Koch -- 205 Higgs
-- 101 Koch -- 205 Higgs -- 204 King
-- 101 Koch -- 205 Higgs -- 206 Gietz

10 rows selected.

To get the full tree, use: start with mng is null

Demo 14: Sometimes the nested levels are indented by using lpad

```
select lpad(' ', 6*level-1) ||
       SYS_CONNECT_BY_PATH(emp_id || ' ' || name_last, '---') AS "Path"
  from hier_emp
 start with emp_id = 101
 connect by prior emp_id = mng
;

```

Path
--101 Koch
--101 Koch--108 Green
--101 Koch--108 Green--109 Fiet
--101 Koch--108 Green--110 Chen
--101 Koch--162 Holme
--101 Koch--200 Whale
--101 Koch--203 Mays
--101 Koch--205 Higgs
--101 Koch--205 Higgs--204 King
--101 Koch--205 Higgs--206 Gietz

10 rows selected.

5. Order Siblings By

This option lets you have the sibling rows sorted. The siblings are rows of the same parent row.

Demo 15:

```
select lpad(' ', 6*level-1) ||
       SYS_CONNECT_BY_PATH(emp_id || ' ' || name_last, '--') AS "Path"
  from hier_emp
 start with emp_id = 101
 connect by prior emp_id = mng
 order siblings by name_last
;

```

Path
--101 Koch
--101 Koch--108 Green
--101 Koch--108 Green--110 Chen
--101 Koch--108 Green--109 Fiet
--101 Koch--205 Higgs
--101 Koch--205 Higgs--206 Gietz
--101 Koch--205 Higgs--204 King
--101 Koch--162 Holme
--101 Koch--203 Mays
--101 Koch--200 Whale

10 rows selected.

Demo 16:

```
select lpad(' ', 6*level-1) || emp_id || ' ' || name_last AS "Path"
, mng
  from hier_emp
 start with emp_id = 101
 connect by prior emp_id = mng
 order siblings by name_last
;
```

Path	MNG
101 Koch	100
108 Green	101
110 Chen	108
109 Fiet	108
205 Higgs	101
206 Gietz	205
204 King	205
162 Holme	101
203 Mays	101
200 Whale	101

10 rows selected.

Table of Contents

1.	An example of an XML document	1
1.1.	Experiments	2
2.	Why XML is helpful	3
3.	What are some of the rules for xml data?.....	4
3.1.	The version line	4
3.2.	A well-formed document needs a root element	4
3.3.	Rules for tags	4
3.4.	Element content	4
3.5.	Elements and subelements	5
3.6.	Attributes.....	5
3.7.	Empty elements	6
4.	Some of our limitations in this discussion	6
5.	Using XML in Other Applications	6
6.	xml Schema.....	7

XML is an important collection of technologies for the exchange of data. The XML format for data is not owned by MySQL or Oracle or Microsoft and is intended to be a way to store and manipulate structured data. We will look at a part of what Oracle can do with XML.

We are going to start with a demo and then follow with an explanation. Please do these steps- don't just read about them unless you already have a good grasp of XML. (XML is not the same as HTML.)

1. An example of an XML document

Copy the following lines and paste them into a text editor such as Notepad. Save the file with the name pets_01.xml.(pets_01.xml file is in the demos)

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--XML document using elements -->
<pets>
  <animal>
    <an_id>136</an_id>
    <an_type>bird</an_type>
    <an_name>ShowBoat</an_name>
    <an_price>75</an_price>
    <an_dob>2000-01-15</an_dob>
  </animal>
  <animal>
    <an_id>137</an_id>
    <an_type>bird</an_type>
    <an_name>Mr. Peanut</an_name>
    <an_price>150</an_price>
    <an_dob>2008-01-12</an_dob>
  </animal>
  <animal>
    <an_id>1201</an_id>
    <an_type>cat</an_type>
    <an_name>Ursula</an_name>
    <an_price>500</an_price>
    <an_dob>2007-12-15</an_dob>
  </animal>
</pets>
```

On a windows machine, the file icon probably looks like this



Right click this file and open it in your browser. Allowing Active X components to run, I get the following display in IE7 browser. The + and - symbols allow me to open or collapse the structure. You should get a similar display in other browsers.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<!-- XML document using elements  --&gt;
- &lt;pets&gt;
- &lt;animal&gt;
&lt;an_id&gt;136&lt;/an_id&gt;
&lt;an_type&gt;bird&lt;/an_type&gt;
&lt;an_name&gt;ShowBoat&lt;/an_name&gt;
&lt;an_price&gt;75&lt;/an_price&gt;
&lt;an_dob&gt;2000-01-15&lt;/an_dob&gt;
&lt;/animal&gt;
- &lt;animal&gt;
&lt;an_id&gt;137&lt;/an_id&gt;
&lt;an_type&gt;bird&lt;/an_type&gt;
&lt;an_name&gt;Mr. Peanut&lt;/an_name&gt;
&lt;an_price&gt;150&lt;/an_price&gt;
&lt;an_dob&gt;2008-01-12&lt;/an_dob&gt;
&lt;/animal&gt;
- &lt;animal&gt;
&lt;an_id&gt;1201&lt;/an_id&gt;
&lt;an_type&gt;cat&lt;/an_type&gt;
&lt;an_name&gt;Ursula&lt;/an_name&gt;
&lt;an_price&gt;500&lt;/an_price&gt;
&lt;an_dob&gt;2007-12-15&lt;/an_dob&gt;
&lt;/animal&gt;
&lt;/pets&gt;</pre>
```

Most of you are probably at least somewhat familiar with html and this looks a bit like html- but it isn't- I don't have any of the html tags that you may know except for the comment tag.

This is a simple XML document. The first thing to notice is that this is all text and except for the first line it is humanly-readable and understandable.

It looks like we are storing data about animals and animal have an id, a name, a type, a price and a date of birth. If I asked you to include data about a dog (named Blue who costs 450 with ID of 834 born on April 3 2001) you could add that data easily. Probably you would copy one set of the animal lines and edit it because you can see that this document has a structure.

- The words inside the angle brackets are called tags.
- This document includes data values that are organized/structured.
- We have three animals- the first animal is a bird named ShowBoat and the third is a cat named Ursula.
- For each beginning tag we have an ending tag which includes a /: <an_name>...</an_name>
- The tags name the data components.
- We have data values between the tags
- The tagged elements can be nested- the <an_name> is inside the <animal> and all of the animals are inside <pets> Note the ending tag for pets </pets> at the bottom.

1.1. Experiments

Do a few experiments with this file. (Really- do this- it will save you a lot of time and frustration if you try these out now.) You should be able to keep the editor open with the file and your browser open. Then make changes to the file, save it and refresh your browser. For each of these, do a single change and if it causes an error- fix the error before going to the next step.

- Try adding another animal to this file. Follow the pattern used
- Change the starting tag <an_id> for one of the rows to <an_ID> but don't change its ending tag.
- Remove one of the ending tags.
- Remove one of the an_id lines completely.
- Put a single blank space at the front of one of the tags: < an_id>
- Change one of the dob values to something that is not a valid date.

From this you can see that xml is fussier about syntax than SQL. And a lot fussier than html!

2. Why XML is helpful

There are a lot of different ways to store data in a computer system. We could use csv (comma separated values) files, fixed length record files, spreadsheets, relational database tables, and xml. XML is another format for storing data that also stores information about the structure of the data.

One of the advantages of storing data in a csv file is that the data is in text format and therefore generally readable by different computer systems. But a csv does not include metadata. You can figure out the following lines of a csv file because it stores the data we have been talking about- but the file itself does not store the metadata.

```
bird,ShowBoat,136,75,2000-01-15
bird,Mr. Peanut,137,150,2008-01-12
cat,Ursula,1201,500,2007-12-15
```

One problem with storing data in a relational database system is that the actual format of the data is proprietary and not always sharable across systems. Believe it or not, there are some people who do not have access to an Oracle database and we might want to share data with those people.

XML data is commonly used to transfer data across different computer systems. XML is all text so it does not use proprietary data formats. People who need to share data could agree on the tag names to use and the organization of the tags for the data – this set of rules for the organization of the data is called an xml schema- the xml schema is written in xml. Many industries and fields of study have set up rules/schema for XML data and can share data more easily. XML is extensible in that you can define more tags as needed.

XML dates from 1996 and is part of W3C; XML is not owned by Microsoft, by Oracle or by any company. W3C provides some standards for XML; companies generally try to meet those standards but if a standard cannot be agreed on, then a company might implement its own version of an XML feature

XML is hierarchical. With a relational table, the order of the rows cannot be logically significant; with XML data the order of the lines can be significant. With relational data, we often store data in two or more tables with relationships between them. With XML data you are more apt to have a single document with the relational structure changed into a hierarchical structure. So there is not a complete matchup between relational data and xml data.

If we wanted to store data about a client and their pets, we would create two relational tables. In XML we might use the following. The root is now <client_list>. We have three <client> elements. Each client element includes a <pets> element. The first client has two <animal> elements in their <pets> element; the second client has one animal in their <pets> element; the third client has an empty <pets> element. It is possible with the XML structure to say that ShowBoat is the first animal for client 8243 and Mr. Peanut is the second. The ordering of the subelements in an XML document can be logically determined. (client_pets_01.xml)

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--XML document using elements -->
<client_list>
  <client>
    <cl_id>8243</cl_id>
    <cl_name>Johnson</cl_name>
    <pets>
      <animal>
        <an_id>"136"</an_id>
        <an_type>bird</an_type>
        <an_name>ShowBoat</an_name>
        <an_price>75</an_price>
        <an_dob>2000-01-15</an_dob>
      </animal>
      <animal>
        <an_id>"137"</an_id>
        <an_type>bird</an_type>
        <an_name>Mr. Peanut</an_name>
        <an_price>150</an_price>
        <an_dob>2008-01-12</an_dob>
      </animal>
    </pets>
  </client>
  <client>
    <cl_id>"1201"</cl_id>
    <cl_name>Ursula</cl_name>
    <pets>
    </pets>
  </client>
</client_list>
```

```

    </pets>
</client>
<client>
  <cl_id>3908</cl_id>
  <cl_name>Nelson</cl_name>
  <pets>
    <animal>
      <an_id>"1201"</an_id>
      <an_type>cat</an_type>
      <an_name>Ursula</an_name>
      <an_price>500</an_price>
      <an_dob>2007-12-15</an_dob>
    </animal>
  </pets>
</client>
<client>
  <cl_id>3775</cl_id>
  <cl_name>Miller</cl_name>
  <pets/>
</client>
</client_list>

```

3. What are some of the rules for xml data?

Syntax rules (simplified)

3.1. The version line

The first line of an XML document is generally

```
<?xml version="1.0" encoding="utf-8" ?>
```

This is a declaration statement that says you are going to follow the syntax rules for XML 1.0.

Note that the declaration tag delimiters includes a ? character. Do not put whitespace before this statement.

3.2. A well-formed document needs a root element.

The declaration is not enough to have a well-formed document. You need a root element.

In the first document <pets></ pets> is the root element. You could have an empty element; you can have a well formed xml document that consists of just an empty root element.

