Case and Character Manipulation
What Will I Learn?

Objectives

In this lesson, you will learn to:

- Select and apply single-row functions that perform case conversion and/or character manipulation
- Select and apply character case-manipulation functions LOWER, UPPER, and INITCAP in a SQL query
- Select and apply character-manipulation functions CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, TRIM, and REPLACE in a SQL query
- Write flexible queries using substitution variables
Why Learn It?

Purpose

Have you ever thought about the different ways in which we present ourselves? We dress up for special occasions, dress casually for play, and put on uniforms for sports events and band concerts. Being able to change the way we look for different situations is important. How would you choose to present yourself for a job interview?
Why Learn It?

Purpose (continued)

Being able to change the way in which data is presented is important when dealing with data from a database. Most of the time in SQL, we need to change the way that data appears depending on the requirements of the task we are trying to accomplish.

In this lesson, you will learn several ways in which to transform data to fit a particular situation.
Tell Me / Show Me

DUAL Table

The DUAL table has one row called "X" and one column called "DUMMY." The DUAL table is used to create SELECT statements and execute functions not directly related to a specific database table. Queries using the DUAL table return one row as a result. DUAL can be useful to do calculations and also to evaluate expressions that are not derived from a table.

<table>
<thead>
<tr>
<th>DUMMY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
</tbody>
</table>
DUAL Table (continued)

DUAL will be used to learn many of the single-row functions. In this example the DUAL table is used to execute a SELECT statement that contains a calculation. As you can see the SELECT statement returns a value that does not exist in the DUAL table. The value returned is a result of the calculation executed.

```
SELECT (319/29) + 12
FROM DUAL;
```

| (319/29)+12 | 23 |
Tell Me / Show Me

Single-Row Character Functions

Single-row character functions are divided into two categories:

- Functions that convert the case of character strings
- Functions that can join, extract, show, find, pad, and trim character strings

Single-row functions can be used in the `SELECT`, `WHERE`, and `ORDER BY` clauses.
Tell Me / Show Me

Single-Row Character Functions (continued)

Case-manipulation functions are important because you may not always know in which case (upper, lower or mixed) the data is stored in the database. Case manipulation allows you to temporarily convert the database data to a case of your choosing. Mismatches between database case storage and query case requests are avoided.
Tell Me / Show Me

Case Manipulation Functions

Case-manipulation functions are used to convert from lower to upper or mixed case. These conversions can be used to format the output and can also be used to search for specific strings.

Case-manipulation functions can be used in most parts of a SQL statement.
Tell Me / Show Me

Case Manipulation Functions (continued)
Case-manipulation functions are often helpful when you are searching for data and you do not know whether the data you are looking for is in upper or lower case. From the point of view of the database, ‘V’ and ‘v’ are NOT the same character and as such, you need to search using the correct case.

`LOWER(column | expression)` converts alpha characters to lower-case.

```sql
SELECT title
FROM d_cds
WHERE LOWER(title) = 'carpe diem';
```
Case Manipulation Functions (continued)

UPPER(column | expression) converts alpha characters to upper-case.

```
SELECT title
FROM d_cds
WHERE UPPER(title) = 'CARPE DIEM';
```

INITCAP(column | expression) converts alpha character values to uppercase for the first letter of each word.

```
SELECT title
FROM d_cds
WHERE INITCAP(title) = 'Carpe Diem';
```
Character Manipulation Functions (continued)

Character-manipulation functions are used to extract, change, format, or alter in some way a character string. One or more characters or words are passed into the function and the function will then perform its functionality on the input character strings and return the changed, extracted, counted, or altered value.

**CONCAT**: Joins two values together.

**SUBSTR**: Extracts a string of a determined length.

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT('Hello', 'World')</td>
<td>HelloWorld</td>
</tr>
<tr>
<td>SUBSTR('HelloWorld', 1, 5)</td>
<td>Hello</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Character Manipulation Functions (continued)

- LENGTH: Shows the length of a string as a number value.
- INSTR: Finds the numeric position of a named character.
- LPAD: Pads the left side of a character, resulting in a right-justified value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH('HelloWorld')</td>
<td>10</td>
</tr>
<tr>
<td>INSTR('HelloWorld','W')</td>
<td>6</td>
</tr>
<tr>
<td>LPAD(salary, 10,'*')</td>
<td>24000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Character Manipulation Functions (continued)

- RPAD: Pads the right-hand side of a character, resulting in a left-justified value.
- TRIM: Removes all specified characters from either the beginning or the ending of a string. The syntax for the trim function is:

```
TRIM( [ leading | trailing | both ] [character(s) to be removed ] ] string to trim)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPAD(salary, 10, '*')</td>
<td>24000*****</td>
</tr>
<tr>
<td>TRIM('H' FROM 'HelloWorld')</td>
<td>elloWorld</td>
</tr>
</tbody>
</table>
Character Manipulation Functions (continued)

REPLACE: Replaces a sequence of characters in a string with another set of characters. The syntax for the REPLACE function is:

\[
\text{REPLACE (string1, string\_to\_replace, [replacement\_string] )}
\]

string1 is the string that will have characters replaced in it; string\_to\_replace is the string that will be searched for and taken out of string1; [replacement\_string] is the new string to be inserted in string1.

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
FROM DUAL;
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE('JACK and JUE', 'J', 'BL')</td>
<td>BLACK and BLUE</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Using Column Aliases With Functions

All functions operate on values that are in parentheses, and each function name denotes its purpose, which is helpful to remember when constructing a query. Often times a column alias is used to name a function. When a column alias is used, the column alias appears in the output instead of the actual function syntax.

In the following examples, the alias "User Name" has replaced the function syntax in the first query.

By default, the column name in a SELECT statement appears as the column heading. In the second query example, however, there is no column in the table for the results produced, so the query syntax is used instead.
Using Column Aliases With Functions (continued)

```
SELECT LOWER(last_name) || LOWER(SUBSTR(first_name,1,1))
AS "User Name"
FROM f_staffs;
```

<table>
<thead>
<tr>
<th>User Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>does</td>
</tr>
<tr>
<td>millerb</td>
</tr>
<tr>
<td>tuttlem</td>
</tr>
</tbody>
</table>

```
SELECT LOWER(last_name) || LOWER(SUBSTR(first_name,1,1))
FROM f_staffs;
```

| LOWER (last_name) || LOWER(SUBSTR(first_name,1,1)) |
|-------------------|-------------------------------|
| does              |
| millerb           |
| tuttlem           |
Substitution Variables

Occasionally you may need to run the same query with many different values to get different result sets. Imagine for instance if you had to write a report of employees and their departments, but the query must only return data for one department at a time. Without the use of substitution variables, this request would mean you would have to continually edit the same statement to change the WHERE-clause.

Luckily for us, Oracle Application Express supports substitution variables. To use them, all you have to do is replace the hardcoded value in your statement with a :named_variable. Oracle Application Express will then ask you for a value when you execute your statement.
Tell Me / Show Me

Substitution Variables (continued)

So this original query:

```
SELECT first_name, last_name, salary, department_id
FROM    employees
WHERE department_id = 10 (and then 20, 30, 40...)
```

could be re-written to:

```
SELECT first_name, last_name, salary, department_id
FROM    employees
WHERE department_id = :dept_id
```

Note the use of : in front of dept_id. It is the colon that is the magic bit and makes Oracle Application Express accept the variable value.
Substitution Variables (continued)

Substitution variables are treated as character strings in Oracle Application Express, which means that when passing in character or date values you do not need the single quotation marks you would normally use to enclose the strings.

So a WHERE-clause would look like this

```
SELECT *  
FROM    employees  
WHERE last_name = :l_name  
```

When you click Run, a pop-up like the following is displayed by Oracle Application Express:
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Character functions
- CONCAT
- DUAL
- Expression
- INITCAP
- Format
- INSTR
- Input
- LENGTH
- LOWER
- LPAD
- Output
- REPLACE
- RPAD
- Single-row functions
- SUBSTR
- TRIM
- UPPER
- Substitution variable
Summary

Objectives Summarized

In this lesson you have learned to:

• Select and apply single-row functions that perform case conversion and/or character manipulation
• Select and apply character case-manipulation functions LOWER, UPPER, and INITCAP in a SQL query
• Select and apply character-manipulation functions CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, TRIM, and REPLACE in a SQL query
• Write flexible queries using substitution variables
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Number Functions
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Select and apply the single-row number functions ROUND, TRUNC, and MOD in a SQL query
• Distinguish between the results obtained when TRUNC is applied to a numeric value and ROUND is applied to a numeric value
• State the implications for business when applying TRUNC and ROUND to numeric values
Why Learn It?

Purpose

One of the reasons we put our money in a bank is to take advantage of the interest it accumulates over time. Banks adjust the interest rate with various economic indicators such as inflation and the stock market. Typically, interest rates are expressed as a percent such as 3.45%.
Why Learn It?

Purpose (continued)

What if a bank decided to round the percentage rate to 3.5%? Would it be to your advantage? What if they decided to just drop the decimal values and calculate the interest at 3%, would you be happy then?

Rounding and truncating numbers play an important part in business and in turn with the databases that support these businesses as they store and access numeric data.
Tell Me / Show Me

Number Functions

The three number functions are:

• ROUND
• TRUNC
• MOD
Tell Me / Show Me

ROUND

ROUND can be used with both numbers and dates. It is mainly used to round numbers to a specified number of decimal places, but it can also be used to round numbers to the left of the decimal point.

Syntax

ROUND(column|expression, decimal places)

Note that if the number of decimal places is not specified or is zero, the number will round to no decimal places.

ROUND(45.926)    46
ROUND(45.926, 0) 46
Tell Me / Show Me

ROUND (continued)

If the number of decimal places is a positive number, the number is rounded to that number of decimal places.

\[
\text{ROUND}(45.926, \ 2) \quad 45.93
\]

If the number of decimal places is a negative number, numbers to the left of the decimal are rounded.

\[
\text{ROUND}(45.926, \ -1) \quad 50
\]
Tell Me / Show Me

TRUNC
The TRUNC function can be used with both numbers and dates. It is mainly used to terminate the column, expression, or value to a specified number of decimal places. When TRUNC is used, if the number of decimal places is not specified, then the specified number defaults to zero.

Syntax

\[
\text{TRUNC}(\text{column}|\text{expression}, \text{decimal places})
\]

\[
\text{TRUNC} \ (45.926, \ 2) \quad 45.92
\]
Tell Me / Show Me

TRUNC (continued)

As with ROUND, if the TRUNC expression does not specify the number of decimal places or specifies a zero, the number is truncated to zero decimal places.

- TRUNC (45.926, 0) = 45
- TRUNC (45.926) = 45

Remember that TRUNC does not round the number. It simply terminates the number at a given point.
Tell Me / Show Me

MOD

The MOD function finds the remainder of one value divided by another value.

For example, the MOD of 5 divided by 2 = 1. MOD can be used to determine whether a value is odd or even. If you divide a value by 2 and there is no remainder, the number must be an even number. Using the MOD function with 2, as in 12/2 and there is no remainder, the number must have been an even number.
Tell Me / Show Me

MOD (continued)

SELECT MOD(1600, 500) FROM DUAL;
100 remainder

SELECT last_name, salary, MOD(salary, 2) As "Mod Demo"
FROM f_staffs
WHERE staff_type IN('Order Taker', 'Cook', 'Manager');

The "Mod Demo" column will show if the salary is an odd or even number.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Number functions
MOD
ROUND
TRUNC
Summary

Objectives Summarized

In this lesson you have learned to:

• Select and apply the single-row number functions ROUND, TRUNC, and MOD in a SQL query
• Distinguish between the results obtained when TRUNC is applied to a numeric value and ROUND is applied to a numeric value
• State the implications for business when applying TRUNC and ROUND to numeric values
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Date Functions
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Select and apply the single-row functions MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, and TRUNC that operate on date data
• Explain how date functions transform Oracle dates into date data or a numeric value
• Demonstrate proper use of the arithmetic operators with dates
• Demonstrate the use of SYSDATE and date functions
• State the implications for world businesses to be able to easily manipulate data stored in date format
Why Learn It?

Purpose
Have you ever wondered how many days remain in the school year or how many weeks there are until graduation? Because the Oracle database stores dates as numbers, it's easy to perform calculations on dates using addition and subtraction.

Businesses depend on being able to use date functions to schedule payrolls and payments, track employee performance reviews and years of service, or keep track of orders and shipments. All of these business needs are easily handled using simple SQL date functions.
Tell Me / Show Me

Displaying Dates
The default display format for dates is `DD-MON-RR` -- that is, `02-DEC-99`.

However, the Oracle database stores dates internally with a numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is `DD-MON-RR`. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D. This represents the range of dates that you can store successfully in an Oracle database.
Tell Me / Show Me

SYSDATE

When a record with a date column is inserted into a table, the century information is picked up from the SYSDATE function. SYSDATE is a date function that returns the current database server date and time.

SYSDATE

To display the current date, use the DUAL table.

