The SQL*Plus Interface

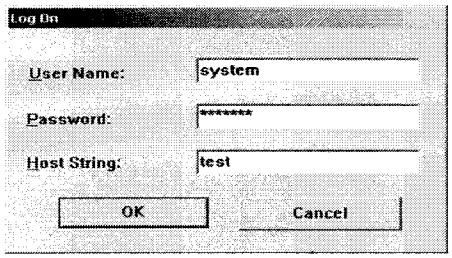
Red Rock Consulting

Logging On to SQL*Plus

- Can log into SQL*Plus either through the command line or through a GUI interface.
- To log on in either format will require a username and a password (unless the database administrator has configured it to accept Operating System authentication)
- You will also need to provide the name or alias of the database that you wish to connect to
- The tnsnames.ora file provides the connection information for clients to connect to remote databases

Red Rock Consulting

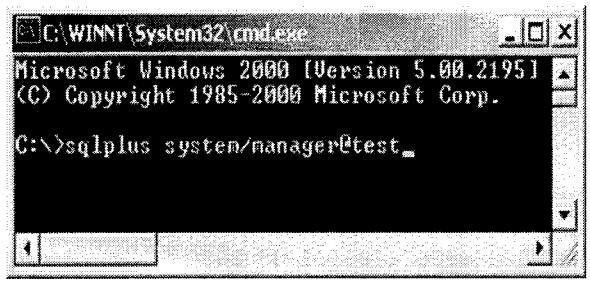
Logging On to SQL*Plus via the GUI



Red Rock Consulting

From the GUI interface you are prompted for username, password and host string where the host string is the name or alias of the database you wish to connect to.

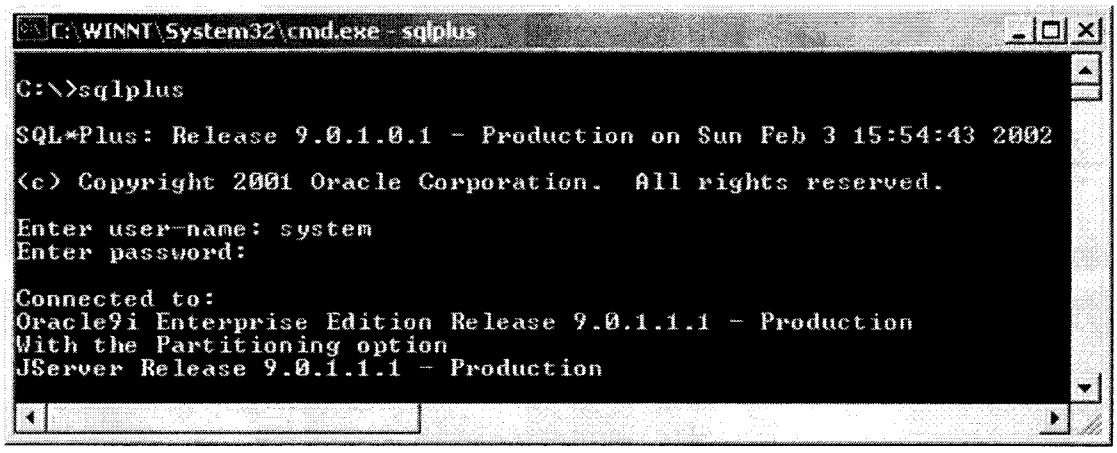
Logging On to SQL*Plus via the Command Line



Red Rock Consulting

Via the command line the SQL*Plus executable is invoked by typing *sqlplus*.

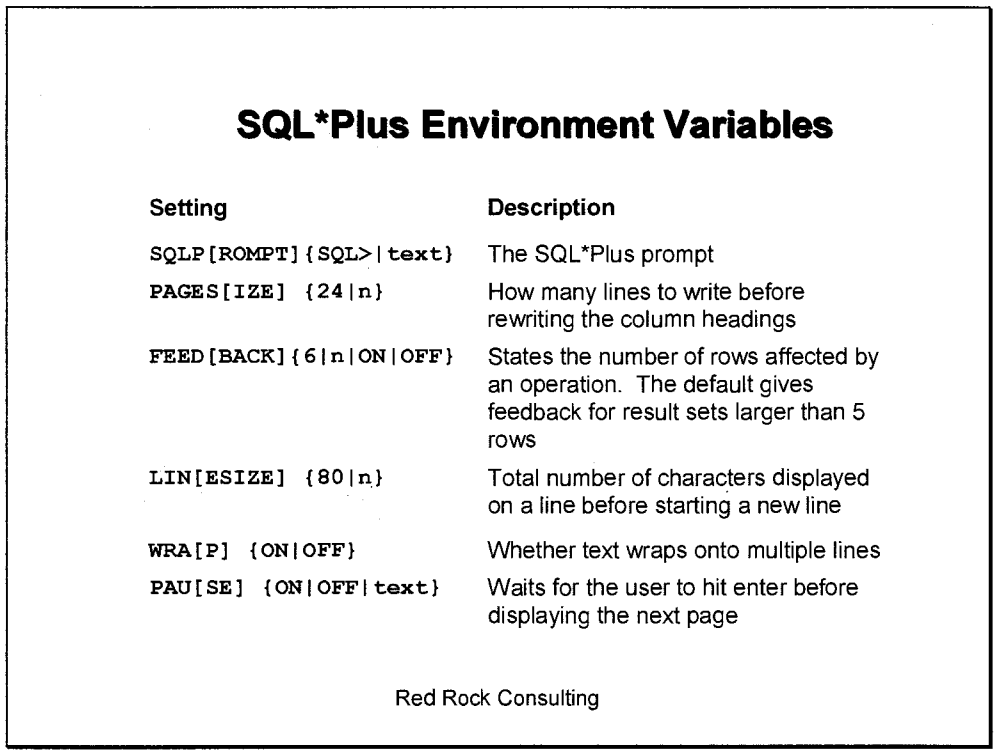
You can also pass in the username and password and the connect string (the name of the database that you are connecting to) on the same line, or if you just hit enter, you will be prompted for the user name and password. You will not be asked for the connect string, so if your environment is not pre-set to that database, you will find yourself unable to connect.



The SQL*Plus Environment

- It is possible to customise the SQL*Plus environment via the use of environment variables
- **SHOW ALL** will list all the current settings for the SQL*Plus environment. It will list each parameter and it's current setting
- **SHOW** *variable_name* will show the current setting of that environment variable
- The environment variables may also be set in the GUI environment by clicking on **OPTIONS** ? **ENVIRONMENT** from the menu

Red Rock Consulting



This is only a sample of some of the more useful environment variables. There are approximately sixty environment variables that may be set

A useful command is the **STORE** command. This command is used to store the current settings of your SQL*Plus session. For example, to store the settings of your current session in a file named **sess_env.txt** the command would be the following:

```
SQL> STORE SET sess_env.txt
```

Commands for the Management of Your SQL*Plus Session

Developers and administrators often find themselves connecting as various different users during a single SQL*Plus session. If you forget what user you are connected as the **SHOW USER** command will display what user you are connected as.

Another useful command is the **HOST** command. This allows you to change context to execute an Operating System command in the host environment without exiting the SQL*Plus session

Example

To copy a file from one location to another (very useful when writing backup scripts!) you could use the command
HOST cp *file_loc1 file_loc2*

Finally, two useful commands are **CLEAR SCREEN**, which erases the output on the screen, and **CLEAR BUFFER**, which deletes the contents of the buffer.