```
<?xml version="1.0" encoding="utf-8" ?>
<pets></pets>
```

<pets> is the start tag . </pets> is the end tag

3.3. Rules for tags

- Every element needs a start tag and an end tag; the end tag is identical to the start tag except that it includes a slash.
- XML element names are case specific.
- The element name cannot start with a space: < animal> is not acceptable; the element name can end with a space <animal > is OK. Do not put spaces inside an element name.

3.4. Element content

We can add content to the element.

```
<?xml version="1.0" encoding="utf-8" ?>
<pets>
  This is simple!
</pets>
```

In our longer example the content is in the sub elements, but you can have content in any element.

3.5. Elements and subelements

XML is hierarchical in nature; XML documents have a tree structure. An XML element can contain text or another element (a child element or a sub element). If you are storing structured data, you will probably have sub elements. Here the root element name is animal and the child element names are an_name and an_type.

```
<?xml version="1.0" encoding="utf-8" ?>
<animal>
  <an_name>Jayson</an_name>
  <an_type>bird</an_type>
</animal>
```

- A well-formed document has only one root element.
- Subelements must be completely contained within their parent element.
- Indentation is helpful for a human reader. Most application programs can be told to ignore whitespace.

Suppose we have two animals. This is not a well-formed document. It does not have an enclosing root element. The error message is generally expressed as the document having multiple root elements.

```
<?xml version="1.0" encoding="utf-8" ?>
<animal>
  <an_name>Tweetie</an_name>
  <an_type>bird</an_type>
</animal>

<animal>
  <an_name>Sylvester</an_name>
  <an_type>cat</an_type>
</animal>
```

We do have a name for a piece of a document; an **XML fragment**. It is well formed with the exception that it does not have a root element.

3.6. Attributes

You can use attributes – name=value pairs – for data values. You can use single or double quotes; it is better style to be consistent in a document. You can include white space around the = operator. (pets_02.xml)

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--XML document using elements -->
<pets>
  <animal an_id="136">
    <an_type>bird</an_type>
    <an_name>ShowBoat</an_name>
    <an_price>75</an_price>
    <an_dob>2000-01-15</an_dob>
  </animal>
  <animal an_id="137">
    <an_type>bird</an_type>
    <an_name>Mr. Peanut</an_name>
    <an_price>150</an_price>
    <an_dob>2008-01-12</an_dob>
  </animal>
  <animal an_id="120">
    <an_type>cat</an_type>
    <an_name>Ursula</an_name>
    <an_price>500</an_price>
    <an_dob>2007-12-15</an_dob>
  </animal>
</pets>
```

- You cannot have two attributes with the same name in the same element.
- If an attribute is storing a numeric value, the value is still entered as a text value enclosed in quotes. Your application may need to cast that string to a numeric value.

There is no general agreement about which items to store as attributes and which as subelements. Often an item which is an identifying item is handled as an attribute.

There are some situations where you have to use subelements:

- Any item which may be repeated has to be handled as an element.
- Any item which is itself complex and needs to be broken down into pieces should be handled as an element.
- If the order of the values is important, then they need to be handled as elements.

If you are used to writing HTML you may be in the habit of using attributes a lot:

```
<td class="body_title" colspan="2">
```

That makes sense for display attributes but it is not as useful for XML and data. When in doubt, use elements rather than attributes for data.

3.7. Empty elements

An element does not have to contain any content

- Empty elements can be written in either of two formats:

```
<an_name></an_name>
<an_name/>
```

4. Some of our limitations in this discussion

Because we are spending only one week on this topic, we cannot discuss everything about XML. Some things we will not work with:

- creating and using xml schemas
- combining xml and html
- all of the functions that Oracle provides for xml data; we will talk about some functions
- all of the features of XML Query; we will look at some XPath features

5. Using XML in Other Applications

Many applications today include the ability to handle xml data.

Take one of the XML documents; open it in Excel. This uses the choice to open the XML file as an XML list.

	A	B	C	D	E
1	an_id	an_type	an_name	an_price	an_dob
2	136	bird	ShowBoat	75	2000-01-15
3	137	bird	Mr. Peanut	150	2008-01-12
4	1201	cat	Ursula	500	2007-12-15
5	*				
6					

Excel provides sorting options

	A	B	C	D	E
1	an_id	an_type	an_name	an_price	an_dob
2	136	bird	ShowBoat	75	2000-01-15
3	137	bird	Mr. Pear	150	2008-01-12
4	1201	cat	Ursula	500	2007-12-15
5	*				
6					
7					
8					
9					
10					

6. xml Schema

If you have a copy of Visual Studio 2008 available, create a project; add an xml file to the project and copy in your xml data. Then tell VS to generate an xml Schema. This is the generated schema for our file. You can see that it describes the file and seems to get the data types correct.

The XML Schema file can be used to check that the data that you are inserting into the XML column is valid according to data types. Oracle can be told to associate an XMLS schema with the XML data column.

The word "schema" has several different meanings- in a discussion of xml and xml schemas it refers to the file that describes the structure of the xml document.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="pets">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="animal">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="an_id" type="xs:unsignedShort" />
                            <xs:element name="an_type" type="xs:string" />
                            <xs:element name="an_name" type="xs:string" />
                            <xs:element name="an_price" type="xs:unsignedShort" />
                            <xs:element name="an_dob" type="xs:date" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Examples of the kinds of validation that can be specified by a schema include:

- The order in which the individual sub elements must appear
- Whether or not an element is optional
- The minimum and maximum number of times an element can be repeated.
- Data type- such as date, integer, string

More complex validation can include

- Range tests
- List tests
- Limitations on the length of a string

A column in a table, a variable, or a value returned by a function can be associated with a schema- that xml would be called typed XML. XML that is not associated with a schema is called untyped XML. If you have a typed xml object and you try to assign an xml value that does not meet the schema rules, that assignment would be rejected.

Table of Contents

1.	Creating a table with an XML column.....	1
1.1.	Create	1
1.2.	Describe	1
1.3.	Inserting data into XmlAnimals	2
1.4.	Displaying the data	4
2.	The second demo table	5
3.	Creating an XML table	8
3.1.	Create	8
3.2.	Describe	8
3.3.	Inserting data into XmlOnlyAnimals	8
3.4.	Displaying the data	9

Oracle has a native data type called **xmltype** which stores well-formed XML. The alternative in the past was to store the XML as a string (varchar2 or clob) and you had to handle all of the XML rules for the data being stored. That can make sense if you are simply storing the data 'raw' - and using the database table as a storage container. But if you want to work with the inner structures of the XML document, it is better to have a native xml type.

We can use the Oracle XmlType as the data type for a column in a table, as the data type for a table, and we can use it as a data type in PL/SQL. The first thing to understand about this is that xml data is considered to be an object in Oracle. It has an internal structure that can be accessed. So the syntax will vary a bit from the SQL syntax that we are used to- but it is not a lot different.

We will start with creating three tables that can store XML data using the native XML data type.

- XmlAnimals will have an integer column (pk) and an xml column
- XmlBooks will have an integer column (pk) and an xml column
- XmlOnlyAnimals will be an xml only table

The pk column in the first two tables is mostly used as a way to filter for specific rows to display for certain techniques. The pk value is not related to the data in the XML column.

1. Creating a table with an XML column

1.1. Create

Demo 01: create table with an xml column

```
create table XmlAnimals (
    id      number(5) primary key
, anXmlData xmltype);
```

Here we just use the keyword `xmltype` as the data type for the column. The table has a numeric primary key to make it easier to deal with the rows. `anXmlData` is not a key word- it is just the name I use for that column.

1.2. Describe

Demo 02:

```
desc XmlAnimals
```

Name	Null?	Type
ID	NOT NULL	NUMBER(5)
ANXMLDATA		PUBLIC.XMLTYPE

1.3. Inserting data into XmlAnimals

Demo 03: insert data

```
insert into XmlAnimals (id, anXmlData) values (
1,
xmltype.createXml(
'<animal>
<an_id>136</an_id>
<an_type>bird</an_type>
<an_name>ShowBoat</an_name>
<an_price>75</an_price>
<an_dob>2000-01-15</an_dob>
</animal>
')
);
```

There is a method, `createXml`, that is part of the `xmltype` that creates an xml data value from a string. The above insert has two values- 1 for an id and the xml data for the second column. Since whitespace outside of the elements is not significant in our use of xml, I can build the xml string one element at a time. Note the quote delimiters for the string. The method can be invoked as

```
xmltype.createXml('MyStringLiteral')
```

You know that Oracle does a lot of implicit casting for you. You can use that here where the string is implicitly cast to xml because the column's data type is `xmltype`.

```
insert into XmlAnimals (id, anXmlData) values (
2,
'<animal>Fluffy</animal>'
);
```

If you do a `Select *` from the table, you will get a result set such as this. The second column will be very wide. The XML values are each valid, but they do not have the same pattern.

```
SQL> select * from XmlAnimals;
      ID
ANXMLDATA
-----
-----  

      1
<animal>
<an_id>136</an_id>
<an_type>bird</an_type>
<an_name>ShowBoat</an_name>
<an_price>75</an_price>
<an_dob>2000-01-15</an_dob>
</animal>  

      2
<animal>Fluffy</animal>

2 rows selected.
```

Demo 04: Try doing an insert where the xml string is not valid xml. The string will be rejected. You should be able to see the error.

```
insert into XmlAnimals (id, anXmlData) values (3,'<animal>Fluffy</Animal>');
insert into XmlAnimals (id, anXmlData) values (3,'<animal>Fluffy</Animal>')
*
ERROR at line 1:
ORA-31011: XML parsing failed
ORA-19202: Error occurred in XML processing
LPX-00225: end-element tag "Animal" does not match start-element tag "animal"
Error at line 1
```

```

insert into XmlAnimals (id, anXmlData) values (3,'<animal>Fluffy</animal>');
insert into XmlAnimals (id, anXmlData) values (3,'<animal>Fluffy</animal>')
*
ERROR at line 1:
ORA-31011: XML parsing failed
ORA-19202: Error occurred in XML processing
LPX-00007: unexpected end-of-file encountered

```

Demo 05: This animal has two names- how would you handle that in a regular database table?