```
SELECT SYSDATE
FROM DUAL;
```
DATE Data Type

The DATE data type always stores year information as a four-digit number internally: two digits for the century and two digits for the year. For example, the Oracle database stores the year as 1996 or 2004, not just as 96 or 04.

Although the internal storage keeps track of the complete date, when the date column is displayed on the screen, the century component is not displayed by default.
Date Functions

Tell Me / Show Me

Working with Dates

SELECT last_name, hire_date + 60
FROM employees;

SELECT last_name, (SYSDATE - hire_date)/7
FROM employees;

SELECT employee_id, (end_date - start_date)/365 AS "Tenure in last job"
FROM job_history;
Tell Me / Show Me

Date Functions

The date functions shown in the table operate on Oracle dates. All of the date functions return a value with a DATE data type except the MONTHS_BETWEEN function, which returns a numeric data type value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTHS_BETWEEN</td>
<td>Number of months between two dates</td>
</tr>
<tr>
<td>ADD_MONTHS</td>
<td>Add calendar months to date</td>
</tr>
<tr>
<td>NEXT_DAY</td>
<td>Next day of the date specified</td>
</tr>
<tr>
<td>LAST_DAY</td>
<td>Last day of the month</td>
</tr>
<tr>
<td>ROUND</td>
<td>Round date</td>
</tr>
<tr>
<td>TRUNC</td>
<td>Truncate date</td>
</tr>
</tbody>
</table>
Date Functions (continued)

The following query shows how the date functions are used.

```
SELECT employee_id, hire_date,
       ROUND(MONTHS_BETWEEN(SYSDATE, hire_date)) AS TENURE,
       ADD_MONTHS (hire_date, 6) AS REVIEW,
       NEXT_DAY(hire_date, 'FRIDAY'),
       LAST_DAY(hire_date)
FROM employees
WHERE MONTHS_BETWEEN (SYSDATE, hire_date) > 36;
```
Tell Me / Show Me

Date Functions (continued)

Here is another example of a query using multiple date functions.

```sql
SELECT employee_id, hire_date,
     ROUND(MONTHS_BETWEEN(SYSDATE, hire_date)) AS TENURE,
     ADD_MONTHS (hire_date, 6) AS REVIEW,
     NEXT_DAY(hire_date, 'FRIDAY'),
     LAST_DAY(hire_date)
FROM employees
WHERE MONTHS_BETWEEN (SYSDATE, hire_date) > 36;
```

The result set from this query returns 20 rows including:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>HIRE_DATE</th>
<th>TENURE</th>
<th>REVIEW</th>
<th>NEXT_DAY</th>
<th>LAST_DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>21-SEP-89</td>
<td>181</td>
<td>21-MAR-90</td>
<td>22-SEP-89</td>
<td>30-SEP-89</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Date Functions (continued)
Below are the results from queries using ROUND and TRUNC date functions with SYSDATE (assume `SYSDATE = '25-JUL-95'`).

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ROUND (SYSDATE, 'MONTH')</code></td>
<td>01-AUG-95</td>
</tr>
<tr>
<td><code>ROUND (SYSDATE, 'YEAR')</code></td>
<td>01-JAN-96</td>
</tr>
<tr>
<td><code>TRUNC (SYSDATE, 'MONTH')</code></td>
<td>01-JUL-95</td>
</tr>
<tr>
<td><code>TRUNC (SYSDATE, 'YEAR')</code></td>
<td>01-JAN-95</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

ADD_MONTHS
LAST_DAY
MONTHS_BETWEEN
NEXT_DAY
SYSDATE
Summary

Objectives Summarized
In this lesson, you have learned to:

- Select and apply the single-row functions MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, and TRUNC that operate on date data
- Explain how date functions transform Oracle dates into date data or a numeric value
- Demonstrate proper use of the arithmetic operators with dates
- Demonstrate the use of SYSDATE and date functions
- State the implications for world businesses to be able to easily manipulate data stored in date format
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Conversion Functions
What Will I Learn?

Objectives

In this lesson, you will learn to:

- Provide an example of an explicit data-type conversion and an implicit data-type conversion
- Explain why it is important, from a business perspective, for a language to have built-in data-conversion capabilities
- Construct a SQL query that correctly applies TO_CHAR, TO_NUMBER and TO_DATE single-row functions to produce a desired result
- Apply the appropriate date and/or character format model to produce a desired output
- Explain and apply the use of YYYY and RRRR to return the correct year as stored in the database
Why Learn It?

Purpose
Imagine having to read all your school books in text files with no paragraphs and no capitalization. It would be difficult to read. Fortunately, there are software programs available to capitalize and color text, underline, bold, center and add graphics.

For databases, format and display changes are done using conversion functions. These functions are able to display numbers as local currency, format dates in a variety of formats, display time to the second, and keep track of what century a date refers to.
Tell Me / Show Me

Data Types
When a table is created for a database, the SQL programmer must define what kind of data will be stored in each field of the table. In SQL, there are several different data types. These data types define the domain of values that each column can contain. For this lesson, you will use:

VARCHAR2
CHAR
NUMBER
DATE
Tell Me / Show Me

Data Types Described

**VARCHAR2**: Used for character data of variable length, including numbers, special characters and dashes.

**CHAR**: Used for text and character data of fixed length, including numbers, dashes and special characters.

**NUMBER**: Used to store variable-length numeric data. No dashes, text, or other nonnumeric data are allowed. Currency is stored as a number data type.

**DATE**: Used for date and time values. Internally, Oracle stores dates as numbers and by default DATE information is displayed as DD-MON-YY (for example, 19-JUN-04).
Tell Me / Show Me

Type Conversion
The Oracle Server can internally convert VARCHAR2 and CHAR data to NUMBER and DATE data types. It can convert NUMBER and DATE data back to CHARACTER datatype. Although this is a convenient feature, it is always best to explicitly make date-type conversions to ensure reliability in SQL statements.

The four data-type conversion functions you will learn are:

- To convert date data type to character data type
- To convert number data type to character data type
- To convert character data type to number data type
- To convert character data type to date data types
Tell Me / Show Me

Date Conversion to Character Data

It is often desirable to convert dates stored in a database in the default DD-MON-YY format to another format specified by you. The function to accomplish this task is:

```
TO_CHAR (date column name, 'format model you specify')
```

- The ‘format model’ must be enclosed in single quotation marks and is case-sensitive.
- Separate the date value from the ‘format model with a comma.
- Any valid date format element can be included.
- Use an fm element to remove padded blanks or remove leading zeroes from the output.
- Use sp to spell out a number.
- Use th to have the number appear as an ordinal. (1st, 2nd, 3rd and so on)
- Use double quotation marks to add character strings to format models.
Convert Me / Show Me

Date Conversion to Character Data (continued)

The tables show the different format models that can be used. When specifying time elements, note that hours (HH), minutes (MI), seconds (SS), and AM or PM can also be formatted.

| YYYY | Full year in numbers |
| YEAR | Year spelled out |
| MM | Two-digit value for month |
| MONTH | Full name of the month |
| MON | Three-letter abbreviation of the month |
| DY | Three-letter abbreviation of the day of the week |
| DAY | Full name of the day of the week |
| DD | Numeric day of the month |

| HH24:MI:SS AM | 15:45:32 PM |
| DD “of” MONTH | 12 of October |

| DDspth | FOURTEENTH |
| Ddspth | Fourteenth |
| ddspth | fourteenth |
| DDD or DD or D | Day of year, month or week |
Date Conversion to Character Data (continued)
If an employee’s hire date was 04-MAY-04, the format model using \textit{fm} would have returned May 4, 2004, suppressing the leading zero.

\begin{verbatim}
SELECT TO_CHAR (hire_date, 'fmMonth dd, YYYY')
FROM employees;
\end{verbatim}

What will be the output of the following query?

\begin{verbatim}
SELECT employee_id, TO_CHAR(hire_date, 'MONTH DD, YYYY')
FROM employees;
\end{verbatim}
Tell Me / Show Me

Date and Time Format Models

The following tables show variations of the format models that can be used with dates and time. Can you identify the format models used to produce today's date as the following output?

August 6th, 2007
August 06, 2007
AUG 6, 2007
August 6th, Friday, Two Thousand Seven

<table>
<thead>
<tr>
<th>Format Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>Full year in numbers</td>
</tr>
<tr>
<td>YEAR</td>
<td>Year spelled out</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit value for month</td>
</tr>
<tr>
<td>MONTH</td>
<td>Full name of the month</td>
</tr>
<tr>
<td>MON</td>
<td>Three-letter abbreviation of the month</td>
</tr>
<tr>
<td>DY</td>
<td>Three-letter abbreviation of the day of the week</td>
</tr>
<tr>
<td>DAY</td>
<td>Full name of the day of the week</td>
</tr>
<tr>
<td>DD</td>
<td>Numeric day of the month</td>
</tr>
<tr>
<td>HH24:MI:SS AM</td>
<td>15:45:32 PM</td>
</tr>
<tr>
<td>DD “of” MONTH</td>
<td>12 of October</td>
</tr>
<tr>
<td>DDspth</td>
<td>FOURTEENTH</td>
</tr>
<tr>
<td>Ddspth</td>
<td>Fourteenth</td>
</tr>
<tr>
<td>ddsdpth</td>
<td>fourteenth</td>
</tr>
<tr>
<td>DDD or DD or D</td>
<td>Day of year, month or week</td>
</tr>
</tbody>
</table>
Number Conversion to Character Data (VARCHAR2)

Numbers stored in the database have no formatting. This means that there are no currency signs/symbols, no commas, no decimals or other formatting. To add formatting, you first need to convert the number to a character format. This conversion is especially useful with concatenation.

The SQL function that you use to convert columns of number data to a desired format is:

```
TO_CHAR(number, 'format model')
```
Tell Me / Show Me

Number Conversion to Character Data (VARCHAR2) (continued)

The table illustrates some of the format elements available to use with TO_CHAR functions.

```sql
SELECT TO_CHAR(cost, '$99,999') COST 
FROM d_events;
```

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>DESCRIPTION</th>
<th>EXAMPLE</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Numeric position (# of 9’s determine width)</td>
<td>999999</td>
<td>1234</td>
</tr>
<tr>
<td>0</td>
<td>Display leading zeros</td>
<td>099999</td>
<td>001234</td>
</tr>
<tr>
<td>$</td>
<td>Floating dollar sign</td>
<td>$999999</td>
<td>$1234</td>
</tr>
<tr>
<td>L</td>
<td>Floating local currency symbol</td>
<td>L999999</td>
<td>FF1234</td>
</tr>
<tr>
<td>.</td>
<td>Decimal point in position specified</td>
<td>999999.99</td>
<td>1234.00</td>
</tr>
<tr>
<td>,</td>
<td>Comma in position specified</td>
<td>999,999</td>
<td>1,234</td>
</tr>
<tr>
<td>MI</td>
<td>Minus signs to right (negative values)</td>
<td>999999Mi</td>
<td>1234-</td>
</tr>
<tr>
<td>PR</td>
<td>Parenthesize negative numbers</td>
<td>999999PR</td>
<td>&lt;1234&gt;</td>
</tr>
<tr>
<td>EEEE</td>
<td>Scientific notation ( must have four EEEE)</td>
<td>99.999EEE</td>
<td>1,23E+03</td>
</tr>
<tr>
<td>V</td>
<td>Multiply by 10 n times (n= number of 9’s after V)</td>
<td>9999V99</td>
<td>9999V99</td>
</tr>
<tr>
<td>B</td>
<td>Display zero values as blank, not 0</td>
<td>B9999.99</td>
<td>1234.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COST</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$8,000</td>
<td></td>
</tr>
<tr>
<td>$10,000</td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Number Conversion to Character Data (VARCHAR2) (continued)

Can you identify the format models used to produce the following output?

$3000.00
4,500
9,000.00
0004422

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>DESCRIPTION</th>
<th>EXAMPLE</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Numeric position (# of 9’s determine width)</td>
<td>999999</td>
<td>1234</td>
</tr>
<tr>
<td>0</td>
<td>Display leading zeros</td>
<td>099999</td>
<td>001234</td>
</tr>
<tr>
<td>$</td>
<td>Floating dollar sign</td>
<td>$999999</td>
<td>$1234</td>
</tr>
<tr>
<td>L</td>
<td>Floating local currency symbol</td>
<td>L999999</td>
<td>FF1234</td>
</tr>
<tr>
<td>.</td>
<td>Decimal point in position specified</td>
<td>999999.99</td>
<td>1234.00</td>
</tr>
<tr>
<td>,</td>
<td>Comma in position specified</td>
<td>999,999</td>
<td>1,234</td>
</tr>
<tr>
<td>MI</td>
<td>Minus signs to right (negative values)</td>
<td>999999MI</td>
<td>1234-</td>
</tr>
<tr>
<td>PR</td>
<td>Parenthesize negative numbers</td>
<td>999999PR</td>
<td>&lt;1234&gt;</td>
</tr>
<tr>
<td>EEEE</td>
<td>Scientific notation ( must have four EEEE)</td>
<td>99.999EEE</td>
<td>1.23E+03</td>
</tr>
<tr>
<td>V</td>
<td>Multiply by 10 n times (n= number of 9’s after V)</td>
<td>9999V99</td>
<td>9999V99</td>
</tr>
<tr>
<td>B</td>
<td>Display zero values as blank, not 0</td>
<td>B9999.99</td>
<td>1234.00</td>
</tr>
</tbody>
</table>
Character Conversion to Number

It is often desirable to convert a character string to a number. The function for this conversion is:

```
TO_NUMBER(character string, 'format model')
```

This converts a nonnumeric value such as "450" to a number, without the single quotes. The single quotes are characters. The "450" was stored in the database as character data, and the following query converts it to a number so that arithmetic operations can be performed. You cannot perform calculations with character data.