Oracle_Home

HKEY\HKLM\SOFTWARE\ORACLE

to get discoverer using the main TNS
put this into NET80\Admin\TNSNAMES.ora
IFILE = D:\oracle\Admin\tnsnames.ora
location of main TNS

ochnet.oracle.com/docs/products/oracle8i

tm.oracle.com/docs/content.html

NTH version
glogin has global settings
login has personal settings

Save file.sql
Ed file.sql

Select * from V$instance

SQL*Plus Buffer – File Management

- When you execute a SQL statement the statement is stored in the SQL*Plus buffer. The contents of the buffer are written to a file named **afiedt.buf** when you attempt editing the buffer
- You can edit the contents of the buffer by issuing the **ED[IT]**
- If you wish to edit a saved file, then the command **ED[IT]** ***filename*** will open the file in the default text editor defined for SQL*Plus
- To define a default text editor

```
SQL> DEFINE _editor=vi, "C:\Program
Files\Microsoft Office\Office\WINWORD.EXE"
```

Red Rock Consulting

The SQL*Plus Buffer only records the last SQL or PL/SQL statement made. It does not record any SQL*Plus environment settings.

For example, if you executed a simple **SELECT** statement and then you changed the setting for the **TIME** environment variable. If you executed the contents of the buffer by typing a slash (/), the **SELECT** statement would re-execute, not the environment setting.

```
SQL> SELECT * FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> SET TIME on
17:12:53 SQL> /

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

SQL*Plus Buffer – File Management

- **GET** *filename* - Retrieves Operating System file and places it in the buffer
- **SAVE** *filename* – Save the contents of the buffer in a named file
- **START** *filename* – Executes the contents of a file (i.e. runs a script)
- *@filename* – Same as **START** *filename*
- **SPOOL** *filename* – Spools the terminal output to a file
- **SPOOL OFF** – Terminates the spooling of output

Red Rock Consulting

Most of the commands listed may also be invoke via menu options in the GUI interface, however, it is often quicker to type the commands than to navigate with the mouse.

Slide 9

SQL*Plus Buffer – Editing Commands

Command	Description
`R[UN]` or `/`	Executes the contents of the buffer
`L[IST] [m] [n]`	Lists the contents of the buffer. When a number is passed in it lists one line (specified by n). When two numbers are passed in a range of lines (m to n) are listed
`DEL [m] [n]`	Deletes contents of the buffer. Behaves similarly to `LIST`
`I[NPUT] text`	inserts after the current (*) line
`A[PPEND] text`	appends after the current (*) line

Red Rock Consulting

The use of the line editing commands in SQL*Plus tend to be more a matter of preference than necessity. Most environments support the use of the GUI version of SQL*Plus where the standard GUI editing options are also supported. (Like **CTRL+c** for copy and **CTRL+v** for pasting).

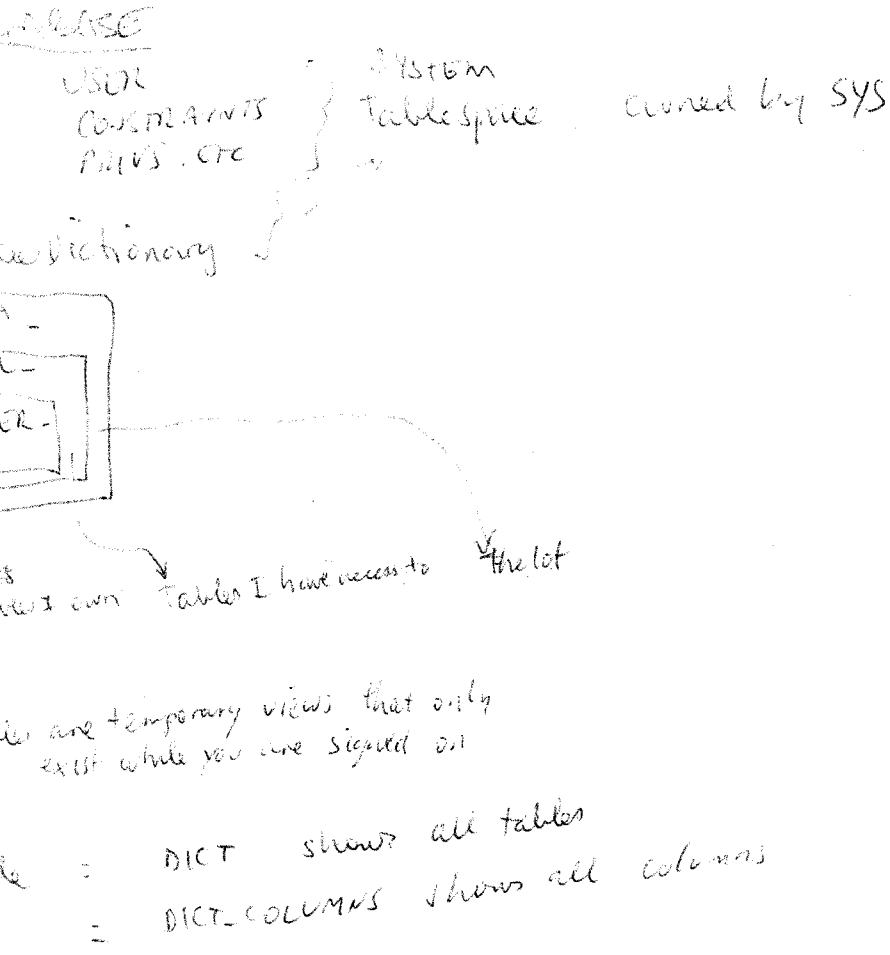
In the GUI SQL*Plus environment, a useful shortcut to copy text to the prompt, is to highlight the text you wish to copy with the left mouse button, and while still holding the left mouse button down, press the right mouse button and release. This will copy the selected text to the command prompt.

Listing the Columns in a Table

- One of the most useful SQL*Plus commands is the **DESCRIBE** command.
- **DESC[RIBE]** *[schema.]tablename*
- For example, issuing the **DESC dept** command would list the column names and datatypes of the **dept** table