We can use the id column in a filer as before.

```

insert into XmlAnimals (id, anXmlData) values
(3,'<animal><an_name>Fluffy</an_name><an_name>Snookums</an_name></animal>');
1 row created.

SQL> select * from XmlAnimals where id = 3;
      ID
ANXMLDATA
-----
-----3-----<animal><an_name>Fluffy</an_name><an_name>Snookums</an_name></animal>
1 row selected.

```

Demo 06: More invalid inserts

```

insert into XmlAnimals (id, anXmlData) values
(4,'<animal>Fluffy</animal><animal>Snookums</animal>');
insert into XmlAnimals (id, anXmlData) values
(4,'<animal>Fluffy</animal><animal>Snookums</animal>')
*
ERROR at line 1:
ORA-31011: XML parsing failed
ORA-19202: Error occurred in XML processing
LPX-00245: extra data after end of document
Error at line 1

```

```

insert into XmlAnimals (id, anXmlData) values
(5,'<animal>Fluffy</animal><pet><an_price>75</an_price></pet>');
insert into XmlAnimals (id, anXmlData) values
(5,'<animal>Fluffy</animal><pet><an_price>75</an_price></pet>')
*
ERROR at line 1:
ORA-31011: XML parsing failed
ORA-19202: Error occurred in XML processing
LPX-00245: extra data after end of document
Error at line 1

```

What we want now is rows that do have the same pattern. If we had associated this column with an XML schema, we could enforce a pattern for the data. Since we are not going into that topic, we will just take care that our data follows a pattern that an animal element has one of each of the following subelements: an_id, an_type, an_name (all of those are strings), an_price (a number) and an_dob (a date value). The date value literal will use an Xml date format YYYY-MM-DD

Demo 07: Delete rows to clear the table. This is the same as any SQL delete.

```
delete from XmlAnimals ;
```

Demo 08: insert three rows

```

insert into XmlAnimals  (id, anXmlData) values (1,
xmldtype.createXml(
'<animal>
<an_id>136</an_id>
<an_type>bird</an_type>
<an_name>ShowBoat</an_name>
<an_price>75</an_price>
<an_dob>2000-01-15</an_dob>
</animal>
')
);

insert into XmlAnimals  (id, anXmlData) values (9,
'<animal>
<an_id>137</an_id>
<an_type>bird</an_type>
<an_name>Mr. Peanut</an_name>
<an_price>150</an_price>
<an_dob>2008-01-12</an_dob>
</animal>');

insert into XmlAnimals  (id, anXmlData) values (14,
'<animal>
<an_id>1201</an_id>
<an_type>cat</an_type>
<an_name>Ursula</an_name>
<an_price>500</an_price>
<an_dob>2007-12-15</an_dob>
</animal>');

```

1.4. Displaying the data

Demo 09: Start with a plain select. I have reformatted the data a bit to fit on the page.

Start with the set commands shown or the output may be truncated

```

set Long 999999
set feedback 1
select * from XmlAnimals ;

```

ID	ANXMLDATA
1	<animal> <an_id>136</an_id> <an_type>bird</an_type> <an_name>ShowBoat</an_name> <an_price>75</an_price> <an_dob>2000-01-15</an_dob> </animal>
9	<animal> <an_id>137</an_id> <an_type>bird</an_type> <an_name>Mr. Peanut</an_name> <an_price>150</an_price> <an_dob>2008-01-12</an_dob> </animal>
14	<animal> <an_id>1201</an_id> <an_type>cat</an_type> <an_name>Ursula</an_name> <an_price>500</an_price> <an_dob>2007-12-15</an_dob> </animal>

2. The second demo table

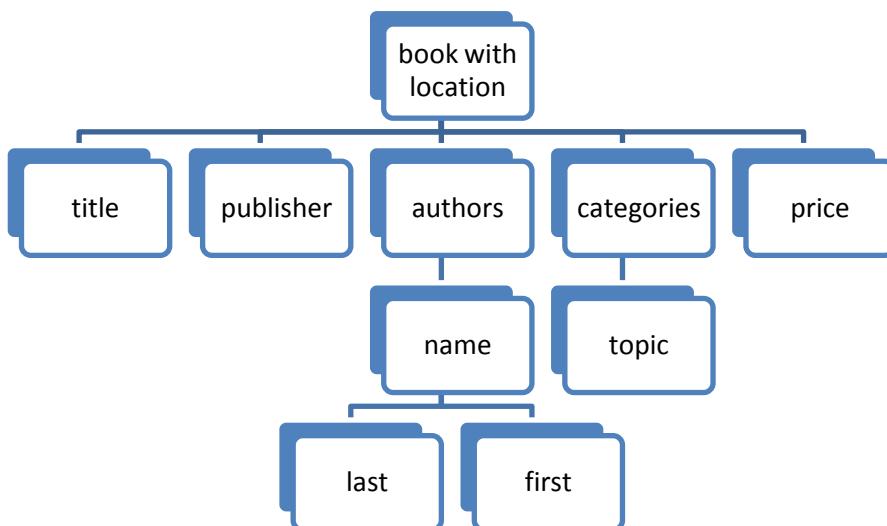
This set of XML has a deeper structure. For each book, we start with a publisher with a name; then we have a book element with a book name; then an author element with possible multiple author names with a first and last name element, then a categories element with possible multiple topics. The book element has an attribute named location. This is storing the same type of data we stored in multiple tables using traditional database tables.

There is another document which has a generated schema for this xml document.

This is one XML string

```
<book location="275">
  <title>SQL is Fun</title>
  <publisher>Addison Wesley</publisher>
  <authors>
    <name>
      <last>Collins</last>
      <first>Joan</first>
    </name>
    <name>
      <last>Effingham</last>
      <first>Jill</first>
    </name>
  </authors>
  <categories>
    <topic>SQL</topic>
    <topic>DB</topic>
  </categories>
  <price>29.95</price>
</book>
```

You can use this picture to help see the various paths through the data. Each of these boxes can be considered an element node in the tree



Demo 10: create the table

```
create table XmlBooks (
  id int primary key
, book_xml xmltype)
;
```

Demo 11: These are the first two inserts- see the demo for the rest- there are 10 inserts.

```
insert into XMLBooks values (1,
'<book location="275">
    <title>SQL is Fun</title>
    <publisher>Addison Wesley</publisher>
    <authors>
        <name>
            <last>Collins</last><first>Joan</first>
        </name>
        <name>
            <last>Effingham</last><first>Jill</first>
        </name>
    </authors>
    <categories>
        <topic>SQL</topic>
        <topic>DB</topic>
    </categories>
    <price>29.95</price>
</book>'
);

insert into XMLBooks values (2,
'<book location="275">
    <title>Databases: an Introduction</title>
    <publisher>Addison Wesley</publisher>
    <authors>
        <name>
            <last>Malone</last>
            <first>Mike</first>
        </name>
        <name>
            <last>Effingham</last>
            <first>Jill</first>
        </name>
    </authors>
    <categories>
        <topic>SQL</topic>
        <topic>DB</topic>
        <topic>XML</topic>
    </categories>
    <price>79.95</price>
</book>'
);
```

Demo 12: Displaying the table- this is a partial display

```
select * from XmlBooks;
```

```
SQL> select * from NullBooks;
   ID
BOOK_HML
-----  
  
1  
<book location="275">  
  <title>SQL is Fun</title>  
  <publisher>Addison Wesley</publisher>  
  <authors>  
    <name>  
      <last>Collins</last><first>Joan</first>  
    </name>  
    <name>  
      <last>Effingham</last><first>Jill</first>  
    </name>  
  </authors>  
  <categories>  
    <topic>GL</topic>  
    <topic>DB</topic>  
  </categories>  
  <price>29.95</price>  
</book>  
  
2  
<book location="275">  
  <title>Databases: an Introduction</title>  
  <publisher>Addison Wesley</publisher>  
  <authors>  
    <name>  
      <last>Malone</last><first>Mike</first>  
    </name>  
    <name>  
      <last>Effingham</last><first>Jill</first>  
    </name>  
  </authors>  
  <categories>  
    <topic>GL</topic>  
    <topic>DB</topic>  
    <topic>ML</topic>  
  </categories>  
  <price>79.95</price>  
</book>  
  
3  
<book location="313">  
  <title>GE and MySQL</title>  
  <publisher>Addison Wesley</publisher>  
  <authors>  
    <name><last>Horn</last><first>Shirley</first></name>  
    <name><last>Cocher</last><first>Loe</first></name>  
    <name><last>Shorter</last><first>Wayne</first></name>  
  </authors>  
  <categories>  
    <topic>GL</topic>  
    <topic>DB</topic>  
    <topic>ML</topic>  
    <topic>GM</topic>  
    <topic>MySQL</topic>  
  </categories>  
</book>  
  
4  
<book location="493:75">  
  <title>Databases— Design and Logic</title>  
  <publisher>McGraw Hill</publisher>  
  <authors>  
    <name><last>LeUette</last><first>Betty</first></name>  
  </authors>  
  <categories>  
  </categories>  
  <price>185.95</price>  
</book>  
  
5  
<book location="558:8">  
  <title>The truth about everything</title>  
  <publisher>Addison Wesley</publisher>  
  <authors>  
    <name><last>Bel Amane</last></name>  
  </authors>  
  <categories>  
    <topic></topic>  
  </categories>  
  <price>79.95</price>  
</book>
```

3. Creating an XML table

The previous tables had an integer column and an xml column. We can also create a table that stores only xml-one xml value per row.