```
SELECT TO_NUMBER('450') AS "Number Change"
FROM DUAL;
```

<table>
<thead>
<tr>
<th>Number Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>450</td>
</tr>
</tbody>
</table>
SELECT TO_NUMBER('450', '9999') + 10 AS "Number Change" FROM DUAL;

<table>
<thead>
<tr>
<th>Number Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>460</td>
</tr>
</tbody>
</table>

SQL*Plus displays a string of hash signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model and rounds numbers to the decimal place provided in the format model.

Oracle Application Express will return an Oracle Error – Invalid Number, if the format model does not match the actual number returned by the database.
Tell Me / Show Me

Character Conversion to Date
To convert a character string to a date format, use:

\[ \text{TO\_DATE}('\text{character string}', '\text{format model}') \]

This conversion takes a nondate value character string such as "November 3, 2001" and converts it to a date value. The format model tells the server what the character string "looks like":

\[ \text{TO\_DATE}('\text{November 3, 2001}', '\text{Month dd, RRRR}') \]
will return 03-NOV-01
Character Conversion to Date (continued)

When making a character-to-date conversion, the fx (format exact) modifier specifies exact matching for the character argument and the date format model.

In the following example, note that "May10" has no space between "May" and "10." The fx format model matches the character argument as it also has no space between "Mon" and "DD."

```
SELECT TO_DATE('May10,1989', 'fxMonDD,RRRR') AS "Convert"
FROM DUAL;
```

```
CONVERT
10-MAY-89
```
fx Modifier Rules

The fx modifier rules are:

• Punctuation and quoted text in the character argument must match the corresponding parts of the format model exactly (except for case).
• The character argument cannot have extra blanks. Without fx, the Oracle Server ignores extra blanks.
• Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without fx, numbers in the character argument can omit leading zeros.
Tell Me / Show Me

RR Date Format and YY Date Format

It hasn't been that long since the century changed from 1900 to 2000. Along with this change came considerable confusion as to whether a date written as 02-JAN-00 would be interpreted as January 2, 1900 or January 2, 2000. Fortunately, Oracle has a way to keep these dates stored and retrievable with the correct century.

<table>
<thead>
<tr>
<th>Current Year</th>
<th>Specified Date</th>
<th>RR Format</th>
<th>YY Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>1917</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>2017</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-95</td>
<td>1995</td>
<td>2095</td>
</tr>
</tbody>
</table>

If the specified two-digit year is:

<table>
<thead>
<tr>
<th>0-49</th>
<th>50-99</th>
</tr>
</thead>
</table>

If two digits of the current year are:

| 0-49         | 50-99          |

- The return date is in the current century
- The return date is in the century before the current one
- The return date is in the century after the current one
Tell Me / Show Me

A Few Simple Rules

If the date format is specified with the YY or YYYYY format, the return value will be in the same century as the current century. So, if the year is 1995 and you use the YY or YYYYY format, all is well and the dates will be in the 1900s. However, if the current year is 2004 and you use the YY or YYYYY format for a date such as 1989, you will get 2089! Maybe not what you intended.

<table>
<thead>
<tr>
<th>Current Year</th>
<th>Specified Date</th>
<th>RR Format</th>
<th>YY Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>1917</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>2017</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-95</td>
<td>1995</td>
<td>2095</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

A Few Simple Rules (continued)

If the date format is specified with the RR or RRRR format, the return value has two possibilities.

If the current year is between 00-49:
- Dates from 0-49: The date will be in the current century
- Dates from 50-99: The date will be in the last century

If the current year is between 50-99:
- Dates from 0-49: The date will be in next century
- Dates from 50-99: The date will be in current century

<table>
<thead>
<tr>
<th>If two digits of the current year are:</th>
<th>If the specified two-digit year is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-49</td>
<td>0-49</td>
</tr>
<tr>
<td></td>
<td>The return date is in the current century</td>
</tr>
<tr>
<td></td>
<td>The return date is in the century before the current one</td>
</tr>
<tr>
<td>50-99</td>
<td>50-99</td>
</tr>
<tr>
<td></td>
<td>The return date is in the century after the current one</td>
</tr>
<tr>
<td></td>
<td>The return date is in the current century</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

A Few Simple Rules
(continued)

When I query my employee database using the following statement, it returns every row in the table. I know there are only a few employees who were hired before 1990. Did I make a mistake?

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YY')
FROM employees
WHERE hire_date < TO_DATE('01-Jan-90', 'DD-Mon-YY');
```

---

### Conversion Functions

<table>
<thead>
<tr>
<th>Current Year</th>
<th>Specified Date</th>
<th>RR Format</th>
<th>YY Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>1917</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>2017</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-95</td>
<td>1995</td>
<td>2095</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>If the specified two-digit year is:</th>
<th>0-49</th>
<th>50-99</th>
</tr>
</thead>
<tbody>
<tr>
<td>If two digits of the current year are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0-49</td>
<td>The return date is in the current century</td>
<td>The return date is in the century before the current one</td>
</tr>
<tr>
<td>50-99</td>
<td>The return date is in the century after the current one</td>
<td>The return date is in the current century</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology

Key terms used in this lesson include:

- CHAR
- DATE
- DD date format
- Conversion function
- fm
- NUMBER
- RR date format
- TO_CHAR
- TO_DATE
- TO_NUMBER
- VARCHAR2
Summary

Objectives Summarized

In this lesson, you have learned to:

• Provide an example of an explicit data-type conversion and an implicit data-type conversion

• Explain why it is important, from a business perspective, for a language to have built-in data-conversion capabilities

• Construct a SQL query that correctly applies TO_CHAR, TO_NUMBER and TO_DATE single-row functions to produce a desired result

• Apply the appropriate date and/or character format model to produce a desired output

• Explain and apply the use of YYYY and RRRR to return the correct year as stored in the database
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
NULL Functions
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Demonstrate and explain the evaluation of a nested function

• List at least four general functions that work with any data type and relate to handling null values

• Explain the use of the COALESCE and the NVL functions

• Explain the use of general functions to deal with null values in data

• Construct and execute a SQL query that correctly applies NVL, NVL2, NULLIF, and COALESCE single-row functions
Why Learn It?

Purpose
Besides functions that control how data is formatted or converted to another type, SQL uses a set of general functions designed specifically to deal with null values. You may be wondering how a value that is unavailable, unassigned, unknown, or inapplicable can deserve so much attention. Null may be "nothing," but it can affect how expressions are evaluated, how averages are computed, and where a value appears in a sorted list. This lesson is all about handling null values.
Tell Me / Show Me

How Functions are Evaluated

Up to now, you have applied single-row functions in simple statements. It is possible, however, to nest functions to any depth. What is important to know is how nested functions are evaluated. The following example, is a nested function. The evaluation process begins from the innermost level to the outermost level.

```
SELECT TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6), 'FRIDAY'),
            'fmDay, Month DDth, YYYY') AS "Next Evaluation"
FROM employees
WHERE employee_id=100;
```

The results are:
Friday, December 18th, 1987
Tell Me / Show Me

How Functions are Evaluated (continued)

```
SELECT TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6), 'FRIDAY'),
'fmDay, Month DDth, YYYY') AS "Next Evaluation"
FROM employees
WHERE employee_id=100;
```

• Step 1: The hire date is going to have six months added to it.
• Step 2: The first Friday following the future day will be identified.
• Step 3: The default date format will be formatted to read and display the Friday in a format similar to: Friday, December 18th, 1987, and will appear in the output under the column name "Next Evaluation."
Functions Pertaining to Null Values
At the beginning of the course, the term "null" was introduced. If you remember, it's the value that is unavailable, unassigned, unknown, or inapplicable. We basically cannot test to see if it is the same as another value, because we do not know what value it has. It isn't equal to anything, not even zero! But just because it really isn't anything doesn't mean that it is not important. Imagine this question: Is it true that $X = Y$? In order to answer you have to know the values of $X$ and $Y$. Oracle has four general functions that pertain to the use of null values. The four functions are:
- NVL
- NVL2
- NULLIF
- COALESCE
Tell Me / Show Me

NVL Function

The NVL function converts a null value to a known value of a fixed datatype, either date, character or number. The data types of the null value column and the new value must be the same. The NVL function is:

NVL (value or column that may contain a null, value to replace the null)

The following query uses the NVL function with character data types:

```sql
SELECT NVL(comments, 'no comment')
FROM D_PLAY_LIST_ITEMS;
```
Tell Me / Show Me

NVL Function (continued)

The data types of the null value column and the new value must be the same as shown in the following examples:

\[
\begin{align*}
\text{NVL(auth\_expense\_amt,0)} \\
\text{NVL(hire\_date,'01-JAN-97')} \\
\text{NVL(speciality,'None Yet')} \\
\end{align*}
\]
Tell Me / Show Me

NVL Function (continued)

You can use the NVL function to convert column values containing nulls to a number before doing calculations. When an arithmetic calculation is performed with null, the result is null. The NVL function can convert the null value to a number before arithmetic calculations are done to avoid a null result.
Tell Me / Show Me

NVL Function (continued)

In the example, the auth_expense_amt column in the D_PARTNERS table contains null values. The NVL function is used to change the null to zero before arithmetic calculations.

```sql
SELECT first_name, last_name, NVL(auth_expense_amt, 0) * 1.05 AS Expenses
FROM D_Partners;
```
Tell Me / Show Me

NVL2 Function

The NVL2 function evaluates an expression with three values. If the first value is not null, then the NVL2 function returns the second expression.

If the first value is null, then the third expression is returned. The values in expression 1 can have any data type. Expression 2 and expression 3 can have any data type except LONG.

The data type of the returned value is always the same as the data type of expression 2, unless expression 2 is character data, in which case the returned type is VARCHAR2.
Tell Me / Show Me

NVL2 Function (continued)

The NVL2 function is:

\[
\text{NVL2 (expression 1 value that may contain a null, expression 2 value to return if expression 1 is not null, expression 3 value to replace if expression 1 is null)}
\]

An easy way to remember NVL2 is to think, "if expression 1 has a value, substitute expression 2; if expression 1 is null, substitute expression 3."

The NVL2 function shown has number data for expression 1 and character data for expressions 2 and 3.

```sql
SELECT last_name, salary,
       NVL2(commission_pct, salary + (salary * commission_pct), salary) AS income
FROM employees;
```
Tell Me / Show Me

NULLIF Function

The NULLIF function compares two functions. If they are equal, the function returns null. If they are not equal, the function returns the first expression.

The NULLIF function is:

```
NULLIF(expression 1, expression 2)
```

Example:

```
SELECT first_name, LENGTH(first_name) "Length FN",
      last_name, LENGTH(last_name) "Length LN",
      NULLIF(LENGTH(first_name),
             LENGTH(last_name)) AS "Compare Them"
FROM D_PARTNERS;
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>Length FN</th>
<th>LAST_NAME</th>
<th>Length LN</th>
<th>Compare Them</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jennifer</td>
<td>8</td>
<td>Cho</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Jason</td>
<td>5</td>
<td>Tsang</td>
<td>5</td>
<td>(null)</td>
</tr>
<tr>
<td>Allison</td>
<td>7</td>
<td>Plumb</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
COALESCE Function

The COALESCE function is an extension of the NVL function, except COALESCE can take multiple values. The word "coalesce" literally means "to come together" and that is what happens.

If the first expression is null, the function continues down the line until a not null expression is found. Of course, if the first expression has a value, the function returns the first expression and the function stops.
COALESCE Function
(continued)
The COALESCE function is:
COALESCE (expression 1, expression 2, ...
expression n)

Examine the SELECT statement from the employees table shown at right. Which employees do not receive a commission? How can you tell? Is there anyone who receives neither a commission percentage nor a salary?