```
SQL> DESC dept
 Name                      Null?    Type
 ------------------------- -------- ------------
 DEPTNO                             NUMBER(2)
 DNAME                              VARCHAR2(14)
 LOC                                VARCHAR2(13)
```

Red Rock Consulting



Summary

- SQL*Plus can be customised by the use of environment variables
- SQL*Plus stores the last SQL (or PL/SQL) statement in the buffer
- The contents of the buffer can be executed by typing a slash (/)
- You can save scripts in files and execute those scripts via SQL*Plus commands

Red Rock Consulting

Viewing Data – The SELECT Statement

Red Rock Consulting

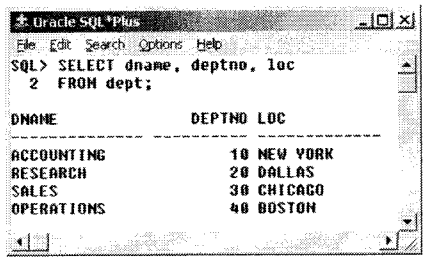
SQL Statements

- SQL operates primarily against database tables. Sometimes SQL statements operate against views and sequences
- Statements are case insensitive, data is not
- It is good practice to separate each clause onto separate line or indent

Red Rock Consulting

The SELECT Statement

- A Query – consists of **SELECT** statements. A basic query only views the information in the database, it does not modify it. Below is an example of a simple select statement and it's output.



Red Rock Consulting

show user

select user from dual

sysdate

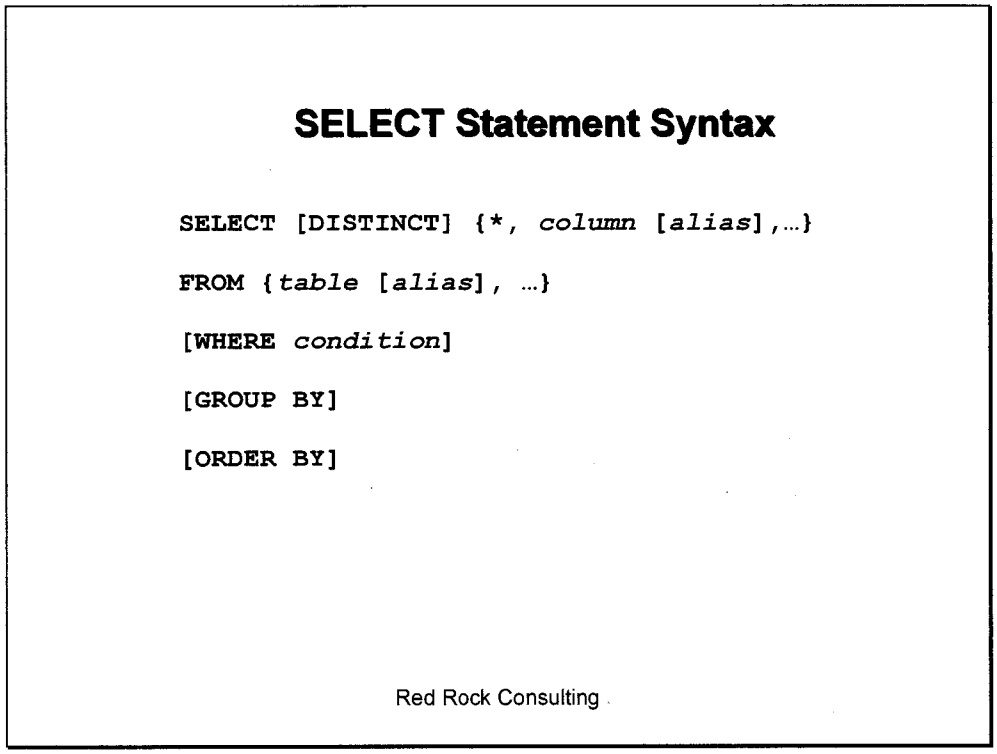
select table_name from user_tables;

select object_name from user_objects;

select * from emp where deptno = 10;

~~select ename||' works in '||deptno||' as '||job from emp;~~

select ename||' works in department '||deptno||' as a '||lower(job) from emp;



This is an abbreviated form of the **SELECT** statement syntax. For the full syntax see the SQL Reference of the Oracle Documentation

SELECT – Mandatory Clauses

- A **SELECT** query is read-only when not used in conjunction with other commands such as **INSERTs**, **UPDATEs** and **DELETEs**
- Two mandatory clauses – **SELECT** and **FROM**
  - **SELECT** - keyword identifying column list
  - **FROM** - keyword identifying data source(s)
- Column list - can include functions and derived values (e.g. **sal * 12**)
- Data sources are tables, views, sequences
- Data sets, single records or 0 records returned

Red Rock Consulting

The column list of the **SELECT** statement may have derived values. For example, if you wished to calculate the yearly salary of all the employees in the **emp** table, you would need to multiply the **sal** column by 12. In the following example output the derived column, **sal*12,** has also been given what is known as a column alias. This is a way of renaming the column when it outputs, so that the column title can be something more meaningful than sometimes obscure column names.

Example

```
SQL> SELECT ename, sal*12 "Yearly Salary"
  2  FROM emp;

ENAME      Yearly Salary
---------- -------------
SMITH               9600
ALLEN              19200
WARD               15000
JONES              35700
MARTIN             15000
BLAKE              34200
CLARK              29400
SCOTT              36000
KING               60000
TURNER             18000
ADAMS              13200
JAMES              11400
FORD               36000
MILLER             15600
```

There are three columns in the **dept** table. If you wished to view all the columns without typing in the column names individually, you could use the asterisk (*) in the column list of the **SELECT** statement. The asterisk simply means to select all columns.

Example

```
SQL> SELECT * FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

SELECT – Non Mandatory Clauses

- Non-mandatory clauses include:
  - **WHERE** - conditional statement (**AND**/**OR**)
  - **GROUP BY** - rollup statement
  - **HAVING** - optional clause with **GROUP BY** - like '**WHERE**' clause, restricts rows returned
  - **ORDER BY** - optional sort of column(s)

Red Rock Consulting

The **WHERE** clause allows you to restrict the quantity of data returned by specifying a condition. A **SELECT** statement with no **WHERE** clause will return an output for every row in the table. If you have a table with millions of rows this is an impractical way of viewing the data.

In the **emp** table, if you wished to view only the rows for employees who work in department 10, you could use a **WHERE** clause to restrict the output in the following manner:

```
SQL> SELECT ename, job, sal, deptno
  2  FROM emp
  3  WHERE deptno=10;

ENAME      JOB              SAL     DEPTNO
---------- --------- ---------- ----------
CLARK      MANAGER          2450         10
KING       PRESIDENT        5000         10
MILLER     CLERK            1300         10
```

It is not necessary to include the column(s) from the **WHERE** clause in the column list of the **SELECT** statement. In the previous example, if you know that you are viewing employees from department 10 it is superfluous to list the department number in the output.

The query could be changed to:

```
SQL> SELECT ename, job, sal
  2  FROM emp
  3  WHERE deptno=10;

ENAME      JOB              SAL
---------- --------- ----------
CLARK      MANAGER          2450
KING       PRESIDENT        5000
MILLER     CLERK            1300
```

If you needed the output sorted, for example, first by the employee name in descending order, and then by the job in ascending order, you can use the **ORDER BY** clause:

```
SQL> SELECT ename, job, sal
  2  FROM emp
  3  WHERE deptno=10
  4  ORDER BY ename DESC, job ASC;

ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK            1300
KING       PRESIDENT        5000
CLARK      MANAGER          2450
```

If you do not specify whether to sort in ascending order (**ASC**) or descending order (**DESC**) then the default of ascending order will be used.

Logical Operators

Operator	Description
AND	Returns TRUE if **all** conditions are TRUE
OR	Returns TRUE if **any** one of the conditions are TRUE
NOT	Returns TRUE if the condition is FALSE and vice versa

Red Rock Consulting

Example - NOT

```
SQL> SELECT ename
  2  FROM emp
  3  WHERE deptno NOT IN (10, 20)
  4  ORDER BY 1;

ENAME
----------
ALLEN
BLAKE
JAMES
MARTIN
TURNER
WARD
```

Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
!=	Not equal to

Red Rock Consulting

Comparison Operators

Operator	Meaning
`BETWEEN... AND ...`	Inclusive comparison between two values
`IN(list)`	Match a value in a list
`LIKE`	Match a character string
`IS NULL`	Is a null value

Red Rock Consulting

Example 1 - `BETWEEN`

```
SQL> SELECT ename, sal
  2  FROM emp
  3  WHERE sal BETWEEN 2000 AND 3000;

ENAME                SAL
---------- ----------
JONES              2975
BLAKE              2850
CLARK              2450
SCOTT              3000
FORD               3000
```

The above query could be equivalently written as:

```
SQL> SELECT ename, sal
  2  FROM emp
  3  WHERE sal >= 2000 AND sal <= 3000;
```

Example 2 – IN

```
SQL> SELECT ename
  2  FROM emp
  3  WHERE deptno IN (10, 20)
  4  ORDER BY 1;

ENAME
----------
ADAMS
CLARK
FORD
JONES
KING
MILLER
SCOTT
SMITH
```