3.1. Create

xmltype is an object type so we can create a table of that type, rather than just defining a column of the xml type.

Demo 13:

```
create table XmlOnlyAnimals of XmlType;
```

3.2. Describe

Demo 14: Describing the table

```
desc XmlOnlyAnimals
```

Name	Null?	Type

If you do this in SQL Developer, the display is a bit different

Name	Null	Type

3.3. Inserting data into XmlOnlyAnimals

Use the same inserts as before except insert one value only; we do not have an ID column in this table. We also do not have a column name. You can use the createXml method or let Oracle handle the cast.

Demo 15: I am inserting three rows.

```
insert into XmlOnlyAnimals values (
xmltype.createXml(
'<animal>
<an_id>406</an_id>
<an_type>bird</an_type>
<an_name>ShowBoat</an_name>
<an_price>75</an_price>
<an_dob>2000-01-15</an_dob>
</animal>
')
);

insert into XmlOnlyAnimals values (
'<animal>
<an_id>407</an_id>
<an_type>bird</an_type>
<an_name>Mr. Peanut</an_name>
<an_price>150</an_price>
<an_dob>2008-01-12</an_dob>
</animal>');
;
```

```
insert into XmlOnlyAnimals values (
'<animal>
<an_id>401</an_id>
<an_type>cat</an_type>
<an_name>Ursula</an_name>
<an_price>500</an_price>
<an_dob>2007-12-15</an_dob>
</animal>')
;
```

3.4. Displaying the data

Demo 16: Note the name of the column for the xml data; this is a system name

```
select *
from XmlOnlyAnimals;
SYS_NC_ROWINFO$  
-----  
<animal>  
<an_id>406</an_id>  
<an_type>bird</an_type>  
<an_name>ShowBoat</an_name>  
<an_price>75</an_price>  
<an_dob>2000-01-15</an_dob>  
</animal>  
  
<animal>  
<an_id>407</an_id>  
<an_type>bird</an_type>  
<an_name>Mr. Peanut</an_name>  
<an_price>150</an_price>  
<an_dob>2008-01-12</an_dob>  
</animal>  
  
<animal>  
<an_id>401</an_id>  
<an_type>cat</an_type>  
<an_name>Ursula</an_name>  
<an_price>500</an_price>  
<an_dob>2007-12-15</an_dob>  
</animal>  
3 rows selected.
```

Demo 17: We can use the term `object_value` to refer to this column which is easier to remember and easier to type

```
select object_value
from XmlOnlyAnimals;
```

Table of Contents

1.	Querying the XML data	1
2.	Simple XPath expressions	2
3.	The extract Function	4
4.	Bad path expressions	5
5.	Including a predicate/test on the path	5
6.	Coalesce and CTEs.....	7
7.	Wandering around the path after a test.....	8
8.	The ExtractValue Technique -- Getting inside the xmldata	9
9.	The existsNode()Technique --Does this node exist?.....	12
10.	Using the XML table	13

So far, we have seen xml and inserted it into tables but we have not done anything useful with it. To do anything with the contents of the xml values, we need to query the data and that requires

- some methods to get to the data (Extract, ExtractValue, ExistsNode).
- a knowledge of how to write a path telling the system how to get to the part of the data we want.
- writing tests along that path.

This document is the crux of this unit and it takes some time to understand- this is a different way to think of data. Some of the output will be from SQL*Plus and some from SQL Developer.

1. Querying the XML data

XQuery is a language used to query XML data. XQuery is part of the W3C standards.

Oracle supports techniques that provide similar functionality.

XPath is part of XQuery and is used in querying XML data. When you use traditional relational SQL queries, you access data by using the tableName.ColumnName syntax (such as vt_animals.an_type). This SQL query syntax is designed for two-dimensional table access.

XML data is tree-structured so you access data values by specifying the path through the tree to the data. XML data is structured differently than relational data so the access paths will be different.

Node: An XML document is composed of nodes organized in a hierarchical/tree fashion. With the XML documents we are using only some types of nodes.

- The entire document is a document node
- Every XML element is an element node
- The text in the XML elements are text nodes
- Every attribute is an attribute node

The plan for this document:

- We are going to start by looking at a few simple paths through our xml document..
- We will use the path with the Extract()function.
- Then we will look at various paths to get to the data we want.
- After that, we will add predicates to the path; the predicates act as filters

Then we will look at two other functions

- ExtractValue() to get the scalar data out of the XML document
- ExistsNode() which we use as a test for data.

2. Simple XPath expressions

The paths used by XML are similar to the paths used in hierarchical file systems. We start at the root and navigate down the nodes.

The symbol `/` has two meanings. It can identify the root of the XML tree. It is also used as a path separator.

The symbol `//` refers to all descendants of the current node.

The symbol `.` refers to the current node in the tree structure.

The symbol `..` refers to going up one level in the tree structure.

The first function we will use is the `extract()` function. The function is passed an XML expression and an XPath string expression. The syntax is `extract(xmlData, 'path as a string literal')`

Demo 01: Here are several examples of queries that use the `extract` function (discussed in the next section) to illustrate several path expression with our tables. I am using a `Where` clause to get only one row from the table. I will not show the column headers for these.

Starts at the root; we get an XML document starting with the `animal` element.

```
select extract(anXmlData, '/animal') as "XMLData"
from XmlAnimals
where id = 1;
<animal><an_id>136</an_id><an_type>bird</an_type><an_name>ShowBoat</an_name><an_price>75</an_price><an_dob>2000-01-15</an_dob></animal>
1 row selected.
```

Since `animal` is the top level element, you could skip the leading `/` but it is probably clearer with it. This is considered a relative path.

```
select extract(anXmlData, 'animal') as "XMLData"
from XmlAnimals
where id = 1;
```

Now I walk down the path from `animal` to `an_name`. I get the XML elements for the animal's name.

```
select extract(anXmlData, '/animal/an_name') as "XMLData"
from XmlAnimals
where id = 1;
<an_name>ShowBoat</an_name>
1 row selected.
```

Demo 02: This uses the `//` symbol to go down the path until you get to the `categories` element. This book has one `categories` element with two `topic` subelements.

```
select extract(book_xml, '//categories') as "XMLData"
from XmlBooks
where id = 1;
<categories><topic>SQL</topic><topic>DB</topic></categories>
1 row selected.
```

What happens if I use //name? I got the author name - there are two authors.

```
select extract(book_xml,'//name') as "XMLData"
from XmlBooks
where id = 1;
<name><last>Collins</last><first>Joan</first></name><name><last>Effingham</last>
<first>Jill</first></name>
1 row selected.
```

This says to get the authors' last name.

```
select extract(book_xml,'book/authors/name/last') as "XMLData"
from XmlBooks
where id = 1;
<last>Collins</last><last>Effingham</last>
```

This also says to get the authors' last name because there is only one element in the xml with the element name last.

```
select extract(book_xml,'//last') as "XMLData"
from XmlBooks
where id = 1;
<last>Collins</last><last>Effingham</last>
```

Demo 03: data that is there and data that is not there

You can filter on the id column to get several rows.

```
select extract(book_xml,'//last') as "XMLData"
from XmlBooks
where id between 2 and 5;
<last>Malone</last><last>Effingham</last>
<last>Horn</last><last>Cocker</last><last>Shorter</last>
<last>LaVette</last>
<last>Del Ansom</last>

4 rows selected.
```

What if we look for topics? Compare these to the inserts for these rows.

```
select id, extract(book_xml,'//topic') as "XMLData"
from XmlBooks
where id between 4 and 7
order by id desc;
ID
XMLData
-----
7
<topic>Hist</topic>
6
<topic>Fiction</topic><topic>Travel</topic>
5
<topic/>
4
4 rows selected.
```

3. The extract Function

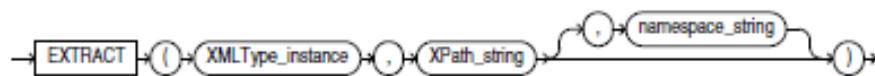
The first technique we will look at is the extract() function. You probably have a good feeling for this function by now. You provide an xml source- I have been using the xmldata column in the table and an XML path expression as a string literal' to the element you want returned as XML.

The previous set of demos often used the ID column in the table to get a single row- that was an integer column. If I do not have a Where filter, then I get all of the rows. The Select clause contains the function so that determines what is returned with each row.

Sometimes the row returned will be an empty result. The SQL function extract returns null if its XPath-expression argument targets no nodes. An error is never raised if no nodes are targeted.

Syntax

`extract_xml ::=`



Demo 04: We get one row in the result set for each row in the table.

```

select extract(book_xml,'//publisher') as "XMLData"
from XmlBooks;
<publisher>Addison Wesley</publisher>
<publisher>Addison Wesley</publisher>
<publisher>Addison Wesley</publisher>
<publisher>McGraw Hill</publisher>
<publisher>Addison Wesley</publisher>
<publisher>Addison Wesley</publisher>
<publisher>Addison Wesley</publisher>
<publisher>Jones Books</publisher>
<publisher>Addison Wesley</publisher>
<publisher>McGraw Hill</publisher>

10 rows selected.
  