```
SELECT last_name, COALESCE(commission_pct, salary, 10) comm
FROM employees
ORDER BY commission_pct;
```

<table>
<thead>
<tr>
<th>Last Name</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grant</td>
<td>.15</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>.2</td>
</tr>
<tr>
<td>Taylor</td>
<td>.2</td>
</tr>
<tr>
<td>Abel</td>
<td>.3</td>
</tr>
<tr>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

NVL
NVL2
NULLIF
COALESCE
Summary

Objectives Summarized
In this lesson, you have learned to:

• Demonstrate and explain the evaluation of a nested function
• List at least four general functions that work with any data type and relate to handling null values
• Explain the use of the COALESCE and the NVL functions
• Explain the use of general functions to deal with null values in data
• Construct and execute a SQL query that correctly applies NVL, NVL2, NULLIF and COALESCE single-row functions
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Conditional Expressions
Objectives

In this lesson, you will learn to:

• Compare and contrast the DECODE and CASE functions

• Construct and execute a SQL query that correctly uses the DECODE and CASE functions

• Construct and execute two methods for implementing IF-THEN-ELSE conditional logic
Why Learn It?

Purpose
Being able to make decisions is essential in data modeling.

Modelers have to decide which business functions need to be modeled and which do not. The data-modeling process requires the designers to analyze information to identify entities, resolve relationships and select attributes.

A typical decision could be, IF a business needs to track data over time, THEN time may need to be an entity or ELSE time should be an attribute.
Why Learn It?

Purpose (continued)

This decision process is not much different from those we make in everyday life.

Think of the last time you had to make an if-then-else kind of decision. If I get my homework done before 9:00 p.m., I can watch television, else I can't watch television.

In SQL, these kinds of choices involve conditional-processing methods. Knowing how to use conditional processing makes decision making to get the data you want easier.
Conditional Expressions

The two conditional expressions are CASE and DECODE. You have already studied NULLIF, which is logically equivalent to the CASE expression in that CASE compares two expressions. If the two expressions are equal, then return null; if they are not equal, then return the first expression.

CASE Expression

The CASE expression basically does the work of an IF-THEN-ELSE statement. Data types of the CASE, WHEN, and ELSE expressions must be the same.
Tell Me / Show Me

CASE Syntax

```
CASE expr WHEN comparison_expr1 THEN return_expr1
    [WHEN comparison_expr2 THEN return_expr2
     WHEN comparison_exprn THEN return_exprn
     ELSE else_expr]
END
```

```
SELECT id, loc_type, rental_fee,
CASE loc_type
    WHEN 'Private Home' THEN 'No Increase'
    WHEN 'Hotel' THEN 'Increase 5%'
    ELSE rental_fee
END AS 'REVISED_FEES'
FROM d_venues;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>LOC_TYPE</th>
<th>RENTAL_FEE</th>
<th>REVISED_FEES</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>105</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>101</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>95</td>
<td>School Hall</td>
<td>75/hour</td>
<td>75/hour</td>
</tr>
<tr>
<td>99</td>
<td>National Park</td>
<td>400/flat fee</td>
<td>400/flat fee</td>
</tr>
<tr>
<td>220</td>
<td>Hotel</td>
<td>300/per person</td>
<td>Increase 5%</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

DECODE Expression

The DECODE function evaluates an expression in a similar way to the IF-THEN-ELSE logic. DECODE compares an expression to each of the search values. The syntax for DECODE is:

```
DECODE(column1|expression, search1, result1
     [, search2, result2,...,]
     [, default])
```

If the default value is omitted, a null value is returned where a search value does not match any of the values.

<table>
<thead>
<tr>
<th>ID</th>
<th>LOC_TYPE</th>
<th>RENTAL_FEE</th>
<th>REVISED_FEES</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>105</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>101</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>95</td>
<td>School Hall</td>
<td>75/hour</td>
<td>75/hour</td>
</tr>
<tr>
<td>99</td>
<td>National Park</td>
<td>400/flat fee</td>
<td>400/flat fee</td>
</tr>
<tr>
<td>220</td>
<td>Hotel</td>
<td>300/per person</td>
<td>Increase 5%</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

DECODE Expression (continued)
Examine the example:

```
SELECT id, loc_type, rental_fee,
DECODE( loc_type , 'Private Home', 'No Increase',
       'Hotel', 'Increase 5%',
       rental_fee )
AS "REVISED_FEES"
FROM d_venues;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>LOC_TYPE</th>
<th>RENTAL_FEE</th>
<th>REVISED_FEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>105</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>101</td>
<td>Private Home</td>
<td>0</td>
<td>No increase</td>
</tr>
<tr>
<td>95</td>
<td>School Hall</td>
<td>75/hour</td>
<td>75/hour</td>
</tr>
<tr>
<td>99</td>
<td>National Park</td>
<td>400/flat fee</td>
<td>400/flat fee</td>
</tr>
<tr>
<td>220</td>
<td>Hotel</td>
<td>300/per person</td>
<td>Increase 5%</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Conditional expression
CASE
DECODE
Summary

Objectives Summarized

In this lesson, you have learned to:

• Compare and contrast the DECODE and CASE functions
• Construct and execute a SQL query that correctly uses the DECODE and CASE functions
• Construct and execute two methods for implementing IF-THEN-ELSE conditional logic
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Cross Joins and Natural Joins
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Construct and execute a natural join using ANSI-99 SQL join syntax
• Create a cross join using ANSI-99 SQL join syntax
• Define the relationship between a cross join and a Cartesian product
• Define the relationship between a natural join and an equijoin
• Explain why it is important to have a standard for SQL as defined by ANSI
• Describe a business need for combining information from multiple data sources
Why Learn It?

Purpose
Up to now, your experience using SQL has been limited to querying and returning information from one database table at a time.

This would not be a problem if all data in the database were stored in only one table. But you know from data modeling that separating data into individual tables and being able to associate the tables with one another is the heart of relational database design. Fortunately, SQL provides join conditions that enable information to be queried from separate tables and combined in one report.
Cross Joins and Natural Joins

Tell Me / Show Me

Join Commands
There are two sets of commands or syntax which can be used to make connections between tables in a database:

- Oracle proprietary joins
- ANSI/ISO SQL 99 compliant standard joins

In this course, you will learn to use both sets of join commands.
Tell Me / Show Me

ANSI

ANSI stands for American National Standards Institute. Founded in 1918, ANSI is a private, non-profit organization that administers and coordinates the U.S. voluntary standardization and conformity assessment system.

The Institute's mission is to enhance both the global competitiveness of U.S. business and the U.S. quality of life by promoting and facilitating voluntary consensus standards and conformity assessment systems, and safeguarding their integrity.

Reference: http://www.ansi.org/default.aspx
SQL

Structured Query Language (SQL) is the information-processing industry-standard language of relational database management systems (RDBMS).

The language was originally designed by IBM in the mid 1970s, came into widespread use in the early 1980s, and became an industry standard in 1986, when it was adopted by ANSI.

So far there have been three ANSI standardizations of SQL, each one building on the previous one. They are named after the year in which they were first proposed, and are widely known by their short names: ANSI-86, ANSI-92 and ANSI-99.
Tell Me / Show Me

NATURAL JOIN
An Oracle proprietary equijoin returns all rows whose values match in both tables.

The ANSI/ISO SQL: 1999 join that accomplishes the same result is called a natural join.

A natural join is based on all columns in the two tables that have the same name and selects rows from the two tables that have equal values in all matched columns.
NATURAL JOIN (continued)
As shown in the sample code, when using a natural join, it is possible to join the tables without having to explicitly specify the columns in the corresponding table. However, the names and data types in both columns must be the same.

```
SELECT event_id, song_id, cd_number
FROM d_play_list_items NATURAL
JOIN d_track_listings
WHERE event_id = 105;
```

The WHERE clause was added to apply an additional restriction to one of the tables, to limit the rows of output.
Here is another example:

```
SELECT first_name, last_name, event_date, description
FROM d_clients NATURAL JOIN d_events;
```

Which column or columns will be used to natural join these two tables?

Notice that the natural join column does not have to appear in the output.
CROSS JOIN

An Oracle Proprietary Cartesian Product joins every row in one table to every row in the other table.

The ANSI/ISO SQL: 1999 SQL equivalent of the Cartesian product is the cross join.

The results returned from both types of joins are the same. The results set represents all possible combinations of columns from both tables.

This could potentially be very large!
Cross Joins and Natural Joins

Tell Me / Show Me

Cross Join Example

```
SELECT name, event_date, loc_type, rental_fee
FROM d_events CROSS JOIN d_venues;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>EVENT_DATE</th>
<th>LOC_TYPE</th>
<th>RENTAL_FEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Private Home</td>
<td>0</td>
</tr>
<tr>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Private Home</td>
<td>0</td>
</tr>
<tr>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Private Home</td>
<td>0</td>
</tr>
<tr>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>School Hall</td>
<td>75/hour</td>
</tr>
<tr>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>National Park</td>
<td>400/flat fee</td>
</tr>
<tr>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Hotel</td>
<td>300/per person</td>
</tr>
<tr>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Private Home</td>
<td>0</td>
</tr>
<tr>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Private Home</td>
<td>0</td>
</tr>
<tr>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Private Home</td>
<td>0</td>
</tr>
<tr>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>School Hall</td>
<td>75/hour</td>
</tr>
<tr>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>National Park</td>
<td>400/flat fee</td>
</tr>
<tr>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Hotel</td>
<td>300/per person</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Cross Join
Natural Join
Summary

Objectives Summarized
In this lesson you have learned to:

- Compose and execute a natural join using ANSI-99 SQL join syntax
- Create a cross join using ANSI-99 SQL join syntax
- Define the relationship between a cross join and a Cartesian product
- Define the relationship between a natural join and an equijoin
- Explain why it is important to have a standard for SQL as defined by ANSI
- Provide evidence to answer the question “Why is it important, from a business perspective, for a language to be able to combine information from multiple data sources?”
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Join Clauses
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Construct and execute a join with the ANSI-99 USING and ON clauses
• Construct and execute an ANSI-99 query that joins three tables
Why Learn It?

Purpose
As you add more commands to your database vocabulary, you will be better able to design queries that return the desired result. The purpose of a join is to bind data together, across tables, without repeating all of the data in every table. Why ask for more data than you really need?
Tell Me / Show Me

**USING Clause**

In a natural join, if the tables have columns with the same names but different data types, the join causes an error.

To avoid this situation, the join clause can be modified with a USING clause.

The USING clause specifies the columns that should be used for the equijoin.
USING Clause (continued)
The query shown is an example of the USING clause. The columns referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement.

```
SELECT client_number, first_name, last_name, event_date
FROM d_clients JOIN d_events
USING (client_number);
```

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EVENT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>14-MAY-04</td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>28-APR-04</td>
</tr>
</tbody>
</table>
Using Clause (continued)

The USING clause allows us to use WHERE to restrict rows from one or both tables:

```
SELECT client_number, first_name, last_name, event_date
FROM d_clients JOIN d_events
USING (client_number)
WHERE last_name = 'Peters';
```

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EVENT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>14-MAY-04</td>
</tr>
</tbody>
</table>

Tell Me / Show Me

ON Clause
What if the columns to be joined have different names, or if the join uses non-equality comparison operators such as <, > or BETWEEN?

We can’t use USING, so instead we use an ON clause. This allows a greater variety of join conditions to be specified.

The ON clause also allows us to use WHERE to restrict rows from one or both tables.
Tell Me / Show Me

ON Clause Example

In this example, the ON clause is used in a self-join where the same table is given two different references. In the employees table, some employees are also managers. The self-join is used to select those employees who are also managers.

```
SELECT e.last_name as "EMP", m.last_name as "MGR"
FROM employees e JOIN employees m
ON (e.manager_id = m.employee_id);
```

<table>
<thead>
<tr>
<th>EMP</th>
<th>MGR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>King</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>King</td>
</tr>
<tr>
<td>Mourgos</td>
<td>King</td>
</tr>
<tr>
<td>De Haan</td>
<td>King</td>
</tr>
<tr>
<td>Kochhar</td>
<td>King</td>
</tr>
<tr>
<td>Higgins</td>
<td>Kochhar</td>
</tr>
<tr>
<td>..........</td>
<td>........</td>
</tr>
</tbody>
</table>
ON Clause with WHERE Clause

Here is the same query with a WHERE clause to restrict the rows selected.

```sql
SELECT e.last_name as "EMP", m.last_name as "MGR"
FROM employees e JOIN employees m
ON (e.manager_id = m.employee_id)
WHERE e.last_name like 'H%';
```

<table>
<thead>
<tr>
<th>EMP</th>
<th>MGR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>King</td>
</tr>
<tr>
<td>Higgins</td>
<td>Kochhar</td>
</tr>
<tr>
<td>Hunold</td>
<td>De Haan</td>
</tr>
</tbody>
</table>
Joining Three Tables

Both USING and ON can be used to join three or more tables.