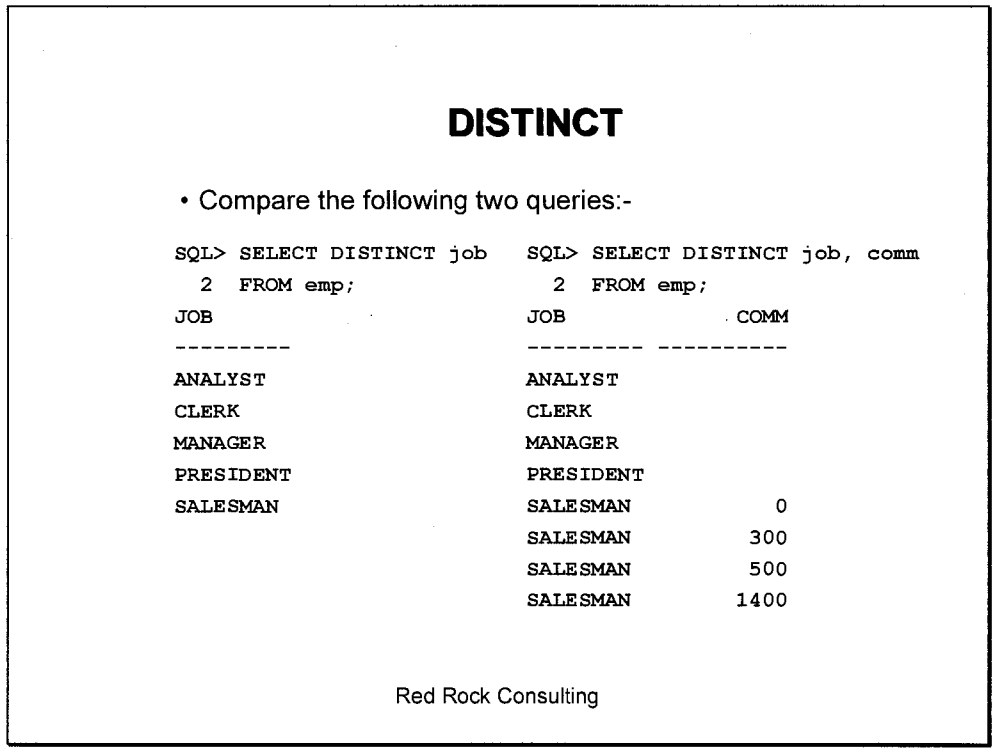
This query could be equivalently written by using a string of ORs instead of the IN operator. The IN operator is simply a shorthand way of comparing multiple OR conditions

```
SQL> SELECT ename
  2  FROM emp
  3  WHERE deptno = 10 OR deptno = 20
  4  ORDER BY 1;
```

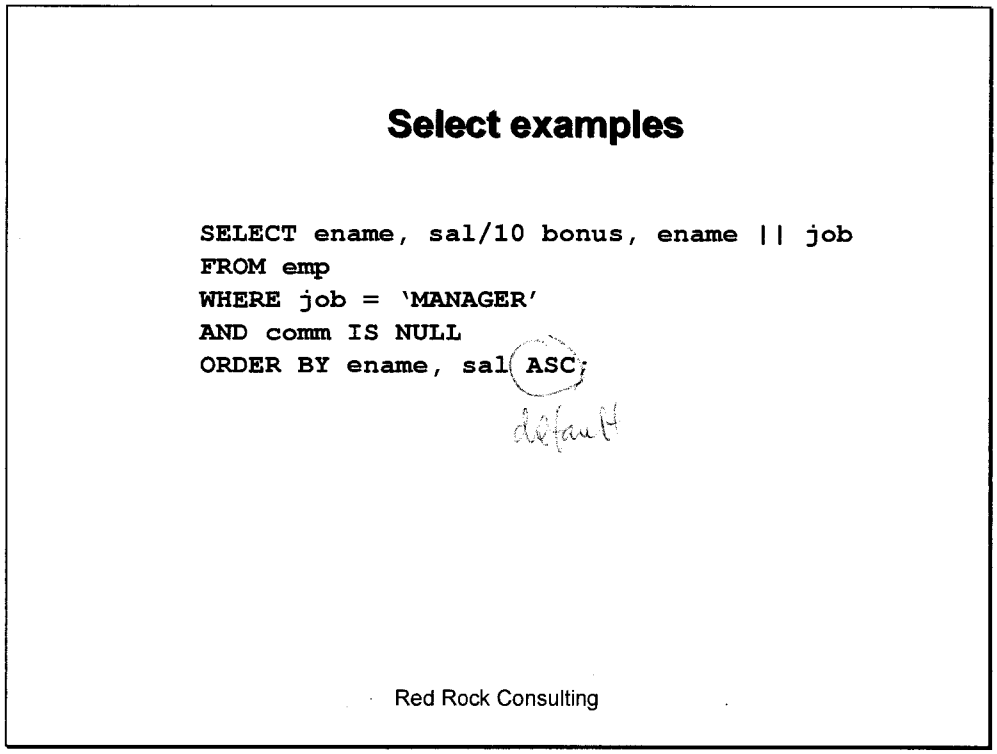
Example 3 – LIKE

The LIKE operator allows searches for matching string patterns. The percent sign (%) is a wildcard for 0 to many characters, while the underscore (_) is for a single character.

```
SQL> SELECT ename, mgr, hiredate
  2  FROM emp
  3  WHERE ename LIKE 'S%';

ENAME                MGR HIREDATE
---------- ---------- ---------
SMITH               7902 17/DEC/80
SCOTT               7566 19/APR/87
```



The **DISTINCT** clause shows only unique values for a given column or across several columns



In the first example the **ename** column is joined to the **job** column via the concatenation operator (a double pipe | |). You can concatenate many columns or strings of text together. When columns or strings have been concatenated together they are treated as a single column in the output of the query.

```
SQL> SELECT ename, sal/10 bonus, ename||job
  2  FROM emp
  3  WHERE job='MANAGER'
  4  AND comm IS NULL
  5  ORDER BY ename, sal;

ENAME            BONUS ENAME||JOB
---------- ---------- ------------------
BLAKE              285 BLAKEMANAGER
CLARK              245 CLARKMANAGER
JONES            297.5 JONESMANAGER
```

A useful capability of the concatenation operator is that it may be used to generate scripts