```

Demo 05: Book id 4 had an empty price element and that is what we get. If that book had no price element, then that column would be empty. select id, substr(extract(book_xml,'//price'), 1,22) as "XMLData"

```

select id, extract(book_xml,'//price') as "XMLData"
from XmlBooks;


| ID | XMLData               |
|----|-----------------------|
| 1  | <price>29.95</price>  |
| 2  | <price>79.95</price>  |
| 3  | (null)                |
| 4  | <price>105.95</price> |
| 5  | <price>79.95</price>  |
| 6  | <price>29.50</price>  |
| 7  | <price/>              |
| 8  | <price>99.95</price>  |
| 9  | (null)                |
| 10 | (null)                |


```

4. Bad path expressions

Demo 06: If you use an XMLPath expression that does not work with your XML document, you get empty rows. You do not get an error. I am using a Where clause just to reduce the number of rows returned. These all return three empty rows.

```
select extract(book_xml,'/publisher/name') as "XMLData"
from XmlBooks
where id in (1,2,3);
```

3 rows selected.

```
select extract(book_xml,'/last') as "XMLData"
from XmlBooks
where id in (1,2,3);
```

```
select extract(book_xml,'//Last') as "XMLData"
from XmlBooks
where id in (1,2,3);
```

5. Including a predicate/test on the path

We can include a test on the path by including the test in square brackets. XML has a somewhat different syntax for writing tests; this is an xml test included in an sql query. The test is case specific.

The test is written inside square brackets [] and is written along the path. In the first example, we are testing the an_name which is on the animal path. The string literal is in double quotes.

The predicate becomes: an_name = "Ursula"] Check if the first an_name element value is the string "Ursula" In the books table, we can have multiple topics; we can use [1] as an index.

If the XPath-expression argument targets no nodes, a null is returned.

For not equals use !=; for inequality use > <

Demo 07:

We have one animal named Ursula.

```
select id, extract(anXmlData, '/animal[an_name = "Ursula"]') as "XMLData"
from XmlAnimals;
```

ID	XMLData
1	(null)
9	(null)
14	<animal><an_id>1201</an_id><an_type>cat</an_type><an_name>Ursula</an_name><an_price>500</an_price>...

We have no animal named Fluffy.

```
select id, extract(anXmlData, '/animal[an_name = "Fluffy"]') as "XMLData"
from XmlAnimals;
```

ID	XMLData
1	(null)
9	(null)
14	(null)

We have two animals with a price over 100.

```
select id, extract(anXmlData, '/animal[an_price > 100]') as "XMLData"
from XmlAnimals;
1
2 <animal><an_id>137</an_id><an_type>bird</an_type><an_name>Mr.
Peanut</an_name><an_price>150</an_price><an_dob>2008-01-12</an_dob></animal>
14
<animal><an_id>1201</an_id><an_type>cat</an_type><an_name>Ursula</an_name><an_price>500<
/an_price><an_dob>2007-12-15</an_dob></animal>
3 rows selected.
```

Demo 08: using an index

We have five books with a topic of SQL. We get 10 rows back because there is no Where clause.

```
select id, extract(book_xml,'//categories[topic= "SQL"]') as "XMLData"
from XmlBooks;
```

This is SQL Developer output.

ID	XMLData
1	<categories><topic>SQL</topic><topic>DB</topic></categories>
2	<categories><topic>SQL</topic><topic>DB</topic><topic>XML</topic></categories>
3	<categories><topic>C#</topic><topic>MySQL</topic><topic>SQL</topic><topic>DB</topic><topic>XML</topic></categories>
4 (null)	
5 (null)	
6 (null)	
7 (null)	
8	<categories><topic>SQL</topic><topic>Hist</topic></categories>
9 (null)	
10	<categories><topic>SQL</topic></categories>

We have three books with the **first** topic is SQL.

```
select id, extract(book_xml,'//categories[topic[1]= "SQL"]') as "XMLData"
from XmlBooks;
```

ID	XMLData
1	<categories><topic>SQL</topic><topic>DB</topic></categories>
2	<categories><topic>SQL</topic><topic>DB</topic><topic>XML</topic></categories>
3 (null)	
4 (null)	
5 (null)	
6 (null)	
7 (null)	
8	<categories><topic>SQL</topic><topic>Hist</topic></categories>
9 (null)	
10	<categories><topic>SQL</topic></categories>

We have no books with the **second** topic is SQL.

```
select id, extract(book_xml,'//categories[topic[2]= "SQL"]') as "XMLData"
from XmlBooks;
```

6. Coalesce and CTEs

Demo 09: Trying to use coalesce. Coalesce does not work if you try to use it to display an alternate value. The extract function returns xml, not scalars. And this gives the coalesce expression two inconsistent types.

```
select coalesce(extract(anXmlData, '//animal[an_price[1] > 100]'),
                 'no return value')
  from XmlAnimals;
```

SQL Error: ORA-00932: inconsistent datatypes: expected - got CHAR

You can use two xml values with coalesce. You would need to know that the element you just created makes sense in how this is used.

```
select coalesce(extract(anXmlData, '//animal[an_price[1] > 100]/an_name'),
                 xmlelement.createXML('<Empty_Element/>')) as "PricyAnimals"
  from XmlAnimals;
```

PricyAnimals
<Empty_Element/>
<an_name>Mr. Peanut</an_name>
<an_name>Ursula</an_name>

(NullIf does not work here.)

You can take the XMLtype expression and cast as a varchar2 and then use coalesce. I cast the xml as a shorter string; you would need to take care with any size limit depending on your xml size. This return is more than 100 characters long and would have been truncated at that length if I used varchar2(100)

```
select coalesce(
      cast(
        extract(anXmlData, '//animal[an_price[1] > 100]')
        as varchar2(200))
      , 'no return value') as "XMLData"
  from XmlAnimals;
```

```
no return value

<animal><an_id>137</an_id><an_type>bird</an_type><an_name>Mr.
Peanut</an_name><an_price>150</an_price><an_dob>2008-01-12</an_dob></animal>

<animal><an_id>1201</an_id><an_type>cat</an_type><an_name>Ursula</an_name><an_price>500</a
n_price><an_dob>2007-12-15</an_dob></animal>

3 rows selected.
```

Some string functions can work with xmlelement data.

Demo 10: Using CTEs. This is easier to see if you use a CTE to do the extract and then handle the format issues in the main query

```
with CTE as
  (select extract(anXmlData, '//animal[an_price[1] > 100]') as rtnValue
   from XmlAnimals)
select coalesce(cast(rtnValue as varchar2(200)), 'no return value') as
"XMLData"
  from CTE;
```

```

no return value
<animal><an_id>137</an_id><an_type>bird</an_type><an_name>Mr.
Peanut</an_name><an_price>150</an_price><an_dob>2008-01-12</an_dob></animal>
<animal><an_id>1201</an_id><an_type>cat</an_type><an_name>Ursula</an_name><an_price>500</a
n_price><an_dob>2007-12-15</an_dob></animal>

```

Demo 11: You also can use a CTE to filter more easily on the original return set; the CTE subquery is the query from a previous demo which returned 3 rows but only one matched the animal name.

```

With CTE as (
    select ID, extract(anXmlData, '//animal[an_name[1] = "Ursula"]') as XMLData
    from XmlAnimals
)
select ID, cast(XMLData as varchar2(200)) as "XMLData"
from CTE
where XMLData is not null
;

ID      XMLData
-----
14
<animal><an_id>1201</an_id><an_type>cat</an_type><an_name>Ursula</an_name><an_price>500
</an_price><an_dob>2007-12-15</an_dob></animal>

```

7. Wandering around the path after a test

Demo 12: This time we have a test along the path and then use .. to go back up the path one level to find the price. The syntax /.../ is walking up the tree just as it does in a file path

```

select id, extract(book_xml, '/book/categories[topic[1]= "SQL"]/../price ')
as "XMLData"
from XmlBooks
order by id;

```

ID	XMLData
1	<price>29.95</price>
2	<price>79.95</price>
3	(null)
4	(null)
5	(null)
6	(null)
7	(null)
8	<price>99.95</price>
9	(null)
10	(null)

Demo 13: Two tests along the path and then go up to get the book title

```

select id,
extract(book_xml, '/book[price[1] < 85]/categories[topic[1]="SQL"]/../title')
as title
from XmlBooks
order by id;

```

ID	TITLE
1	<title>SQL is Fun</title>
2	<title>Databases: an Introduction</title>
3	(null)
4	(null)
5	(null)
6	(null)
7	(null)
8	(null)
9	(null)
10	(null)

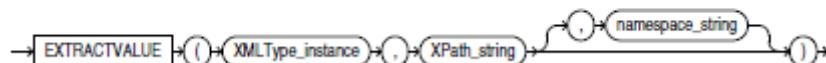
8. The ExtractValue Technique -- Getting inside the xmldata

Obviously we will want a way to get into the xml data and pull out the piece of data we want. We should be able to do that based on the tag names and specify that we want the contents. The function that we want is called ExtractValue; we give it two arguments- the first is the xmldtype instance and the second argument is a string representing the XPath expression to the value we want.

The result has to be a single node (use the [9] syntax).The node has to give you a scalar. That could be a text node or an element with a single text node. If the function call does not follow these rules, you get an error.

The return will be a varchar2 value unless you have an attached Schema (we don't have one).