Suppose we need a report of our clients, their events and the themes for those events? We need to join three tables: d_clients, d_events and d_themes.
Tell Me / Show Me

Joining Three Tables Example

```sql
SELECT last_name, event_date, t.description
FROM d_clients c JOIN d_events e
USING (client_number)
JOIN d_themes t
ON (e.theme_code = t.code);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>EVENT_DATE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peters</td>
<td>14-MAY-04</td>
<td>Tropical</td>
</tr>
<tr>
<td>Vigil</td>
<td>28-APR-04</td>
<td>Tropical</td>
</tr>
</tbody>
</table>
## Join Comparison


<table>
<thead>
<tr>
<th>Oracle Proprietary Join</th>
<th>ANSI/ISO SQL: 1999 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartesian Product</td>
<td>Cross Join</td>
</tr>
<tr>
<td>Equijoin</td>
<td>Natural Join (if the join columns have the same name and data type)</td>
</tr>
<tr>
<td></td>
<td>USING clause (if the columns have the same name but different data types)</td>
</tr>
<tr>
<td>Non-equijoin</td>
<td>ON clause (if the columns have different names)</td>
</tr>
<tr>
<td></td>
<td>ON clause</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

USING clause
ON clause
Summary

Objectives Summarized
In this lesson you have learned to:

- Construct and execute a join with the ANSI-99 USING and ON clauses
- Construct and execute an ANSI-99 query that joins three tables
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Inner Versus Outer Joins
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Compare and contrast an inner and an outer join
• Construct and execute a query to use a left outer join
• Construct and execute a query to use a right outer join
• Construct and execute a query to use a full outer join
Why Learn It?

Purpose
Up to now, all of the joins returned data that matched the join condition. Sometimes, however, we want not only to retrieve data that meets the join condition, but also to retrieve data that doesn’t meet a join condition. This should sound familiar!

The outer joins in ANSI-99 SQL allow this functionality.
Tell Me / Show Me

INNER And OUTER Joins

In ANSI-99 SQL, a join of two or more tables that return only matched rows is called an inner join.

When a join returns the unmatched rows as well as matched rows, it is called an outer join.

Outer join syntax uses the terms “left, full, and right.” These names are associated with the order of the table names in the FROM clause of the SELECT statement.
**Tell Me / Show Me**

**LEFT and RIGHT OUTER Joins**

In the example shown of a left outer join, note that the table name listed to the left of the words “left outer join” is referred to as the "left table."

This query will return all matched rows as well as all employee last names even if they aren’t assigned to a department.

```
SELECT e.last_name, d.department_id, d.department_name
FROM employees e
LEFT OUTER JOIN departments d
ON (e.department_id = d.department_id);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>DEPT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Grant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Tell Me / Show Me**

**LEFT and RIGHT OUTER Joins (continued)**

This right outer join would return all department IDs and department names even if no employees were assigned to them.

```
SELECT e.last_name, d.department_id, d.department_name
FROM employees e
RIGHT OUTER JOIN departments d
ON (e.department_id = d.department_id);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>DEPT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td></td>
<td>190</td>
<td>Contracting</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

FULL OUTER Join

It is possible to create a join condition to retrieve all matching rows and all unmatched rows from both tables in a join.

Using a full outer join solves this problem. The results set of a full outer join includes all rows in both tables even if there is no match in the other table.
**Tell Me / Show Me**

**FULL OUTER Join Example**
The example shown is a full outer join.

```sql
SELECT e.last_name, d.department_id, d.department_name
FROM employees e
FULL OUTER JOIN departments d
ON (e.department_id = d.department_id);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>DEPT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>King</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Grant</td>
<td>190</td>
<td>Contracting</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Join Scenario
Construct a join to display a list of Global Fast Foods customers and their orders. Include all customers whether or not they have placed an order.

```
SELECT c.first_name, c.last_name, o.order_number, o.order_date, o.order_total
FROM   f_customers c
LEFT OUTER JOIN f_orders o
ON     (c.id = o.cust_id);
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>ORDER_NUMBER</th>
<th>ORDER_DATE</th>
<th>ORDER_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cole</td>
<td>Bee</td>
<td>5678</td>
<td>10-DEC-02</td>
<td>103.02</td>
</tr>
<tr>
<td>Zoe</td>
<td>Twee</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology

Key terms used in this lesson include:

FULL OUTER JOIN
Inner join
LEFT OUTER JOIN
Outer join
RIGHT OUTER JOIN
Summary

Objectives Summarized
In this lesson you have learned to:

• Compare and contrast an inner and an outer join
• Construct and execute a query to use a left outer join
• Construct and execute a query to use a right outer join
• Construct and execute a query to use a full outer join
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Self-Joins and Hierarchical Queries
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Construct and execute a SELECT statement to join a table to itself using a self-join
• Interpret the concept of a hierarchical query
• Create a tree-structured report
• Format hierarchical data
• Exclude branches from the tree structure
Why Learn It?

Purpose
In data modeling, it was sometimes necessary to show an entity with a relationship to itself. For example, an employee can also be a manager. We showed this using the "pig’s ear" relationship. Once we have a real employees table, a special kind of join called a self-join is required to access this data. You probably realize by now the importance of a data model once it becomes a database. It’s no coincidence that the data model looks similar to the tables we now have in the database.

```
SELECT worker.last_name || ' works for' || manager.last_name
FROM employees worker, employees manager
WHERE worker.manager_id = manager.employee_id;
```

(This EMPLOYEES table output shows columns resulting from a join)
**SELF-JOIN**

To join a table to itself, the table is given two names or aliases. This will make the database “think” that there are two tables. Choose alias names that relate to the data's association with that table.

Manager_id in the worker table is equal to employee_id in the manager table.
Tell Me / Show Me

SELF-JOIN Example
Here is another example of a self-join. In this example of a band, we have members of the band who play an instrument and members of the band who play an instrument and are their section's lead player or chair. A readable way to show this self-join is:

SELECT chair.last_name || ' is the section chair of ' || player.last_name
FROM band_members chair, band_members player
WHERE player.chair_id = chair.member_id;
Hierarchical Queries
Closely related to self-joins are hierarchical queries. On the previous pages you saw how you can use self-joins to see who is someone's direct manager. With hierarchical queries we can also see who that manager works for and so on.
We can basically build an Organization Chart showing the structure of a company or a department. Imagine a family tree with the eldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree. Branches can have their own branches, and so on.
Tell Me / Show Me

Using Hierarchical Queries

Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called tree walking enables the hierarchy to be constructed. A hierarchical query is a method of reporting the branches of a tree in a specific order.
Tell Me / Show Me

Hierarchical Queries Data
Examine the sample data from the EMPLOYEES table below, and look at how you can manually make the connections to see who works for whom starting with Steven King and going down the tree from there.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>COMM_PCT</th>
<th>MGR_ID</th>
<th>DEPT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
<td>515.123.4567</td>
<td>17-JUN-87</td>
<td>AD_PRES</td>
<td>24000</td>
<td>(null)</td>
<td>(null)</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOCHHAR</td>
<td>515.123.4568</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>17000</td>
<td>(null)</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
<td>LDEHAAN</td>
<td>515.123.4569</td>
<td>13-JAN-93</td>
<td>AD_VP</td>
<td>17000</td>
<td>(null)</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
<td>AHUNOLD</td>
<td>590.423.4567</td>
<td>03-JAN-90</td>
<td>IT_PROG</td>
<td>9000</td>
<td>(null)</td>
<td>102</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
<td>BERNST</td>
<td>590.423.4568</td>
<td>21-MAY-91</td>
<td>IT_PROG</td>
<td>6000</td>
<td>(null)</td>
<td>103</td>
<td>60</td>
</tr>
<tr>
<td>124</td>
<td>Kevin</td>
<td>Mourgos</td>
<td>KMOURGOS</td>
<td>650.123.5234</td>
<td>16-NOV-99</td>
<td>ST_MAN</td>
<td>5800</td>
<td>(null)</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>141</td>
<td>Trenna</td>
<td>Rajs</td>
<td>TRAJS</td>
<td>650.121.8009</td>
<td>17-OCT-95</td>
<td>ST_CLERK</td>
<td>3500</td>
<td>(null)</td>
<td>124</td>
<td>50</td>
</tr>
</tbody>
</table>
So the organization chart we can draw from the data in the EMPLOYEES table will look like this:
Hierarchical Queries Keywords

Hierarchical queries have their own new keywords: START WITH, CONNECT BY PRIOR and LEVEL.

START WITH is used to tell Oracle which row to use as its Root of the tree it is constructing, CONNECT BY PRIOR tells Oracle how to do the inter-row joins and LEVEL is how many steps down from the top of the tree we have taken.
Hierarchical Queries Keyword Example

```
SELECT employee_id, last_name, job_id, manager_id
FROM   employees
START  WITH  employee_id = 100
CONNECT BY PRIOR employee_id = manager_id
```

<table>
<thead>
<tr>
<th>employee_id</th>
<th>last_name</th>
<th>job_id</th>
<th>manager_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>(null)</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>AD_ASST</td>
<td>101</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>AC_MGR</td>
<td>101</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>205</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>100</td>
</tr>
<tr>
<td>103</td>
<td>Hunould</td>
<td>IT_PROG</td>
<td>102</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>124</td>
</tr>
</tbody>
</table>
Hierarchical Queries Another Example

```
SELECT  last_name||' reports to '||PRIOR   last_name "Walk Top Down"
FROM    employees
START   WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id
```

<table>
<thead>
<tr>
<th>Walk Top Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>King reports to</td>
</tr>
<tr>
<td>Kochhar reports to King</td>
</tr>
<tr>
<td>Whalen reports to Kochhar</td>
</tr>
<tr>
<td>Higgins reports to Kochhar</td>
</tr>
<tr>
<td>Gietz reports to Higgins</td>
</tr>
<tr>
<td>De Haan reports to King</td>
</tr>
<tr>
<td>Hunold reports to De Haan</td>
</tr>
<tr>
<td>Ernst reports to Hunold</td>
</tr>
</tbody>
</table>
Hierarchical Queries Level Example

```
SELECT LEVEL, last_name||' reports to '||PRIOR last_name "Walk Top Down"
FROM employees
START WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id
```

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>Walk Top Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>King reports to</td>
</tr>
<tr>
<td>2</td>
<td>Kochhar reports to King</td>
</tr>
<tr>
<td>3</td>
<td>Whalen reports to Kochhar</td>
</tr>
<tr>
<td>3</td>
<td>Higgins reports to Kochhar</td>
</tr>
<tr>
<td>4</td>
<td>Gietz reports to Higgins</td>
</tr>
<tr>
<td>2</td>
<td>De Haan reports to King</td>
</tr>
<tr>
<td>3</td>
<td>Hunold reports to De Haan</td>
</tr>
<tr>
<td>4</td>
<td>Ernst reports to Hunold</td>
</tr>
</tbody>
</table>

LEVEL is a pseudo-column used with hierarchical queries, and it counts the number of steps it has taken from the root of the tree.
Hierarchical Query Report

If you wanted to create a report displaying company management levels, beginning with the highest level and indenting each of the following levels, then this would be easy to do using the LEVEL pseudo column and the LPAD function to indent employees based on their level.

```
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')  AS ORG_CHART
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```
Hierarchical Query Output Levels

SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')
       AS ORG_CHART
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id

As you can see in the result on the left, each row is indented by two underscores per level.
Hierarchical Queries Pruning

Pruning branches from the tree can be done using either the WHERE clause or the CONNECT BY PRIOR clause. If the WHERE clause is used then only the row named in the statement is excluded and if the CONNECT BY PRIOR is used that entire branch is excluded.
Hierarchical Queries

So if you wanted to just not include a single row in your result you would use the WHERE clause to exclude that row, but in the result it would then look like Gietz worked directly for Kochhar, which he does not.

```
SELECT last_name
FROM employees
WHERE last_name != 'Higgins'
START WITH last_name = 'Kochhar'
CONNECT BY PRIOR employee_id = manager_id
```
Hierarchical Queries (continued)

If, however, you wanted to exclude one row and all the rows below that one you should make the exclusion part of the CONNECT BY statement. In this example by excluding Higgins, we are also excluding Gietz in the result.

```
SELECT last_name
FROM employees
START WITH last_name = 'Kochhar'
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'Higgins
```
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Self join
Hierarchical Queries
Level
Start with
Connect By prior
Summary

Objectives Summarized
In this lesson you have learned to:

• Construct and execute a SELECT statement to join a table to itself using a self-join
• Interpret the concept of a hierarchical query
• Create a tree-structured report
• Format hierarchical data
• Exclude branches from the tree structure
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Review of Joins
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Determine the correct join syntax to use given a scenario requiring the join of data from two or more tables.
Why Learn It?

Purpose
Knowing when to use the correct join syntax to meet the needs stated in a business scenario requiring the join of data is very important to your success. This lesson will afford you the opportunity to review the join syntax.
Try It / Solve It

Classroom Activity

Try every example listed below. Confirm that your results match the expected result. If you need help, ask another student or the teacher. All example code is based on the Oracle database.

Cross Join

```
SELECT last_name, department_name
FROM employees CROSS JOIN departments;
```

All rows will show.

Natural Join

```
SELECT employee_id, last_name, department_name
FROM employees NATURAL JOIN departments;
```

Joins by column names and data types that are identical in each table. Both the employees and departments tables have the columns department_id and manager_id. Therefore, the query will return the rows where the values in both columns match.
Try It / Solve It

Classroom Activity (Continued)

Joins .. Using

SELECT employee_id, last_name, department_name
FROM employees JOIN departments
USING (department_id);

Joins by column names and data types that are identical in each table
but USING statement limits to one column.