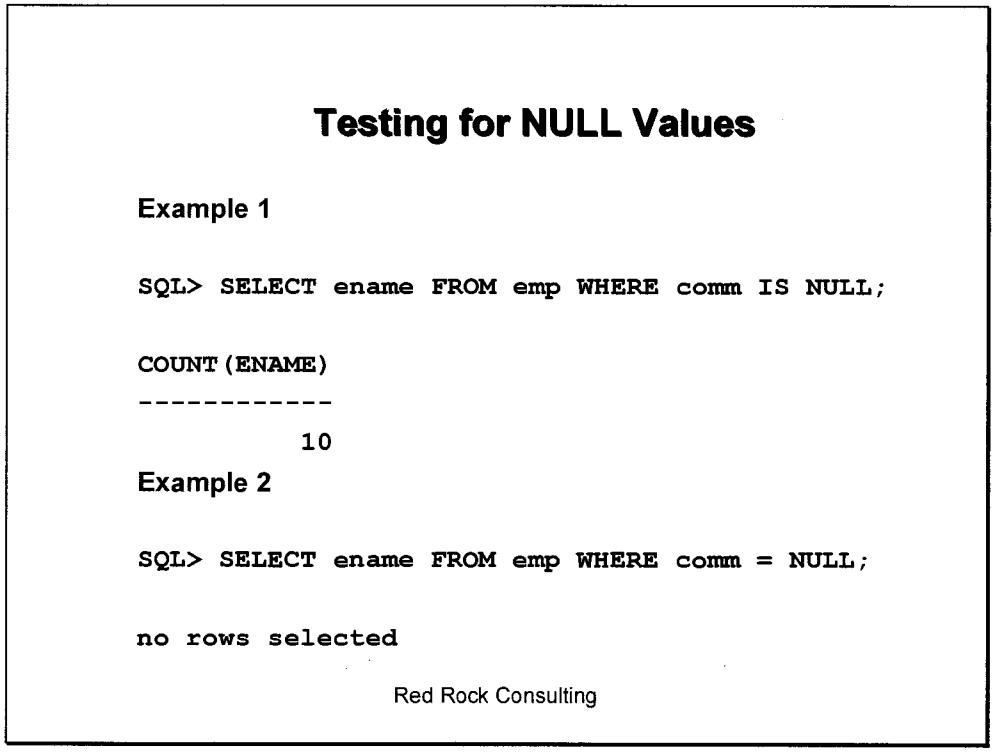
Example Script

```
/*
turn spooling on to place the output into a file that can
later be run as a script
*/
SPOOL compile.sql

/*
set the SQL*Plus environment variables not to spool
output other than the script that is being generated
*/
SET FEEDBACK OFF
SET ECHO OFF
SET HEADING OFF

/*
The SELECT statement generates a script to compile
packages that are invalid by concatenating text strings
together with the owner and name of the object
*/
SELELCT 'ALTER PACKAGE '||owner||'.'||object_name||'
COMPILE;'
FROM DBA_OBJECTS
WHERE object_type='PACKAGE' AND status='INVALID';

SPOOL OFF
-- turns the spooling off
```

Null Value concept

- **NULL** is an unknown or absence of a value
- It is not a blank, space or 0 and cannot be compared to anything, including **NULL**
- It does not take up any space but it can be used and checked for using the **IS NULL** operator
- A **NULL** in any arithmetic expression will return a **NULL**
  For example `10 * NULL = NULL`

Red Rock Consulting



The **IS NULL** operator is used to test for the existence of **NULL** values and returns a result of ten rows. In comparison, the second example uses the equal to (=) comparison operator and returns no rows. Any expression testing for **NULL** values with an arithmetic comparison operators like those listed on page 3-3-10 will always return no result, because any arithmetic expression with a **NULL** in it equates to **NULL**.

Summary

- The **SELECT** statement has two mandatory clauses; **SELECT** and **FROM**
- The **WHERE** predicate is used for restricting the data returned
- The output of a query can be sorted by the use of the **ORDER BY** clause
- A **NULL** in any arithmetic expression returns a **NULL**

Red Rock Consulting

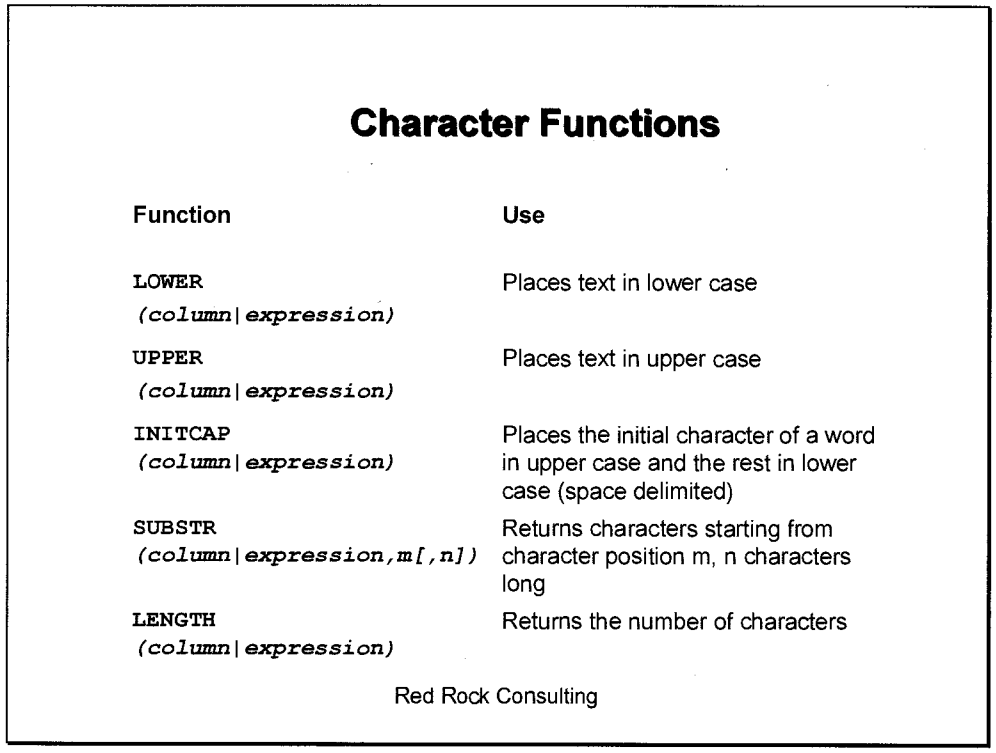
Built-In Functions

Red Rock Consulting

Built-In Functions

- Oracle provides many built-in functions for use with SQL and PL/SQL
- Single-row functions which return a row of output for every row of input. An example of a single row function is the **UPPER** function which places text in upper case
- Group functions which for multiple inputs give a single output. An example of a group function is the SUM function which can sum the values in a specified column
- Datatype conversion functions for switching datatypes
- Miscellaneous functions

Red Rock Consulting



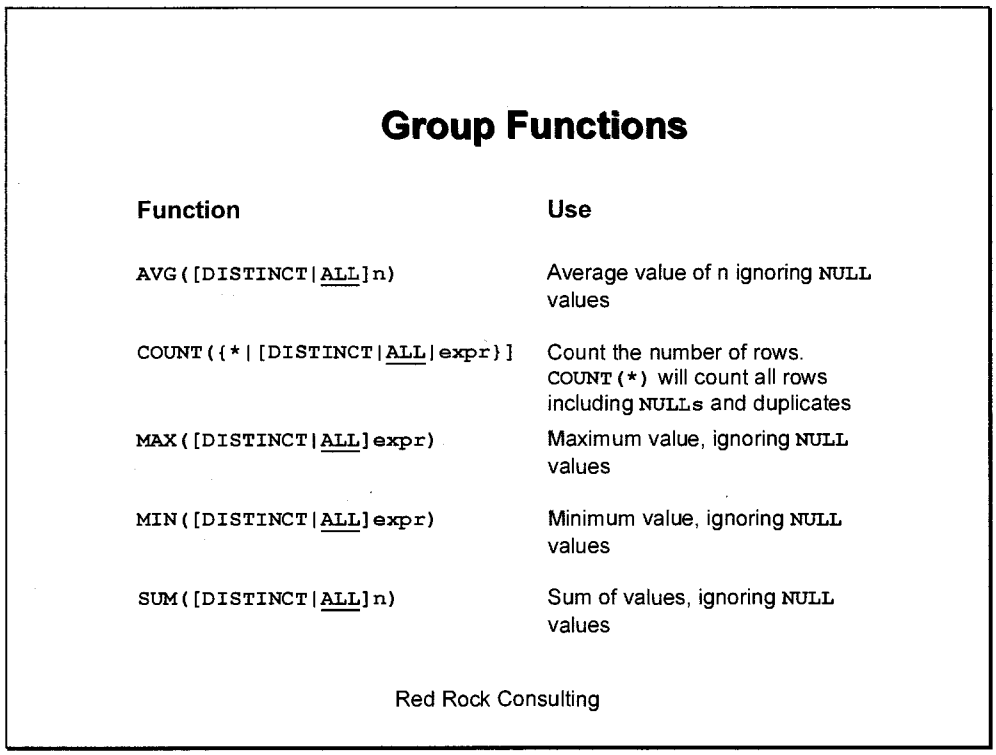
Character functions are single row functions because they have an output for every row of input