Syntax



Demo 14: extractValue from xml

```

select      extractvalue(anXmlData, '/animal/an_name[1]')   as "AnimalName"
from        XmlAnimals
;

```

AnimalName

ShowBoat
Mr. Peanut
Ursula

Demo 15: I can return the an_dob and then cast it to a date value. The format returned by extractValue is the standard that XML uses. We can format it to the Oracle version.

```

select extractvalue(anXmlData, '//animal/an_dob') as "AnimalDOB "
, to_date(extractvalue(anXmlData, '//animal/an_dob'), 'yyyy-mm-dd') as
"AnimalDOB_Oracle"
from XmlAnimals;

```

AnimalDOB AnimalDob_Oracle

2000-01-15 15-JAN-00
2008-01-12 12-JAN-08
2007-12-15 15-DEC-07

Demo 16: The use of //topic and index lets us find the first two topics

```
select id
, extractvalue(book_xml,'(//topic)[1]') as "First topic"
, extractvalue(book_xml,'(//topic)[2]') as "Second topic"
from XmlBooks;
```

ID	First topic	Second topic
1 SQL	DB	
2 SQL	DB	
3 C#	MySQL	
4 (null)	(null)	
5 (null)	(null)	
6 Fiction	Travel	
7 Hist	(null)	
8 SQL	Hist	
9 Hist	(null)	
10 SQL	(null)	

Demo 17: The following gets up to four topics for each book. With this query you can use coalesce directly with the extractValue expression, because this is a scalar value, not an xml element

```
select id
, coalesce(extractvalue(book_xml,'(//topic)[1]'), 'no first topic') as "Topic 1"
, coalesce(extractvalue(book_xml,'(//topic)[2]'), 'no second topic') as "Topic 2"
, coalesce(extractvalue(book_xml,'(//topic)[3]'), 'no third topic') as "Topic 3"
, coalesce(extractvalue(book_xml,'(//topic)[4]'), 'no fourth topic') as "Topic 4"
from XmlBooks;
```

ID	Topic 1	Topic 2	Topic 3	Topic 4
1 SQL	DB		no third topic	no fourth topic
2 SQL	DB	XML		no fourth topic
3 C#	MySQL	SQL	DB	
4 no first topic	no second topic	no third topic	no fourth topic	
5 no first topic	no second topic	no third topic	no fourth topic	
6 Fiction	Travel		no third topic	no fourth topic
7 Hist		no second topic	no third topic	no fourth topic
8 SQL	Hist		no third topic	no fourth topic
9 Hist		no second topic	no third topic	no fourth topic
10 SQL		no second topic	no third topic	no fourth topic

Demo 18: We can use the extractValue method in the Where clause to filter on the contents of an xml value. Here we compare the scalar returned by the function ot a string literal. Which books are written by author Collins? I am using an index here also.

```
select extractvalue(book_xml,'(/book/title)[1]') as BookTitle,
       extractvalue(book_xml,'(//authors/name/first)[1]') as FirstName
  from XmlBooks
 where extractvalue(book_xml,'(//authors/name/last)[1]') = 'Collins' ;
```

BOOKTITLE	FIRSTNAME
SQL is Fun	Joan
The House Inside	Peter
Inside SQL Server	Peter

Demo 19: Once you extract the values of the data you can use regular functions and other techniques. What is the average price for books published by Addison Wesley?

```
select count(*) as "Addison Wesley Count"
, avg(extractvalue(book_xml,'(//price)')) as "AvgPrice"
from XmlBooks
where extractvalue(book_xml,'(//publisher)[1]') ='Addison Wesley' ;
Addison Wesley Count AvgPrice
----- -----
7      54.8375
```

Demo 20: Supposed you wanted the average price by publisher? That is a group by query. It probably makes sense to do this one step at a time and use CTE to work it out.

Step A get the data out; you can test this by itself; supply simple names for the columns (sql friendly column aliases)

```
select cast(extractvalue(book_xml,'(//publisher)[1]') as varchar2(25)) as Publ
, extractvalue(book_xml,'(//price)') as price
from XmlBooks;
```

Step B do the grouping

```
with ExtractedData as (
    select cast(extractvalue(book_xml,'(//publisher)[1]') as varchar2(25)) as Publ
    , extractvalue(book_xml,'(//price)') as price
    from XmlBooks
)
select Publ, count(*) as NumBk, avg(price) as avgPrice
From ExtractedData
group by Publ ;
```

Step C do any final formatting and supply user friendly column aliases ; You can combine steps B and C if there is no particular formatting to do in the last step

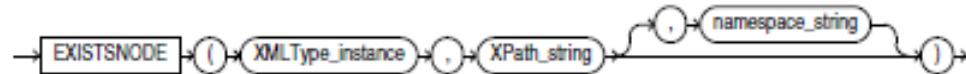
```
with ExtractedData as (
    select cast(extractvalue(book_xml,'(//publisher)[1]') as varchar2(25)) as Publ
    , extractvalue(book_xml,'(//price)') as price
    from XmlBooks
)
, GroupedData as (
    select Publ
    , count(*) as NumBk
    , avg(price) as avgPrice
    From ExtractedData
    group by Publ
)
select
    Publ      as "Publisher"
    , numBk    as "BookCount"
    , Round( Avgprice,2) as "AveragePrice"
from GroupedData
;
```

Publisher	BookCount	AveragePrice
McGraw Hill	2	105.95
Addison Wesley	7	54.84
Jones Books	1	99.95

9. The existsNode() Technique -Does this node exist?

existsNodes is a function with two parameters- an xmlType instance and an xPath string. The function returns a 0 or a 1. If there are no matches for the xPath then the function returns 0; if there is one or more matches, then the function returns 1. **existsNode is generally used in the Where clause of a query.**

Syntax



Demo 21: Using exists in the Select we get a return of 1 if there is a second author for a book

```
select id, existsNode(book_xml, '(/authors//last)[2]') as result
from XmlBooks;
```

ID	RESULT
1	1
2	1
3	1
4	0
5	0
6	1
7	1
8	0
9	0
10	1

Demo 22: Using exists in the Select to determine if a book price is 79.95

```
select id, existsNode(book_xml, '/book[price[1] = 79.95]') as Result
from XmlBooks;
```

ID	RESULT
1	0
2	1
3	0
4	0
5	1
6	0
7	0
8	0
9	0
10	0

Demo 23: You are more likely to use ExistsNode in the Where clause. The predicate is in the path.

```
select extract(book_xml, '(/book/title)[1]') as "XMLData"
from XmlBooks
where existsNode(book_xml, '/book[price[1] = 79.95]') = 1
;
```

```
XMLData
-----
<title>Databases: an Introduction</title>
<title>The truth about everything</title>
```

What do these find?

```
Select id, extract(book_xml, '//authors//last') as "XMLData"
From XmlBooks
where existsNode(book_xml, '(//authors//last)[2]') = 1;

Select id, extract(book_xml, '//authors//last') as "XMLData"
From XmlBooks
where existsNode(book_xml, '(//authors//last)[2]') = 0;
```

10. Using the XML table

We have a table XmlOnlyAnimals which is a pure xml table. We can use the same methods with the table.

Demo 24: extractvalue

```
select extractvalue(sys_nc_rowinfo$, '/animal/an_name') As BirdNames
from XmlOnlyAnimals
where extractvalue(sys_nc_rowinfo$, '/animal/an_type') = 'bird';
BIRDNAMES
-----
ShowBoat
Mr. Peanut
```

Demo 25: You can also use the word object_value instead of SYS_NC_ROWINFO\$, which will be easier to remember and easier to type.

```
select extractvalue(object_value, '/animal/an_name') As BirdNames
from XmlOnlyAnimals
where extractvalue(object_value, '/animal/an_type') = 'bird';
```

Table of Contents

1.	Set up the relationalZoo OR table.....	1
2.	Generating Elements and Attributes	2
3.	Using xmlElement	2
3.1.	Demos.....	2
3.2.	xmlElement syntax	3
3.3.	subelements	3
4.	Using xmlAttributes	4
5.	Using xmlForest	6
6.	Special issues	7
6.1.	Nulls.....	7
6.2.	DateTime values	8
7.	xmlQuery and FLWOR (OPTIONAL).....	8

1. Set up the relationalZoo OR table

I am going to create a table relationalZoo that is a regular table. I am naming the columns with a pattern that starts `zoo_` so that we can distinguish them from the xml tags. (Please create this table and follow along- that way you can try more experiments and have a model that works when it comes time to experiment.)

Demo 01: Create and populate relationalZoo

```
Drop table relationalZoo ;

create table relationalZoo (
    zoo_ID      number(5) primary key
, zoo_type    varchar2(25)
, zoo_name    varchar2(25)
, zoo_price   number(7,2)
, zoo_dob     date
);
```

Demo 02: This is a version of an insert statement that is useful for entering several rows of data with one statement.

```
insert into relationalZoo
    select 51, 'giraffe', 'Dora', 2000, '25-SEP-2015' from dual
union select 52, 'elephant', 'Elmer', 5000, '15-FEB-2000' from dual
union select 53, 'pig', 'Wilbur', 75.5, '18-FEB-2014' from dual
union select 54, 'elephant', 'Arnold', 4000, '28-OCT-2004' from dual
union select 55, 'eland', 'Elvira', 2000, '07-FEB-2014' from dual
union select 61, 'dragon', 'Susan', null, '29-FEB-1988' from dual
union select 62, 'dragon', null, 2000, null from dual
union select 63, 'dragon', 'Minerva', 2000, '29-FEB-1952' from dual
;
```

We are going to generate xml data from this table.

Do the following set commands to make the output easier to read.

```
set long 999999
set pagesize 999
set feedback 1
```

2. Generating Elements and Attributes

We want to take object-relational data and produce XML instances from that data. It is likely that you have a lot of object-relational data and may need to use it only occasionally as XML. (Oracle refers to its database now as ObjectRelational databases.)

3. Using xmlElement

We will start with the `xmlElement` method; this generates an XML element.

3.1. Demos

Demo 03: First a few demos; I am going to generate one XML element from one row in the table.

```
select xmlElement (an_name, zoo_Name) as "myXML"
from relationalZoo
where zoo_id = 55;
myXML
-----
<AN_NAME>Elvira</AN_NAME>
1 row selected.
```

What we can see here is that we gave the `xmlElement` function two arguments; the first was used to create the element name and was placed inside the start and end tags- which were correctly created with a start and an end tag. The second argument is the name of a column in the table and that supplied the content for the element.

We filtered for a single row and we got one row of well-formed XML returned in the result set. Note that we used this function inside a regular SQL Select statement and took the data from a regular Oracle table with varchar, number and date columns.