Join .. On

SELECT e.employee_id, e.last_name, d.department_id, d.location_id
FROM employees e JOIN departments d
ON (e.department_id = d.department_id);

All employees and their work locations.
Try It / Solve It

Classroom Activity (Continued)

Join .. On
SELECT e.employee_id, e.last_name, e.salary, j.grade_level
FROM employees e JOIN job_grades j
ON (e.salary BETWEEN j.lowest_sal AND j.highest_sal);

This displays the grade level for each employee based on salary.
Try It / Solve It

Classroom Activity (Continued)

Outer Joins

Right Outer Join
SELECT e.employee_id, e.last_name,
    e.department_id, d.department_name
FROM employees e RIGHT OUTER JOIN departments d
ON (e.department_id = d.department_id);

Retrieves all data in the right table (DEPARTMENTS), including nulls.
Returns departments with employees and departments without employees.

Left Outer Join
SELECT e.employee_id, e.last_name,
    e.department_id, d.department_name
FROM employees e LEFT OUTER JOIN departments d
ON (e.department_id = d.department_id);

Retrieves all data in the left table (EMPLOYEES). Returns employees who
are assigned to a department as well as employees who are not assigned
to a department.
Full Outer Join

```
SELECT e.employee_id, e.last_name,
     e.department_id, d.department_name
FROM employees e FULL OUTER JOIN departments d
ON (e.department_id = d.department_id);
```

Retrieves all data in the left table and all data in the right table. This includes departments with employees and departments without employees. It also includes employees who are assigned to a department as well as employees who are not assigned to a department.
Summary

Objectives Summarized
In this lesson you have learned to:

• Determine the correct join syntax to use given a scenario requiring the join of data from two or more tables
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Group Functions
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Define and give an example of the seven group functions: SUM, AVG, COUNT, MIN, MAX, STDDEV, VARIANCE
• Construct and execute a SQL query using group functions
• Construct and execute group functions that operate only with numeric data types
Why Learn It?

Purpose
What if you were writing an article for the school newspaper and, to make a point, you wanted to know the average age of the students at your school?

What would you have to do to get this information? You could ask all students for their age in years, months, and days, add up all of these numbers, and then divide by the number of students in your school. That would be one way -- a very slow and difficult way -- to find this information.

What if you needed to know this now so that you could meet a 3:00 p.m. deadline? You might have a problem!
Why Learn It?

Purpose (continued)

What if all of the students’ dates of birth were in a school database in the STUDENT table? It would be so easy then!

In this lesson, you are going to learn about the power of group functions in SQL.
GROUP Functions
In SQL, the following group functions can operate on a whole table or on a specific grouping of rows. Each function returns one result.

AVG
COUNT
MIN
MAX
SUM
VARIANCE
STDDEV
Tell Me / Show Me

GROUP Functions List

**MIN**: Used with columns that store any data type to return the minimum value.

**MAX**: Used with columns that store any data type to return the maximum value.

**SUM**: Used with columns that store numeric data to find the total or sum of values.

**AVG**: Used with columns that store numeric data to compute the average.

```
SELECT MAX(salary)
FROM employees;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>60</td>
<td>11000</td>
</tr>
<tr>
<td>60</td>
<td>8600</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>4400</td>
</tr>
</tbody>
</table>

MAX (SALARY)

24000
**Tell Me / Show Me**

**GROUP Functions List (continued)**

**COUNT**: Returns the number of rows.

**VARIANCE**: Used with columns that store numeric data to calculate the spread of data around the mean. For example, if the average grade for the class on the last test was 82% and the student's scores ranged from 40% to 100%, the variance of scores would be greater than if the student's scores ranged from 78% to 88%.

**STDDEV**: Similar to variance, standard deviation measures the spread of data. For two sets of data with approximately the same mean, the greater the spread, the greater the standard deviation.

```sql
SELECT STDDEV(salary) 
FROM employees;
```

---

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>60</td>
<td>11000</td>
</tr>
<tr>
<td>60</td>
<td>8600</td>
</tr>
<tr>
<td>60</td>
<td>7000</td>
</tr>
<tr>
<td>10</td>
<td>4400</td>
</tr>
</tbody>
</table>

**STDDEV (SALARY)**

5659.633
GROUP Functions SELECT Clause

Group functions are written in the SELECT clause:

```
SELECT column, group_function(column), ..
FROM table
WHERE condition
GROUP BY column;
```

What are Group Functions?
Group Functions operate on sets of rows to give one result per group.

The maximum salary in the EMPLOYEES table

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>60</td>
<td>10500</td>
</tr>
<tr>
<td>60</td>
<td>11000</td>
</tr>
<tr>
<td>60</td>
<td>8600</td>
</tr>
<tr>
<td>60</td>
<td>7000</td>
</tr>
<tr>
<td>10</td>
<td>4400</td>
</tr>
</tbody>
</table>

MAX (SALARY) 24000
Tell Me / Show Me

GROUP Function Cautions

There are a few important things you should know about group functions:

Group functions cannot be used in the WHERE clause:

```
SELECT type_code
FROM d_songs
WHERE SUM (duration) = 100;
```

ORA-00934: group function is not allowed here
Tell Me / Show Me

GROUP Function and NULL

Group functions ignore NULL values. In the example below, the (null) values were not used to find the average overtime rate.

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>OVERTIME_RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>10.25</td>
</tr>
<tr>
<td>Bob</td>
<td>(null)</td>
</tr>
<tr>
<td>Monique</td>
<td>(null)</td>
</tr>
</tbody>
</table>

```
SELECT AVG(overtime_rate)
FROM f_staffs;
```

<table>
<thead>
<tr>
<th>AVG(OVERTIME_RATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.25</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

More Than One Group Function
You can have more than one group function in the SELECT clause, on the same or different columns.

You can also restrict the group function to a subset of the table using a WHERE clause.

```
SELECT MAX(salary), MIN(salary), MIN(employee_id)
FROM employees
WHERE department_id = 60;
```

<table>
<thead>
<tr>
<th>MAX(salary)</th>
<th>MIN(salary)</th>
<th>MIN(employee_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000</td>
<td>4200</td>
<td>103</td>
</tr>
</tbody>
</table>
**Tell Me / Show Me**

**MIN and MAX Group Functions**

Two group functions, MIN and MAX, can be used with any data type.

Using these functions, it is possible to find the name of the last person in a list, the smallest salary, or the earliest hire date.

For example, it is easy to find the person whose name is first in an alphabetical list of employees.

```
SELECT MIN(last_name)
FROM employees;
```

<table>
<thead>
<tr>
<th>MIN(LAST_NAME)</th>
<th>Abel</th>
</tr>
</thead>
</table>

Tell Me / Show Me

Rules for Group Functions

• Group functions ignore null values.
• Group functions cannot be used in the WHERE clause.
• MIN and MAX can be used with any data type; SUM, AVG, STDDEV and VARIANCE can be used only with numeric data types.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Aggregate
AVG
COUNT
Group functions
MAX
MIN
STDDEV
SUM
VARIANCE
Summary

Objectives Summarized
In this lesson you have learned to:

• Define and give an example of the seven group functions: SUM, AVG, COUNT, MIN, MAX, STDDEV, VARIANCE

• Construct and execute a SQL query using group functions

• Construct and execute group functions that operate only with numeric data types
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
COUNT, DISTINCT, NVL
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Construct and execute a SQL query using the COUNT group function
• Use DISTINCT and the NVL function with group functions
Why Learn It?

Purpose
Being able to aggregate data using SQL functions enables businesses to do calculations that would otherwise have to be done by hand.

Remember the example of having to count all the students in your school? A daunting task! There just aren't enough hands to accomplish it manually.

Fortunately, the SQL group functions can easily process these types of requests.
Tell Me / Show Me

COUNT
COUNT(expression) returns the number of non-null values in the expression column.

COUNT(DISTINCT expression) returns the number of unique non-null values in the expression column.

SELECT COUNT (YEAR) FROM d_cds WHERE year < 2001;

<table>
<thead>
<tr>
<th>COUNT (YEAR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

SELECT COUNT (DISTINCT year) FROM d_cds WHERE year < 2001;

<table>
<thead>
<tr>
<th>COUNT (DISTINCT YEAR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

COUNT and NULL Values
Why are null values returned in the query shown? There are six comments listed, but the count function returned only five. Why?

Because COUNT ignores the null value in the column.

<table>
<thead>
<tr>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play late</td>
</tr>
<tr>
<td>(null)</td>
</tr>
<tr>
<td>Play early</td>
</tr>
<tr>
<td>Play after cake cutting</td>
</tr>
<tr>
<td>Play first</td>
</tr>
<tr>
<td>Play for the father</td>
</tr>
</tbody>
</table>

```
SELECT comments
FROM d_play_list_items;
```

```
SELECT COUNT(comments)
FROM d_play_list_items;
```
**Tell Me / Show Me**

**COUNT All Rows**

COUNT(*) returns the number of rows in a table that satisfy the criteria of the SELECT statement.

For example, to find out how many of DJs on Demand's CDs were produced before 2001, COUNT can be used in the SELECT statement.

We use COUNT(*) when we want to make sure that we count all the rows, including those that may have nulls in one or more columns.

```
SELECT COUNT (*)
FROM d_cds
WHERE year < 2001;
```
**Tell Me / Show Me**

**DISTINCT**

The keyword DISTINCT is used to return only nonduplicate values or combinations of nonduplicate values in a query.

Examine the query on the right. Without using the keyword DISTINCT, the query returned all of the year values from the DJs on Demand D_CDS table.

```
SELECT year as 'CD Year'
FROM d_cds;
```

<table>
<thead>
<tr>
<th>CD Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2002</td>
</tr>
<tr>
<td>1999</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2001</td>
</tr>
<tr>
<td>1998</td>
</tr>
<tr>
<td>2004</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

DISTINCT Example
To eliminate duplicate rows, use the DISTINCT keyword as shown here.

Using the DISTINCT keyword returned all the CD years exactly once, with no duplicate values.

```
SELECT DISTINCT year AS 'CD Year'
FROM d_cds;
```

<table>
<thead>
<tr>
<th>CD Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
</tr>
<tr>
<td>1998</td>
</tr>
<tr>
<td>1999</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2001</td>
</tr>
<tr>
<td>2002</td>
</tr>
<tr>
<td>2004</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

DISTINCT Nonduplicate

The keyword DISTINCT, when used in a query selecting more than one column, will return nonduplicate combinations of the columns. Examine the two results sets shown here. Can you tell which query used the DISTINCT keyword?

In this case, it's hard to tell, isn't it? The results set on the top was returned using the DISTINCT keyword. In both examples, there are no duplicate combinations of year and title even though there are duplicate years.

```
YEAR  TITLE
1997  The Celebrants Live in Concert
1998  Graduation Songbook
1999  Songs from My Childhood
2000  Cape Diem
2000  Party Music for All Occasions
2001  Here Comes the Bride
2002  Back to the Shire
2004  Whirled Peas
```

```
SELECT DISTINCT year, title
FROM d_cds;
```

```
YEAR  TITLE
1997  The Celebrants Live in Concert
1999  Songs from My Childhood
2000  Carpe Diem
2001  Here Comes the Bride
1998  Graduation Songbook
2004  Whirled Peas
```
Tell Me / Show Me

Using DISTINCT

The keyword DISTINCT can be used with all group functions. Using DISTINCT makes the function consider only nonduplicate values.

Why do the two statements on the right produce different results?

- SELECT SUM(salary) FROM employees WHERE department_id = 90;
  - SALARY
    - 24000
    - 17000
  - SUM(SALARY) 58000

- SELECT SUM(DISTINCT salary) FROM employees WHERE department_id = 90;
  - SALARY
    - 24000
    - 17000
  - SUM(DISTINCT SALARY) 41000
Tell Me / Show Me

DISTINCT and COUNT
When using DISTINCT with a group function such as COUNT, the result set will return the number of nonduplicate column values.

How many different jobs do our employees do?

How many different salaries are there?

SELECT COUNT (DISTINCT job_id) FROM employees;

<table>
<thead>
<tr>
<th>COUNT (DISTINCT job_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

SELECT COUNT (DISTINCT salary) FROM employees;

<table>
<thead>
<tr>
<th>COUNT (DISTINCT salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

NVL

Sometimes it is desirable to include null values in group functions. For example, knowing the average number of customer orders served each day could be used to judge how much food to order each month. Some days the restaurant is closed and there are no customers, but the owner has found that computing the average including every day is a better indicator than just counting the days with customers. The SELECT statement to include null values could be written starting with:

```
SELECT AVG(NVL(customer_orders, 0))
```
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

COUNT (expression)
COUNT (DISTINCT expression)
DISTINCT
Summary

Objectives Summarized
In this lesson you have learned to:

• Construct and execute a SQL query using the COUNT group function
• Use DISTINCT and the NVL function with group functions
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Using Group By and Having Clauses
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Construct and execute a SQL query using GROUP BY
• Construct and execute a SQL query using GROUP BY … HAVING
• Construct and execute a GROUP BY on more than one column
• Nest group functions
Why Learn It?

Purpose
What if you wanted to know the average height of all students? You could write a query that looks like this:

```
SELECT AVG(height) FROM students;
```

But what if you wanted to know the average height of the students based on their year in school? Right now, you would have to write a number of different SQL statements to accomplish this:

```
SELECT AVG(height) FROM students WHERE year_in_school = 10;
SELECT AVG(height) FROM students WHERE year_in_school = 11;
SELECT AVG(height) FROM students WHERE year_in_school = 12;
```

And so on! To simplify problems like this with just one statement you use the GROUP BY and HAVING clauses.
Tell Me / Show Me

GROUP BY Use

You use the GROUP BY clause to divide the rows in a table into smaller groups. You can then use the group functions to return summary information for each group.

In the SELECT statement shown, the rows are being grouped by department_id. The AVG function is then automatically applied to each group.

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>AVG (SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>9500</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>60</td>
<td>6400</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

GROUP BY Example

What if we wanted to find the maximum salary of employees in each department? We use a GROUP BY clause stating which column to use to group the rows.