Example – `LOWER`

```
SQL> SELECT LOWER(dname) FROM dept;

LOWER(DNAME)
--------------
accounting
research
sales
operations
```

Example – `SUBSTR`

```
SQL> SELECT SUBSTR(dname,1,3) FROM dept;

SUB
---
ACC
RES
SAL
OPE
```



AVG and **SUM** are for numeric data only. **MAX** and **MIN** are for any datatype

Common Problems with Group Functions

- When using a group function in the column list, if any other columns in the list are not contained within a group function, then the statement will fail with an error message *unless* that column is also listed in the **GROUP BY** clause
- The **WHERE** clause will not accept a column with a group function. If you wish to restrict the data received with a group function use the **HAVING** clause.

Red Rock Consulting

Example – All Columns in Group Functions

```
SQL> SELECT AVG(comm),AVG(NVL(comm,0)),SUM(comm)/COUNT(*)
  2  FROM emp;

 AVG(COMM) AVG(NVL(COMM,0)) SUM(COMM)/COUNT(*)
---------- ---------------- ------------------
       550       157.142857          157.142857
```

The above query works because *all* the columns in the **SELECT** list are in group functions.

Example – Column Not in Group Function

The following query fails because the column **deptno** is in the **SELECT** list but there is no **GROUP BY** clause

```
SQL> SELECT SUM(sal), deptno
  2  FROM emp;
SELECT SUM(sal), deptno
                 *
ERROR at line 1:
ORA-00937: not a single-group group function
```

If a **GROUP BY** clause is added to the previous statement then the statement compiles correctly

```
SQL> SELECT SUM(sal), deptno
  2  FROM emp
  3  GROUP BY deptno;

  SUM(SAL)     DEPTNO
---------- ----------
      8750         10
     10875         20
      9400         30
```

Same as @CNT p 'sal','deptno' ',',1,1.

t ename, job, deptno from emp where ename like '%a%' and sal > 1000;

t ename from emp where ename like '%L%L%' and (deptno = 30 or mgr = 7782);

t ename, job, sal from emp where job in ('CLERK','ANALYST') and sal in (1000,3000,5

t max(sal), min(sal), sum(sal), avg(sal) from emp;

t max(sal), min(sal), sum(sal), avg(sal), job from emp group by job,

t count(job), job from emp group by job

t mgr, min(sal) from emp where mgr is not null group by mgr having min(sal) >= 1000

t d.dname, d.loc, count(e.deptno), avg(e.sal) from emp e, dept d
where d.deptno = e.deptno grouping d.dname, d.loc;

The HAVING Clause

```
SQL> SELECT SUM(sal), deptno
  2  FROM emp
  3  GROUP by deptno
  4  WHERE SUM(sal)>10000
  5  ORDER BY deptno DESC;
WHERE SUM(sal)>10000
*
ERROR at line 4:
ORA-00933: SQL command not properly ended
```

Red Rock Consulting

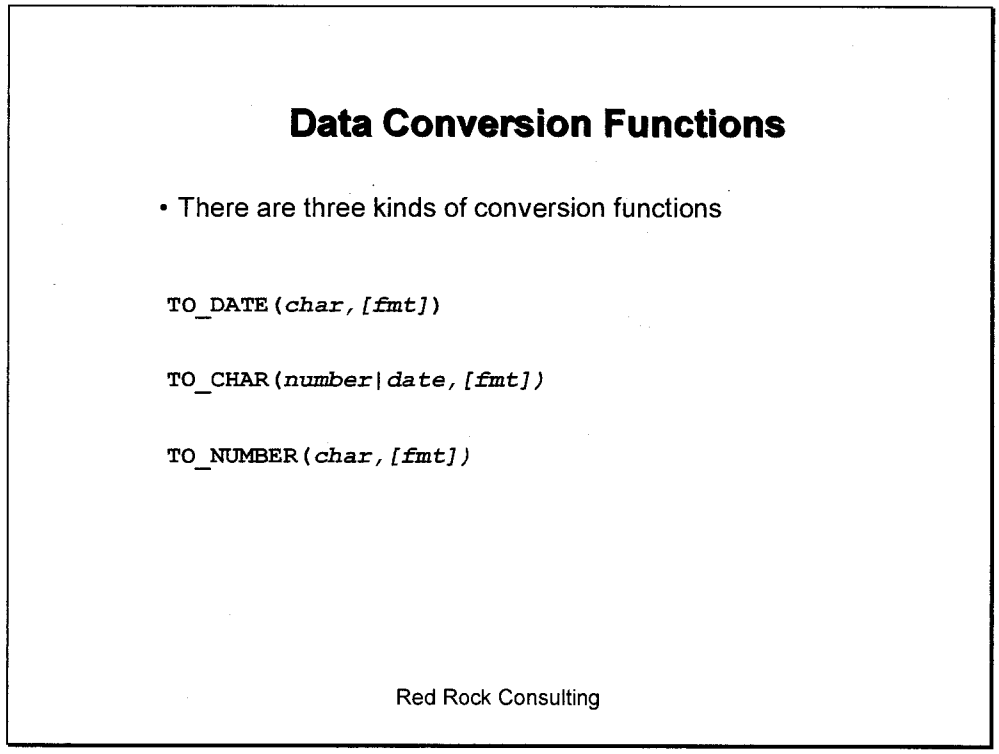
When trying to restrict the output with a group function in the **WHERE** clause the statement will fail.
Instead, place the group function in the **HAVING** clause as below:

```
SQL> SELECT SUM(sal), deptno
  2  FROM emp
  3  GROUP by deptno
  4  HAVING SUM(sal)>10000
  5  ORDER BY deptno DESC;

  SUM(SAL)     DEPTNO
---------- ----------
     10875         20
```

It is not necessary to have a group function in the **SELECT** list to use the **HAVING** clause:

HAVING is only used for group functions. It's the same as a where clause

```
SQL> SELECT job, sal
  2  FROM emp
  3  GROUP BY job, sal
  4  HAVING COUNT(*) > 1;

JOB              SAL
--------- ----------
ANALYST          3000
SALESMAN         1250
```



The '**fmt**' is the format mask for the datatype conversion. The following table shows some common elements:

Number Format Elements

Element	Description	Example	Result
9	Numeric position (number of 9s determine display width)	999999	1234
$	Floating dollar sign	$999999	$1234
L	Floating local currency symbol	L999999	FF1234
.	Decimal point in position specified	999999.99	1234.00
,	Comma in position specified	999,999	1,234
EEEE	Scientific notation (format must specify four Es)	99.999EEEE	1.234E+03
V	Multiply by 10 *n* times (*n* = number of 9s after V)	9999V99	123400
B	Display zero values as blank, not 0	B9999.99	1234.00

Date Format Elements

Element	Description
YYY or YY or Y	Last three, two, or one digits of year
Y,YYY	Year with comma in this position
IYYY,IYY,IY,I	Four, three, two, or one digit year based on the ISO standard
SYEAR or YEAR	Year spelled out; S prefixes BC date with -
Q	Quarter of year
MM	Month, two-digit value
MONTH	Name of month padded with blanks to length of nine characters
MON	Name of month, three-letter abbreviation
RM	Roman numeral month
WW or W	Week of year or month
DDD or DD or D	Day of year, month, or week
DAY	Name of day padded with blanks to length of 9 characters
DY	Name of day; three-letter abbreviation