Demo 04: Since XML is case sensitive, we might need the element name to be in lower case. In that case, we quote it- that is the standard Oracle approach to things like column headers- use quotes to preserve case. This needs the double quote character.

```
select xmlElement ("an_name", zoo_Name) as "myXML"
from relationalZoo
where zoo_id = 55;
myXML
-----
<an_name>Elvira</an_name>
1 row selected.
```

Demo 05: selecting a row with a null; empty elements are still well formed XML

```
select xmlElement ("an_name", zoo_Name) as "myXML"
from relationalZoo
where zoo_id = 62;
myXML
-----
<an_name></an_name>
1 row selected.
```

Demo 06: filtering for multiple rows

```
select xmlelement ("an_name", zoo_Name) as "myXML"
from relationalZoo
where zoo_id in (51,52,53);
myXML
-----
<an_name>Dora</an_name>
<an_name>Elmer</an_name>
<an_name>Wilbur</an_name>
3 rows selected.
```

But we no longer have a single well-formed XML document. We do not have a single root element. You could describe this situation as missing a root element or as have a multi-rooted document. Each row taken by itself is well-formed XML.

These elements are sometimes called XML fragments because we may be thinking of them as part of an XML document.

3.2. xmlelement syntax

xmlelement is a function that returns an xmlelement instance. The output is not a string but SQL*Plus can display it as a string.

xmlelement has a required parameter which is the identifier for the element name. This becomes the root name of the xmlelement instance returned. Since XML documents are case specific, you will usually quote that argument to control the case of the element name.

There is an optional second parameter for attributes which we will get to shortly.

The last parameter, which can repeat, is an expression that makes up the element contents.

3.3. subelements

What we want to get is an xml fragment like this for each selected row from the table.

```
<animal>
  <an_id>51</an_id>
  <an_name>Dora</an_name>
  <an_cost>2000</an_cost>
</animal>
```

Thinking about this, we have the child elements (which look like something returned by xmlelement) nested inside the <animal> element which we can get from xmlelement.

Demo 07: This is the SQL to use- where the "third" parameter is repeated three times using an xmlelement function return value.

```
select xmlelement (
  "animal"
  , xmlelement ("an_id",      zoo_Id)
  , xmlelement ("an_name",    zoo_Name)
  , xmlelement ("an_cost",    zoo_Price)
) as "myXML"
from relationalZoo
where zoo_ID = 51
;
```

I am reformatting this by hand, adding line breaks, to make the result more obvious

```
myXML
-----
<animal>
  <an_id>51</an_id>
  <an_name>Dora</an_name>
```

```
<an_cost>2000</an_cost>
</animal>
1 row selected.
```

Demo 08: filtering for multiple rows – we get multiple xml fragments- this has three rows since there were 3 dragons in the relationalZoo table.

```
select xmlelement (
    "animal"
    , xmlelement ("an_id",      zoo_Id)
    , xmlelement ("an_name",    zoo_Name)
    , xmlelement ("an_cost",    zoo_Price)
) as "myXML"
from relationalZoo
where zoo_type = 'dragon';
myXML
-----
<animal><an_id>61</an_id><an_name>Susan</an_name><an_cost></an_cost></animal>
<animal><an_id>62</an_id><an_name></an_name><an_cost>2000</an_cost></animal>
<animal><an_id>63</an_id><an_name>Minerva</an_name><an_cost>2000</an_cost></animal>
3 rows selected.
```

The SQL for this may look complicated mostly because of the nested parentheses. With any nested expression, build it one item at a time. We will see a better function for this in a moment.

4. Using **xmlAttributes**

Perhaps we were supposed to generate an XML document where the animalID was to be an attribute of the <animal> element rather than a subelement.

For this we can use the function `xmlAttributes`; this function can be used only inside an `xmlelement` function call and it has to be the second argument. Remember the third argument can repeat.

Demo 09: one attribute

```
select xmlelement("animal"
    , xmlAttributes(zoo_ID as "an_id") ) as "myXML"
from relationalZoo
where zoo_type in ('elephant')
;
myXML
-----
<animal an_id="52"></animal>
<animal an_id="54"></animal>
2 rows selected.
```

Demo 10: two attributes

```
select xmlelement("animal"
    , xmlAttributes(zoo_ID as "an_id", zoo_name as "an_name") ) as "myXML"
from relationalZoo
where zoo_type in ('elephant')
;
myXML
```

```
-----
<animal an_id="52" an_name="Elmer"></animal>
<animal an_id="54" an_name="Arnold"></animal>
```

The syntax for `xmlAttributes` uses the syntax (*columnName* as "*attribute_name*").

We can generate several attributes by including multiple arguments in the call to `xmlAttributes`. This differs from `xmlElement` where we repeated the function call for each subelement

Quote the attribute name to preserve the case. If you omit the alias clause, then the column name is used as the attribute and it defaults to uppercase. If a column is null, the corresponding attribute is not generated.

Demo 11: You may want to generate XML which includes both attributes and subelements.

```
select      xmlElement("animal"
,    xmlAttributes(zoo_ID as "an_id")
,    xmlElement ("an_name", zoo_Name)
,    xmlElement ("an_price", zoo_price)
) as "myXML"
from relationalZoo ;
myXML
-----
<animal an_id="51"><an_name>Dora</an_name><an_price>2000</an_price></animal>
<animal an_id="52"><an_name>Elmer</an_name><an_price>5000</an_price></animal>
<animal an_id="53"><an_name>Wilbur</an_name><an_price>75.5</an_price></animal>
<animal an_id="54"><an_name>Arnold</an_name><an_price>4000</an_price></animal>
<animal an_id="55"><an_name>Elvira</an_name><an_price>2000</an_price></animal>
<animal an_id="61"><an_name>Susan</an_name><an_price></an_price></animal>
<animal an_id="62"><an_name></an_name><an_price>2000</an_price></animal>
<animal an_id="63"><an_name>Minerva</an_name><an_price>2000</an_price></animal>

8 rows selected.
```

Demo 12: More subelements.- this is picking up all of the columns from relationalZoo.

```
select
  xmlElement("animal"
,    xmlAttributes(zoo_ID as "an_id")
,    xmlElement ("an_name", zoo_Name)
,    xmlElement ("an_price", zoo_price)
,    xmlElement ("an_type", zoo_type)
,    xmlElement ("an_dob", zoo_dob)
) as "myXML"
from relationalZoo
;
myXML
-----
<animal
an_id="51"><an_name>Dora</an_name><an_price>2000</an_price><an_type>giraffe</an_type><an_d
ob>2002-09-25</an_dob></animal>

<animal
an_id="52"><an_name>Elmer</an_name><an_price>5000</an_price><an_type>elephant</an_type><an
_dob>2000-02-15</an_dob></animal>
```

```

<animal
an_id="53"><an_name>Wilbur</an_name><an_price>75.5</an_price><an_type>pig</an_type><an_dob
>2004-02-18</an_dob></animal>

<animal
an_id="54"><an_name>Arnold</an_name><an_price>4000</an_price><an_type>elephant</an_type><a
n_dob>2004-10-28</an_dob></animal>

<animal
an_id="55"><an_name>Elvira</an_name><an_price>2000</an_price><an_type>eland</an_type><an_d
ob>2006-02-07</an_dob></animal>

<animal
an_id="61"><an_name>Susan</an_name><an_price></an_price><an_type>dragon</an_type><an_dob>1
988-02-29</an_dob></animal>

<animal
an_id="62"><an_name></an_name><an_price>2000</an_price><an_type>dragon</an_type><an_dob></
an_dob></animal>

<animal
an_id="63"><an_name>Minerva</an_name><an_price>2000</an_price><an_type>dragon</an_type><an
_dob>1952-02-29</an_dob></animal>

8 rows selected.

```

5. Using xmlForest

The method `xmlForest` is a function that produces multiple XML sub-elements; these are considered forests of XML elements. This simply makes it easier to do the above queries. There are a few subtle differences in the output of `xmlForest` and `xmlElement`.

With `xmlForest`, we specify the column expressions to be used as sub-elements along with an element name. This uses the "as *column alias*" syntax.

Notice the difference between `xmlForest` and `xmlElement` when we have null values (compare rows 61-63)

Demo 13: `xmlForest`.

```

select      xmlElement("animal"
                    , xmlAttributes(zoo_ID as "an_id")
                    , xmlForest (
                        zoo_Name    as "an_name"
                        , zoo_price  as "an_price"
                        , zoo_type   as "an_type"
                        , zoo_dob    as "an_dob"
                    )
                    ) as "myXML"
  from relationalZoo
;

```

```

myXML
-----
<animal an_id="51"><an_name>Dora</an_name><an_price>2000</an_price><an_type>gira
ffe</an_type><an_dob>2002-09-25</an_dob></animal>

<animal an_id="52"><an_name>Elmer</an_name><an_price>5000</an_price><an_type>ele
phant</an_type><an_dob>2000-02-15</an_dob></animal>

```

```

<animal an_id="53"><an_name>Wilbur</an_name><an_price>75.5</an_price><an_type>pig</an_type><an_dob>2004-02-18</an_dob></animal>

<animal an_id="54"><an_name>Arnold</an_name><an_price>4000</an_price><an_type>elephant</an_type><an_dob>2004-10-28</an_dob></animal>

<animal an_id="55"><an_name>Elvira</an_name><an_price>2000</an_price><an_type>el and</an_type><an_dob>2006-02-07</an_dob></animal>

<animal an_id="61"><an_name>Susan</an_name><an_type>dragon</an_type><an_dob>1988-02-29</an_dob></animal>

<animal an_id="62"><an_price>2000</an_price><an_type>dragon</an_type></animal>

<animal an_id="63"><an_name>Minerva</an_name><an_price>2000</an_price><an_type>dragon</an_type><an_dob>1952-02-29</an_dob></animal>

8 rows selected.

```

this is row 61 from the previous query

```

<animal
  an_id="61"><an_name>Susan</an_name><an_price></an_price><an_type>dragon</an_type><an_dob>1988-02-29</an_dob></animal>

```

this is row 61 from the current query- no price subelement

```

<animal an_id="61"><an_name>Susan</an_name><an_type>dragon</an_type><an_dob>1988-02-29</an_dob></animal>

```

and this is the insert values for that row

```
select 61, 'dragon', 'Susan', null, '29-FEB-1988' from dual
```

6. Special issues

6.1. Nulls

We have seen that when a value in the table is null, the default behaviour for an attribute is to skip the attribute and the default for a child element is to generate an empty element.