```
SELECT MAX(salary)
FROM employees
GROUP BY department_id;
```

But how can we tell which maximum salary belongs to which department?

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

MAX (SALARY)

24000
9000
5800
...
Tell Me / Show Me

GROUP BY in SELECT

Usually we want to include the GROUP BY column in the SELECT list.

SELECT department_id, MAX(salary)
FROM employees
GROUP BY department_id;
Tell Me / Show Me

GROUP BY Clause
Group functions require that any column listed in the SELECT clause that is not part of a group function must be listed in a GROUP BY clause.

What is wrong with this example?

SELECT job_id, last_name, AVG(salary)
FROM employees
GROUP BY job_id;

ORA-00979: not a GROUP BY expression
This example shows how many students wear shirts of each color. Remember that group functions ignore null values, so if any student does not have a first name, he or she will not be included in the COUNT. Of course this is unlikely, but when constructing SQL statements we have to think about all the possibilities.

It would be better to start with:

```
SELECT COUNT(*), shirt_color
FROM students
GROUP BY shirt_color
```
Tell Me / Show Me

WHERE Clause
We can also use a WHERE clause to exclude rows before the remaining rows are formed into groups.

```
SELECT department_id, MAX(salary)
FROM employees
WHERE last_name <> 'King'
GROUP BY department_id;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>Hunold</td>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>Ernst</td>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>Lorentz</td>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>MAX (SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

More GROUP BY Examples

1. Show the average graduation rate of the schools in several cities; include only those students who have graduated in the last few years

   SELECT AVG(graduation_rate), city
   FROM students
   WHERE graduation_date >= '01-JUN-07'
   GROUP BY city;

2. Count the number of students in the school, grouped by grade; include all students

   SELECT COUNT(first_name), grade
   FROM students
   GROUP BY grade;
Tell Me / Show Me

GROUP BY Guidelines
Important guidelines to remember when using a GROUP BY clause are:

• If you include a group function (AVG, SUM, COUNT, MAX, MIN, STDDEV, VARIANCE) in a SELECT clause and any other individual columns, each individual column must also appear in the GROUP BY clause.
• You cannot use a column alias in the GROUP BY clause.
• The WHERE clause excludes rows before they are divided into groups.
Tell Me / Show Me

Groups Within GROUPS

Sometimes you need to divide groups into smaller groups. For example, you may want to group all employees by department; then, within each department, group them by job.

This example shows how many employees are doing each job within each department.

```
SELECT department_id, job_id, count(*)
FROM employees
WHERE department_id > 40
GROUP BY department_id, job_id;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>JOB_ID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>ST_MAN</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>4</td>
</tr>
<tr>
<td>60</td>
<td>IT_PROG</td>
<td>3</td>
</tr>
<tr>
<td>80</td>
<td>SA_MAN</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>SA_REP</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Nesting Group Functions

Group functions can be nested to a depth of two when GROUP BY is used.

```
SELECT max(avg(salary))
FROM employees
GROUP by department_id;
```

How many values will be returned by this query? The answer is one – the query will find the average salary for each department, and then from that list, select the single largest value.
Tell Me / Show Me

HAVING

Suppose we want to find the maximum salary in each department, but only for those departments which have more than one employee? What is wrong with this example?

SELECT department_id, MAX(salary)
FROM employees
WHERE COUNT(*) > 1
GROUP BY department_id;

ORA-00934: group function is not allowed here
HAVING (continued)

In the same way you used the WHERE clause to restrict the rows that you selected, you can use the HAVING clause to restrict groups.

In a query using a GROUP BY and HAVING clause, the rows are first grouped, group functions are applied, and then only those groups matching the HAVING clause are displayed.

The WHERE clause is used to restrict rows; the HAVING clause is used to restrict groups returned from a GROUP BY clause.

```
SELECT department_id, MAX(salary)
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 1;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MAX(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

HAVING (continued)
Although the HAVING clause can precede the GROUP BY clause in a SELECT statement, it is recommended that you place each clause in the order shown. The ORDER BY clause (if used) is always last!

SELECT column, group_function
FROM table
WHERE
GROUP BY
HAVING
ORDER BY
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

GROUP BY
HAVING
Summary

Objectives Summarized
In this lesson you have learned to:

- Construct and execute a SQL query using GROUP BY
- Construct and execute a SQL query using GROUP BY … HAVING
- Construct and execute a GROUP BY on more than one column
- Nest group functions
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Using Rollup and Cube Operations, and Grouping Sets
What Will I Learn?

Objectives

In this lesson, you will learn to:

- Use ROLLUP to produce subtotal values
- Use CUBE to produce cross-tabulation values
- Use GROUPING SETS to produce a single result set
- Use the GROUPING function to identify the extra row values created by either a ROLLUP or CUBE operation
Why Learn It?

Purpose
What if you wanted to know the average height of all students? You could write a query that looks like this:
SELECT AVG(height) FROM students;

But what if you wanted to know the average height of the students based on their year in school? Right now, you would have to write a number of different SQL statements to accomplish this:
SELECT AVG(height) FROM students WHERE year_in_school = 10;
SELECT AVG(height) FROM students WHERE year_in_school = 11;
SELECT AVG(height) FROM students WHERE year_in_school = 12;

And so on! To simplify problems like this with just one statement you use the GROUP BY and HAVING clauses.
Why Learn It?

Purpose (continued)

What if, once you have selected your groups and computed your aggregates across these groups, you also wanted subtotals per group and a grand total of all the rows selected.

You could import the results into a spreadsheet application, or you could get out your calculator, or you could compute the totals using manual arithmetic. Or you could use some of the extensions to the GROUP BY clause: ROLLUP, CUBE and GROUPING SETS.

Using these extensions requires less work on your part and they are all highly efficient to use, from the point of view of the database.
Tell Me / Show Me

ROLLUP

In GROUP BY queries you are quite often required to produce subtotals and totals, and the ROLLUP operation can do that for you.

The action of ROLLUP is straightforward: it creates subtotals that roll up from the most detailed level to a grand total. ROLLUP uses an ordered list of grouping columns as its argument.

- First, it calculates the standard aggregate values specified in the GROUP BY clause.
- Next, it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns.
- Finally, it creates a grand total.
Tell Me / Show Me

ROLLUP Result Table

In the result table below, the rows highlighted in red are generated by the ROLLUP operation:

```
SELECT department_id, job_id, SUM(salary)
FROM employees
WHERE department_id < 50
GROUP BY ROLLUP (department_id, job_id)
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>19000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23400</td>
</tr>
</tbody>
</table>

- Subtotal for dept_id 10
- Subtotal for dept_id 20
- Grand Total for report
**Tell Me / Show Me**

**ROLLUP Result Formula**

The number of columns or expressions that appear in the ROLLUP clause determine the number of groupings. The formula is \((\text{number of columns}) + 1\) where the number of columns listed is the number of columns listed in the ROLLUP clause. In the example query below there are 2 columns listed in the ROLLUP clause and therefore you will see 3 values generated automatically.

SELECT department_id, job_id, SUM(salary)
FROM employees
WHERE department_id < 50
GROUP BY ROLLUP (department_id, job_id)
Tell Me / Show Me

Without ROLLUP
If you use GROUP BY without ROLLUP for the same query what would the results look like?

SELECT department_id, job_id, SUM(salary)
FROM   employees
WHERE department_id < 50
GROUP BY (department_id, job_id)

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
</tbody>
</table>

You would have to execute multiple queries to get the subtotals produced by ROLLUP.
CUBE

CUBE is an extension to the GROUP BY clause like ROLLUP. It produces cross-tabulation reports.

It can be applied to all aggregate functions including AVG, SUM, MIN, MAX and COUNT.

Columns listed in the GROUP BY clause are cross-referenced to create a superset of groups. The aggregate functions specified in the SELECT list are applied to this group to create summary values for the additional super-aggregate rows. Every possible combination of rows is aggregated by CUBE. If you have \( n \) columns in the GROUP BY clause, there will be \( 2^n \) possible super-aggregate combinations. Mathematically these combinations form an \( n \)-dimensional cube, which is how the operator got its name.
Tell Me / Show Me

CUBE (Continued)

CUBE is typically most suitable in queries that use columns from multiple tables rather than columns representing different rows of a single table.

Imagine for example a user querying the Sales table for a company like AMAZON.COM. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of Month, Region and Product.

These are three independent tables, and analysis of all possible subtotal combinations is commonplace. In contrast, a cross-tabulation showing all possible combinations of year, month and day would have several values of limited interest, because there is a natural hierarchy in the time table. Subtotals such as profit by day of month summed across year would be unnecessary in most analyses. Relatively few users need to ask "What were the total sales for the 16th of each month across the year?"
CUBE (Continued)

In the following statement the rows in red are generated by the CUBE operation:

```sql
SELECT department_id, job_id, SUM(salary)
FROM employees
WHERE department_id < 50
GROUP BY CUBE (department_id, job_id)
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>23400</td>
</tr>
<tr>
<td>-</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>-</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>-</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>19000</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
</tbody>
</table>

Total for report
Subtotal for MK_MAN
Subtotal for MK_REP
Subtotal for AD_ASST
Subtotal for dept 10
Subtotal for dept 20
GROUPING SETS

GROUPING SETS is another extension to the GROUP BY clause, like ROLLUP and CUBE. It is used to specify multiple groupings of data. It is like giving you the possibility to have multiple GROUP BY clauses in the same SELECT statement, which is not allowed in the syntax.

The point of GROUPING SETS is that if you want to see data from the EMPLOYEES table grouped by (department_id, job_id, manager_id), but also by (department_id, manager_id) and also by (job_id, manager_id) then you would normally have to write 3 different select statements with the only difference being the GROUP BY clauses. For the database this means retrieving the same data in this case 3 times, and that can be quite a big overhead. Imagine if your company had 3,000,000 employees. Then you are asking the database to retrieve 9 million rows instead of just 3 million rows – quite a big difference.

So GROUPING SETS are much more efficient when writing complex reports.
Tell Me / Show Me

GROUPING SETS (continued)

In the following statement the rows highlighted in color are generated by the GROUPING SETS operation:

SELECT department_id, job_id, manager_id, SUM(salary)
FROM employees
WHERE department_id < 50
GROUP BY GROUPING SETS
((job_id, manager_id), (department_id, job_id), (department_id, manager_id))

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MK_MAN</td>
<td>100</td>
<td>13000</td>
</tr>
<tr>
<td></td>
<td>MK_MAN</td>
<td>201</td>
<td>6000</td>
</tr>
<tr>
<td></td>
<td>AD_ASST</td>
<td>101</td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>-</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>-</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>-</td>
<td>6000</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>101</td>
<td>19000</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>100</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>201</td>
<td>6000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

GROUPING Functions

When you create reports with subtotals, using either ROLLUP or CUBE you quite often also have to be able to tell which rows in the output are actual rows returned from the database and which rows are computed subtotal rows, resulting from the ROLLUP or CUBE operations.

If you look at the report on the right, how will you be able to differentiate between the actual database rows and the calculated rows?

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
<td>23400</td>
</tr>
<tr>
<td>-</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>-</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>-</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>19000</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
</tbody>
</table>
GROUPING Functions (continued)

You may also need to find the exact level of aggregation for a given subtotal. You often need to use subtotals in calculations such as percent-of-totals, so you need an easy way to determine which rows are the subtotals.

You might also be presented with a problem when you are trying to tell the difference between a stored NULL value returned by the query and "NULL" values created by a ROLLUP or CUBE. How can you differentiate between the two?
GROUPING Functions (continued)

The GROUPING function handles these problems. Using a single column from the query as its argument, GROUPING returns 1 when it encounters a NULL value created by a ROLLUP or CUBE operation. That is, if the NULL indicates the row is a subtotal, GROUPING returns a 1. Any other type of value, including a stored NULL, returns a 0. So the GROUPING function will return a 1 for an aggregated (computed) row and a 0 for a not aggregated (returned) row.