XL syntax

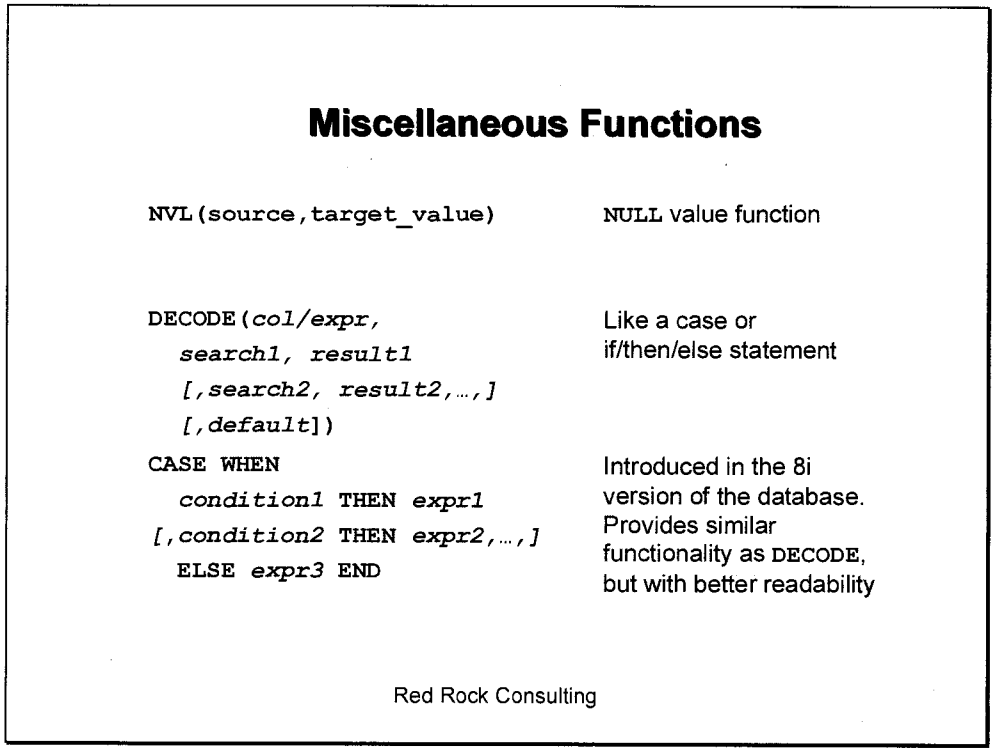
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYYMMDD';

Example – TO_CHAR with Dates

```
SQL> SELECT TO_CHAR(hiredate, 'Month Day Year HH24:MI')
"Hire Date"
  2  FROM emp;

Hire Date
-------------------------------------------------------
December  Wednesday Nineteen Eighty 00:00
February  Friday    Nineteen Eighty-One 00:00
February  Sunday    Nineteen Eighty-One 00:00
April     Thursday  Nineteen Eighty-One 00:00
September Monday    Nineteen Eighty-One 00:00
May       Friday    Nineteen Eighty-One 00:00
June      Tuesday   Nineteen Eighty-One 00:00
April     Sunday    Nineteen Eighty-Seven 00:00
November  Tuesday   Nineteen Eighty-One 00:00
September Tuesday   Nineteen Eighty-One 00:00
May       Saturday  Nineteen Eighty-Seven 00:00
December  Thursday  Nineteen Eighty-One 00:00
December  Thursday  Nineteen Eighty-One 00:00
January   Saturday  Nineteen Eighty-Two 00:00
```

Example – TO_CHAR with Numbers

```
SQL> SELECT ename name, TO_CHAR(sal,'$99,999') salary
  2  FROM emp
  3  WHERE sal BETWEEN 0 AND 2000
  4  ORDER BY name;

NAME       SALARY
---------- --------
ADAMS        $1,100
ALLEN        $1,600
JAMES          $950
MARTIN       $1,250
MILLER       $1,300
SMITH          $800
TURNER       $1,500
WARD         $1,250
```



The **NULL** Value function, **NVL**, allows a specified value to be substituted where a **NULL** exists. This is particularly useful when performing calculations like **AVG**.

Compare the result of the following two **SELECT** statements, one using **NVL** and the other not:

Example 1

```
SQL> SELECT AVG(comm)
  2  FROM emp;

 AVG(COMM)
----------
       550
```

Example 2

```
SQL> SELECT AVG(NVL(comm,0))
  2  FROM emp;

AVG(NVL(COMM,0))
----------------
      157.142857
```

The results are different because different calculations are performed. There are four rows with non **NULL** values in the **emp** table.

In the first example, the sum of the commissions are divided by four – the other **NULL** values in the **comm** column are ignored.

In the second example, the sum of the commissions are divided by 14. A zero is substituted to every row in the **emp** table that has a **NULL** value in the **comm** column. This results in the sum of the commissions being divided by every row in the table.

Example – DECODE

```
SQL> SELECT DECODE(job, 'MANAGER', 'This is the boss',
  2                     'ANALYST', 'This is an analyst',
  3                     'This is a clerk')
  4  FROM emp
  5  WHERE deptno=20;

DECODE(JOB,'MANAGE
------------------
This is a clerk
This is the boss
This is an analyst
This is a clerk
This is an analyst
```

Translating the above **DECODE** statement into **IF/THEN/ELSE** pseudo code:

```
IF job='MANAGER' THEN
     Output 'This is the boss'
ELSIF job='ANALYST' THEN
     Output 'This is an analyst'
ELSE
     Output 'This is a clerk'
END IF
```

Indenting and aligning your **DECODE** statements, clause by clause, as in the above example, greatly enhances the readability of the code.

Table Joins

Red Rock Consulting

et sysdate as "Date" from dual; "new salary"
t empno, ename, sal, round(sal * 1.15), from emp;
.*0.15

et ename, nvl(TO_CHAR(comm), 'No commission') from emp;