Demo 14: If you want to skip the empty subelements if the column is null, you can use a case expression. I also use an_cost as the tag.

```

select xmlelement (
  "animal"
  , xmlelement ("an_id", zoo_Id)
  , case when zoo_name is null then null
         else xmlelement ("an_name", zoo_Name) end
  , case when zoo_price is null then null
         else xmlelement ("an_cost", zoo_price) end
) as "myXML"
from relationalZoo
where zoo_type = 'dragon'
;

```

```
myXML
-----
<animal><an_id>61</an_id><an_name>Susan</an_name></animal>
<animal><an_id>62</an_id><an_cost>2000</an_cost></animal>
<animal><an_id>63</an_id><an_name>Minerva</an_name><an_cost>2000</an_cost></animal>

3 rows selected.
```

6.2. DateTime values

Demo 15: The format for date values matches the xml date standard format, not the Oracle default format.

```
select XMLElement("animal"
    , XMLElement("an_id", zoo_ID)
    , XMLElement("an_dob", zoo_DOB)
    ) as "myXMLOutput"
from relationalZoo
where zoo_Type='elephant'
;
myXMLOutput
-----
<animal><an_id>52</an_id><an_dob>2000-02-15</an_dob></animal>
<animal><an_id>54</an_id><an_dob>2004-10-28</an_dob></animal>

2 rows selected.
```

7. xmLQuery and FLWOR (OPTIONAL)

Suppose we have our relationalZoo table and also a relational table of animal food and the type of animal that eats them.

Demo 16: Create and insert and display joined result

```
create table animalFood (an_type varchar2(25), an_food varchar2(25));
insert into animalFood Values ('elephant', 'hay');
insert into animalFood Values ('eland', 'hay');
insert into animalFood Values ('dragon', 'gold');
insert into animalFood Values ('dragon', 'elephants');
insert into animalFood Values ('dragon', 'DragonChow');

select zoo_id, zoo_type, an_food
from relationalZoo
left join animalFood on relationalZoo.zoo_type = animalFood.an_type
;
```

ZOO_ID	ZOO_TYPE	AN_FOOD
54	elephant	hay
52	elephant	hay
55	eland	hay
63	dragon	gold
62	dragon	gold
61	dragon	gold
63	dragon	elephants
62	dragon	elephants

```

61 dragon           elephants
63 dragon           DragonChow
62 dragon           DragonChow
61 dragon           DragonChow
53 pig              DragonChow
51 giraffe          DragonChow

14 rows selected.

```

Demo 17: Working with XMLElement and XMLForrest with the joined tables. This works but notice that any animal which has three foods gets three rows. For example rows 4,7,10 are all for animal id 63.

```

select XMLElement("animal"
    , XMLElement("an_id", zoo_ID)
    , XMLElement("an_type", zoo_type
        , xmlForest(an_food as "an_food") )
    ) as "myXMLOutput"
from relationalZoo
left join animalFood on relationalZoo.zoo_type = animalFood.an_type;

```

myXMLOutput
1 <animal><an_id>54</an_id><an_type>elephant<an_food>hay</an_food></an_type></animal>
2 <animal><an_id>52</an_id><an_type>elephant<an_food>hay</an_food></an_type></animal>
3 <animal><an_id>55</an_id><an_type>eland<an_food>hay</an_food></an_type></animal>
4 <animal><an_id>63</an_id><an_type>dragon<an_food>gold</an_food></an_type></animal>
5 <animal><an_id>62</an_id><an_type>dragon<an_food>gold</an_food></an_type></animal>
6 <animal><an_id>61</an_id><an_type>dragon<an_food>gold</an_food></an_type></animal>
7 <animal><an_id>63</an_id><an_type>dragon<an_food>elephants</an_food></an_type></animal>
8 <animal><an_id>62</an_id><an_type>dragon<an_food>elephants</an_food></an_type></animal>
9 <animal><an_id>61</an_id><an_type>dragon<an_food>elephants</an_food></an_type></animal>
10 <animal><an_id>63</an_id><an_type>dragon<an_food>DragonChow</an_food></an_type></animal>
11 <animal><an_id>62</an_id><an_type>dragon<an_food>DragonChow</an_food></an_type></animal>
12 <animal><an_id>61</an_id><an_type>dragon<an_food>DragonChow</an_food></an_type></animal>
13 <animal><an_id>53</an_id><an_type>pig</an_type></animal>
14 <animal><an_id>51</an_id><an_type>giraffe</an_type></animal>

Demo 18: This is something called a FLWOR query. I will discuss it very briefly at the end of the document. This is a very powerful(although exasperating at times) of working with data to get XML.

```

select xmlQuery (
  '
    <animal_list>
    {
      for $zooanimal in ora:view ("RELATIONALZOO" )
      let $anName   := $zooanimal /ROW/ZOO_NAME/text()
        , $zooType := $zooanimal /ROW/ZOO_TYPE/text()
      order by $anName
      return
        <animal name="{{$anName }}">
          <type>{{$zooType}}</type>
            <menu>
              {for $food in ora:view ("ANIMALFOOD" )
              let $anFood   := $food /ROW/AN_FOOD/text()
                , $anType   := $food /ROW/AN_TYPE/text()
              where $zooType = $anType
              return
                <food>{{$anFood}}</food>
              }
            </menu>
        </animal>
    }
  </animal_list>
  ,
  returning content).extract('/**') as "myXML"
  from dual
';

```

This is the result. This is a single row which contains an XML document. Notice that Susan the dragon only shows up once and her menu includes three foods. I have reformatted this in the next section.

```

myXML
-----
<animal_list><animal name="Arnold"><type>elephant</type><menu><food>hay</food></menu></animal><animal name="Dora"><type>giraffe</type><menu></menu></animal><animal name="Elmer"><type>elephant</type><menu><food>hay</food></menu></animal><animal name="Elvira"><type>eland</type><menu><food>hay</food></menu></animal><animal name="Minerva"><type>dragon</type><menu><food>gold</food><food>elephants</food><food>DragonChow</food></menu></animal><animal name="Susan"><type>dragon</type><menu><food>gold</food><food>elephants</food><food>DragonChow</food></menu></animal><animal name=""><type>dragon</type><menu><food>gold</food><food>elephants</food><food>DragonChow</food></menu></animal>
```

```
</food></menu></animal></animal_list>

1 row selected.
```

This is the same result set formatted by adding line breaks and whitespace to make the structure clearer. Now we can see that Susan gets one element and the food items are nested within that element. Nicer.

```
<animal name="Susan">
  <type>dragon</type>
  <menu>
    <food>gold</food>
    <food>elephants</food>
    <food>DragonChow</food>
  </menu>
</animal>
```

```
myXML
-----
<animal_list>
  <animal name="Arnold">
    <type>elephant</type>
    <menu>
      <food>hay</food>
    </menu>
  </animal>
  <animal name="Dora">
    <type>giraffe</type>
    <menu></menu>
  </animal>
  <animal name="Elmer">
    <type>elephant</type>
    <menu>
      <food>hay</food>
    </menu>
  </animal>
  <animal name="Elvira">
    <type>eland</type>
    <menu>
      <food>hay</food>
    </menu>
  </animal>
  <animal name="Minerva">
    <type>dragon</type>
    <menu>
      <food>gold</food>
      <food>elephants</food>
      <food>DragonChow</food>
    </menu>
  </animal>
  <animal name="Susan">
    <type>dragon</type>
    <menu>
```

```
<food>gold</food>
<food>elephants</food>
<food>DragonChow</food>
</menu>
</animal>
<animal name="Wilbur">
  <type>pig</type>
  <menu></menu>
</animal>
<animal name="">
  <type>dragon</type>
  <menu>
    <food>gold</food>
    <food>elephants</food>
    <food>DragonChow</food>
  </menu>
</animal>
</animal_list>
```

1 row selected.

Extremely brief discussion of this.

1. `xmlQuery` is an SQL function.
2. `ora:view` is an Oracle function that transforms the data in the table into a series of XML document fragments. The argument to `ora:view` is a string literal which is the name of a table or view. Each XML document fragment is a `<ROW>` element for a row in the table.
3. We get back a single row for the table; the row is made up of a sequence of `<ROW>` elements.
4. The words `for` and `return` inside the quotes are XQuery keywords and must be in lower case letters.
5. The `for` clause loops over the XML document fragments provided by the `ora:view` function.
6. `$v_row` is a variable that gets one XML document fragment at a time.
7. The `return` clause returns each input element
8. The `{...}` syntax represents a loop

You may have seen this type of loop in another programming language:

For each crt in my_collection Loop ... End loop

For each element in my_array Loop ... End Loop

for rec_data in cur_data Loop ... End Loop

This uses a different syntax but we have the idea that we have a collection of data items and we will process them one at a time. We get a variable to refer to each data item, one after the other.

|select xmlQuery (

<<< this is an XQuery function

```

' <<< start of the XML structure
<animal_list>           <<< this starts to build the xml result.

{
    <<<< starts the first loop
    for $zooanimal in ora:view ("RELATIONALZOO" ) <<< this is a for loop
    let $anName := $zooanimal /ROW/ZOO_NAME/text() <<< let does
                                                assignments to variables- use $name
                                                this is getting the zoo_name from the relationalZoo table via the view
    , $zooType := $zooanimal /ROW/ZOO_TYPE/text() <<< same for animal type
    order by $anName
    return <<< this is what we build in XML from those values
        <animal name ="{$anName }"> <<< attribute
        <type>{$zooType}</type>      <<< subelement
        <menu>
            {for $food in ora:view ("ANIMALFOOD" ) <<< start of inner loop
            let $anFood := $food /ROW/AN_FOOD/text()
            , $anType := $food /ROW/AN_TYPE/text()
            where $zooType = $anType <<< this joins the "tables/loops"
            return
                <food>{$anFood}</food> <<< build more of the XML
            } <<< end of inner loop
        </menu>
    </animal>
} <<< ends the first loop
</animal_list>
' <<< end of the XML structure
returning content).extract('/*') as "myXML"
from dual
;

```

Aren't you glad the flwor part was optional. That is

F for

L let

W where

O order by

R return