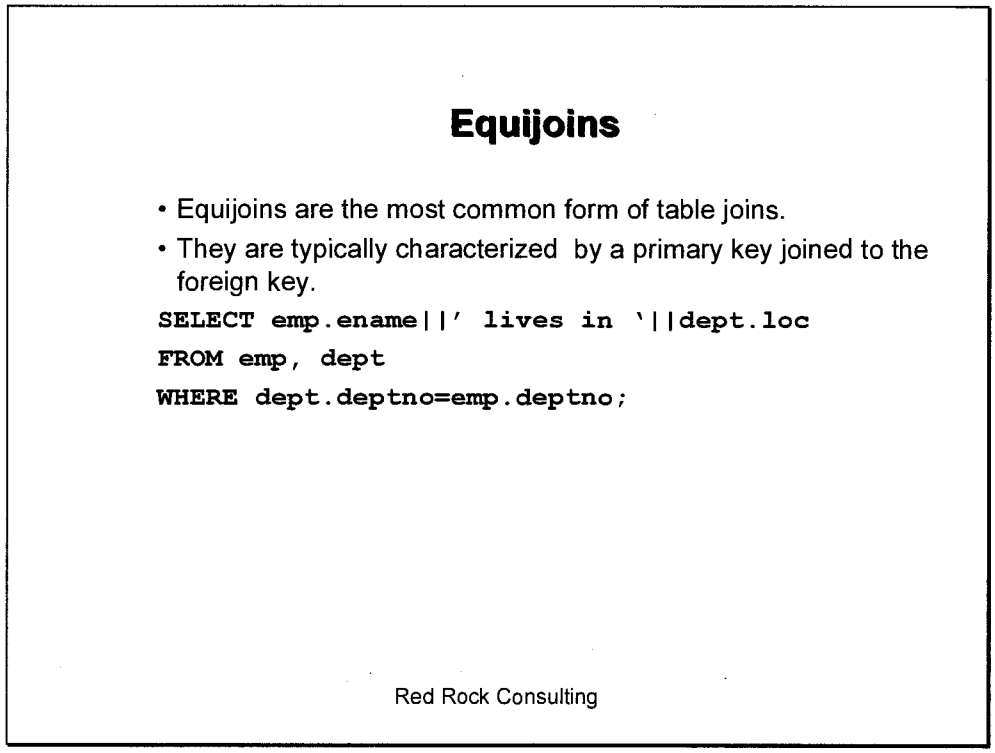
nd(exp, 2)
round to 2 dec places

nd(exp) round to nearest integer

et ename, hiredate, to_char(next_day(add_months(hiredate,6),2), 'DAY', the "
Spth" of "MONTH", YYYY) as "REVIEW" from emp;

add 6 months

then push to next monday (2nd day of week)



The column names are prefixed by the table names in the following query. It is only strictly necessary to prefix the column name by the table name when ambiguity could result from the same column name appearing in multiple tables. It is more efficient, however, to prefix the column name with the source table.

```
SQL> SELECT emp.ename||' lives in '||dept.loc
  2  FROM emp, dept
  3  WHERE dept.deptno=emp.deptno;

EMP.ENAME||'LIVESIN'||DEPT.LOC
--------------------------------
CLARK lives in NEW YORK
KING lives in NEW YORK
MILLER lives in NEW YORK
SMITH lives in DALLAS
ADAMS lives in DALLAS
FORD lives in DALLAS
SCOTT lives in DALLAS
JONES lives in DALLAS
ALLEN lives in CHICAGO
BLAKE lives in CHICAGO
MARTIN lives in CHICAGO
JAMES lives in CHICAGO
TURNER lives in CHICAGO
WARD lives in CHICAGO
```

```
SELECT d.dname, e.ename, e.mgr
FROM emp e, dept d
WHERE d.deptno=e.deptno
```

Table Aliases

- Table aliases appear in **FROM** clause and reduce typing (and typos)
- You simply list the alias that you wish to use for the table next to the table name in the **FROM** clause

```
SELECT e.ename, d.dname, e.sal * .1
bonus
FROM emp e, dept d
WHERE e.deptno = d.deptno;
```

Red Rock Consulting

Outer Joins

- An outer join is used when there is data on one side of the join and a **NULL** on the other
- For example, the **dept** table has 4 departments listed, but no-one in the **emp** table works for department 40. To write a query which selects from both tables and displays department 40, an outer join is required
- An outer join is characterised by a plus sign in parenthesis **(+)**. This symbol is place on the side where the **NULL** exists and tells the compiler to show that row anyway

Red Rock Consulting

Outer Join Example

```
SQL> SELECT e.ename, d.deptno, d.dname
  2  FROM emp e, dept d
  3  WHERE e.deptno(+)=d.deptno
  4  AND d.deptno NOT IN(10,20);

ENAME          DEPTNO DNAME
---------- ---------- --------------
ALLEN              30 SALES
BLAKE              30 SALES
MARTIN             30 SALES
JAMES              30 SALES
TURNER             30 SALES
WARD               30 SALES
                   40 OPERATIONS
```

Red Rock Consulting

Inner Join

- An inner join is when you join a table to itself
- This is sometimes required when a table has a 'self-referencing foreign key'. This is when a foreign key references a primary key in the same table
- Can often bypass use of an inner join by using a sub-query

Red Rock Consulting

Inner Join Example

```
SELECT boss.ename||' is '||worker.ename||'''s manager' MANAGERS
FROM emp boss, emp worker
WHERE worker.mgr=boss.empno;
```

```
MANAGERS
---------------------------------
JONES is SCOTT's manager
JONES is FORD's manager
BLAKE is ALLEN's manager
BLAKE is WARD's manager
```

Red Rock Consulting

Cartesian Product

- A Cartesian product is generated when no join condition is specified in the WHERE clause when selecting from multiple tables
- This generates a result set of all the rows in table1 multiplied by all the rows in table2
- A query involving a Cartesian product of the **emp** table, which has 14 rows, and the **dept** table, which has 4 rows, generates a result set of **4 x 14 = 56 rows**
- A Cartesian product is very inefficient and is usually the result of a mistake in the code

Red Rock Consulting

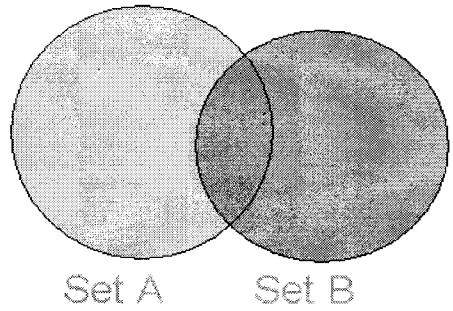
Cartesian Product Example

```
SQL> SELECT e.ename, e.job, d.dname
  2  FROM emp e, dept d;

ENAME      JOB        DNAME
---------- ---------- --------------
SMITH      CLERK      ACCOUNTING
ALLEN      SALESMAN   ACCOUNTING
WARD       SALESMAN   ACCOUNTING
JONES      MANAGER    ACCOUNTING
MARTIN     SALESMAN   ACCOUNTING
. . .

56 rows selected.
```

Red Rock Consulting

UNION

- the **UNION** operator joins the result set of two queries:
  SET A + SET B



Red Rock Consulting

UNION Example

```
SELECT deptno FROM emp
UNION
SELECT deptno FROM dept;

SELECT deptno FROM emp
UNION ALL
SELECT deptno FROM dept;
```

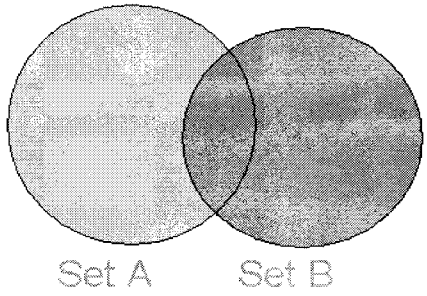
Red Rock Consulting

select ename, emp.deptno, dname from emp, dept where emp.deptno = dept.dept
select distinct(job) dname from emp, dept where emp.deptno = dept.deptno and emp.dep
select ename, dname, loc from emp, dept where emp.deptno = dept.deptno and comm is no
select ename, dname from emp, dept where emp.deptno = dept.deptno and ename like '%A%'

INTERSECT and MINUS

- The **INTERSECT** operator returns a result of the overlap of two queries
- The **MINUS** operator subtracts the result set of one query from another:

```
SET A - SET B
```



Red Rock Consulting

INTERSECT and MINUS Examples

```
SELECT deptno FROM dept
MINUS
SELECT deptno FROM emp;

SELECT deptno FROM emp
INTERSECT
SELECT deptno FROM dept;
```

Red Rock Consulting

Summary

- Queries can select from multiple tables
- The most frequently used form of join is the equijoin, where a pk=fk relationship is established in the **WHERE** clause
- Tables may also be joined using inner joins, outer joins and Cartesian product
- The **UNION**, **INTERSECT** and **MINUS** operators allow you to manipulate the results sets of two or more queries

Red Rock Consulting