The syntax for the GROUPING is simply GROUPING (column_name). It is used only in the SELECT clause and it takes only a single column expression as argument.
Tell Me / Show Me

GROUPING Functions (continued)

```sql
SELECT department_id, job_id, SUM(salary),
    GROUPING(department_id) Dept_sub_total,
    DECODE(GROUPING(department_id),
        1,'Dept Aggregate row', department_id) AS DT,
    GROUPING(job_id) job_sub_total,
    DECODE(GROUPING(job_id),
        1,'JobID Aggregate row',job_id) AS JI
FROM employees
WHERE department_id < 50
GROUP BY CUBE (department_id, job_id)
```
GROUPING Functions (continued)

The output below is from the query on the previous slide, and it shows the use of the GROUPING function along with a DECODE to translate the 0 and 1 returned by GROUPING into either a text string or the actual table data:

```
DECODE(GROUPING(job_id), 1,'JobID Aggregate row',job_id) AS JI
DECODE(GROUPING(department_id),1,'Dept Aggregate row', department_id) AS DT,
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
<th>DEPT_SUB_TOTAL</th>
<th>DT</th>
<th>JOB_SUB_TOTAL</th>
<th>JI</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>23400</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>1</td>
<td>JobID Aggregate row</td>
</tr>
<tr>
<td>-</td>
<td>MK_MAN</td>
<td>13000</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>MK_MAN</td>
</tr>
<tr>
<td>-</td>
<td>MK_REP</td>
<td>6000</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>MK_REP</td>
</tr>
<tr>
<td>-</td>
<td>AD_ASST</td>
<td>4400</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>AD_ASST</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>4400</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>JobID Aggregate row</td>
</tr>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>AD_ASST</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>19000</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>JobID Aggregate row</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>MK_MAN</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>MK_REP</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

GROUPING Functions (continued)

Quite often queries like these are used as a basis for reporting these numbers in graphical reporting tools, and all sorts of charts are created from the data to present the results in an easy to understand format to the end user. To save the end-user query tools having to do the summaries and calculations, you get that work done by the database, but if you total all the numbers below you would get a very wrong result. The total sum(salary) for employees working in department_id < 50 is only 23400, not 93600, which is the total of all the rows returned by the query. So you need to be able to tell the real rows apart from the aggregated rows, and the 0 and 1 makes that job really easy.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
<th>DEPT_SUB_TOTAL</th>
<th>DT</th>
<th>JOB_SUB_TOTAL</th>
<th>JI</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>23400</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>1</td>
<td>JobID Aggregate row</td>
</tr>
<tr>
<td>-</td>
<td>MK_MAN</td>
<td>13000</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>MK_MAN</td>
</tr>
<tr>
<td>-</td>
<td>MK_REP</td>
<td>6000</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>MK_REP</td>
</tr>
<tr>
<td>-</td>
<td>AD_ASST</td>
<td>4400</td>
<td>1</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>AD_ASST</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>4400</td>
<td>0</td>
<td>Dept Aggregate row</td>
<td>1</td>
<td>JobID Aggregate row</td>
</tr>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
<td>0</td>
<td>Dept Aggregate row</td>
<td>0</td>
<td>AD_ASST</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>19000</td>
<td>0</td>
<td>Dept Aggregate row</td>
<td>20</td>
<td>JobID Aggregate row</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
<td>0</td>
<td>Dept Aggregate row</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
<td>0</td>
<td>Dept Aggregate row</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>
GROUPING Functions (continued)

If the result of the query on the previous slide were to be imported into a spreadsheet and graphed without any filters applied it would result in the following graph, which is wrong, as it includes the same numbers multiple times.
GROUPING Functions (continued)

Here is the same query displayed with filters applied to only include rows with a 0 in the GROUPING(department_id) or GROUPING(job_id). In this report no number is included more than once. With the 0 and 1 from the GROUPING function present, it is very easy to filter out the subtotals.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

ROLLUP
CUBE
GROUPING SETS
GROUPING FUNCTION
Summary

Objectives Summarized
In this lesson you have learned to:

- Use ROLLUP to produce subtotal values
- Use CUBE to produce cross-tabulation values
- Use GROUPING SETS to produce a single result set
- Use the GROUPING function to identify the extra row values created by either a ROLLUP or CUBE operation
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Using Set Operators
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Define and explain the purpose of Set Operators
• Use a set operator to combine multiple queries into a single query
• Control the order of rows returned using set operators
Why Learn It?

Purpose
Set operators are used to combine the results from different SELECT statements into one single result output.

Sometimes you want a single output from more than one table. If you join the tables, the rows that match are returned, but what if you don’t want to do a join, or can’t do a join because a join will give the wrong result?

This is where SET operators come in. They can return the rows found in both statements, the rows that are in one table and not the other, or the rows common to both statements.
Tell Me / Show Me

Setting the Stage

In order to explain the SET operators the following two lists will be used throughout this lesson:

A = \{1, 2, 3, 4, 5\}
B = \{4, 5, 6, 7, 8\}

Or in reality: two tables, one called A and one called B.

<table>
<thead>
<tr>
<th>A_ID</th>
<th>B_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Rules to Remember

There are a few rules to remember when using SET operators:

• The number of columns and the data types of the columns must be identical in all of the SELECT statements used in the query.

• The names of the columns need not be identical.

• Column names in the output are taken from the column names in the first SELECT statement. So any column aliases should be entered in the first statement as you would want to see them in the finished report.
**UNION**

The UNION operator returns all rows from both tables, after eliminating duplicates.

```sql
SELECT a_id
FROM    a
UNION
SELECT b_id
FROM    b;
```

The result of listing all elements in A and B eliminating duplicates is \{1, 2, 3, 4, 5, 6, 7, 8\}.

If you joined A and B you would get only \{4, 5\}. You would have to perform a full outer join to get the same list as above.
UNION ALL
The UNION ALL operator returns all rows from both tables, without eliminating duplicates.

SELECT a_id
FROM a
UNION ALL
SELECT b_id
FROM b;

The result of listing all elements in A and B without eliminating duplicates is \{1, 2, 3, 4, 5, 4, 5, 6, 7, 8\}.

Tell Me / Show Me
**Tell Me / Show Me**

**INTERSECT**  
The INTERSECT operator returns all rows common to both tables.

```
SELECT a_id  
FROM    a
INTERSECT
SELECT b_id  
FROM    b;
```

The result of listing all elements found in both A and B is \{4, 5\}.  

![Intersection Venn Diagram](image)
MINUS
The MINUS operator returns all rows found in one table but not the other.

SELECT a_id
FROM a
MINUS
SELECT b_id
FROM b;

The result of listing all elements found in A but not B is \{1, 2, 3\}, and B MINUS A would give \{6, 7, 8\}.
Tell Me / Show Me

Set Operator Examples
Sometimes if you are selecting rows from tables that do not have columns in common, you may have to make up columns in order to match the queries. The easiest way to do this is to include one or more NULL values in the select list. Remember to give them suitable aliases and matching datatypes.
For example:
Table A contains a location id and a department name.
Table B contains a location id and a warehouse name.
You can use the TO_CHAR(NULL) function to fill in the missing columns as shown below.

SELECT location_id, department_name "Department", TO_CHAR(NULL) "Warehouse"
FROM departments
UNION
SELECT location_id, TO_CHAR(NULL) "Department", warehouse_name
FROM warehouses;
Tell Me / Show Me

Set Operator Examples (continued)

The keyword NULL can be used to match columns in a SELECT list. One NULL is included for each missing column. Furthermore, NULL is formatted to match the datatype of the column it is standing in for, so TO_CHAR, TO_DATE or TO_NUMBER functions are often used to achieve identical SELECT lists.
Tell Me / Show Me

Set Operator Examples (continued)
The WAREHOUSES table description:

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Data Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Primary Key</th>
<th>Nullable</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAREHOUSES</td>
<td>LOCATION_ID</td>
<td>Number</td>
<td>6</td>
<td>0</td>
<td></td>
<td>✔</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>WAREHOUSES</td>
<td>WAREHOUSE_NAME</td>
<td>Varchar2</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>✔</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

The WAREHOUSES table data:

<table>
<thead>
<tr>
<th>LOCATION_ID</th>
<th>WAREHOUSE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1700</td>
<td>London</td>
</tr>
<tr>
<td>1800</td>
<td>Paris</td>
</tr>
<tr>
<td>2100</td>
<td>Copenhagen</td>
</tr>
<tr>
<td>4000</td>
<td>Shanghai</td>
</tr>
<tr>
<td>3000</td>
<td>Atlanta</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Set Operator Examples (continued)

Select location_id, department_name "Department", TO_CHAR(NULL) "Warehouse"
FROM Departments
UNION
SELECT location_id, TO_CHAR(NULL) "Department", warehouse_name
FROM warehouses;
SET Operations ORDER BY

If you want to control the order of the returned rows when using SET operators in your query, the ORDER BY statement must only be used once, in the last SELECT statement in the query. Using the example on the previous slide, if you need to sort the rows returned by employee_id, you would need to add an ORDER BY.

```sql
SELECT hire_date, employee_id, to_date(null) start_date, to_date(null) end_date, job_id, department_id
FROM employees
UNION
SELECT to_date(null), employee_id, start_date, end_date, job_id, department_id
FROM job_history
```

A partial query result is shown. Notice the rows from EMPLOYEES and JOB_HISTORY for the same employees are not displayed in any order. By default the first column in the first query is used to sort the rows.
SET Operations ORDER BY (continued)

The same query ordered by `employee_id` gives the following result, and on this output you can see the entire employment history of one individual employee without scrolling up and down.

```sql
SELECT hire_date, employee_id,
to_date(null) start_date, to_date(null) end_date, job_id, department_id
FROM employees
UNION
SELECT to_date(null), employee_id,
start_date, end_date, job_id, department_id
FROM job_history
ORDER BY employee_id
```

<table>
<thead>
<tr>
<th>HIRE_DATE</th>
<th>EMPLOYEE_ID</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>17-JUN-87</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>AD_PRES</td>
<td>90</td>
</tr>
<tr>
<td>21-SEP-89</td>
<td>101</td>
<td>-</td>
<td>-</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>21-SEP-89</td>
<td>27-OCT-93</td>
<td>AC_ACCOUNT</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>28-OCT-93</td>
<td>15-MAR-97</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>13-JAN-93</td>
<td>102</td>
<td>-</td>
<td>-</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>13-JAN-93</td>
<td>24-JUL-98</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>03-JAN-90</td>
<td>103</td>
<td>-</td>
<td>-</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>21-MAY-91</td>
<td>104</td>
<td>-</td>
<td>-</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>07-FEB-99</td>
<td>107</td>
<td>-</td>
<td>-</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>114</td>
<td>24-MAR-98</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>122</td>
<td>01-JAN-99</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>16-NOV-99</td>
<td>124</td>
<td>-</td>
<td>-</td>
<td>ST_MAN</td>
<td>50</td>
</tr>
<tr>
<td>17-OCT-95</td>
<td>141</td>
<td>-</td>
<td>-</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>29-JAN-97</td>
<td>142</td>
<td>-</td>
<td>-</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>15-MAR-98</td>
<td>143</td>
<td>-</td>
<td>-</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>09-JUL-98</td>
<td>144</td>
<td>-</td>
<td>-</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>29-JAN-00</td>
<td>149</td>
<td>-</td>
<td>-</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>11-MAY-96</td>
<td>174</td>
<td>-</td>
<td>-</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td>24-MAR-98</td>
<td>175</td>
<td>-</td>
<td>-</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>175</td>
<td>24-MAR-98</td>
<td>31-DEC-98</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>176</td>
<td>01-JAN-99</td>
<td>31-DEC-99</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
</tbody>
</table>
```

The ORDER BY used in SET operator queries can only be used in the LAST statement, and only columns from the first statement are valid in the ORDER BY clause.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

SET operators
UNION
UNION ALL
INTERSECT
MINUS
TO_CHAR(null) – matching the select list
Summary

Objectives Summarized
In this lesson you have learned to:

• Define and explain the purpose of Set Operators
• Use a set operator to combine multiple queries into a single query
• Control the order of rows returned using set operators
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Fundamentals of Subqueries
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Define and explain the purpose of subqueries for retrieving data
• Construct and execute a single-row subquery in the WHERE clause
• Distinguish between single-row and multiple-row subqueries
• Distinguish between pair-wise and non-pair-wise subqueries
• Use the EXISTS and NOT EXISTS operators in a query
Why Learn It?

Purpose
Has a friend asked you to go to a movie, but before you could answer "yes" or "no," you first had to check with your parents? Has someone asked you the answer to a math problem, but before you can give the answer, you had to do the problem yourself?

Asking parents, or doing the math problem, are examples of subqueries.

In SQL, subqueries enable us to find the information we need so we can get the information we want.
Subquery Overview

Throughout this course, you have written queries to extract data from a database. What if you wanted to write a query only to find out you didn't have all the information you needed to construct it?

You can solve this problem by combining two queries, placing one query inside the other query. The inner query is called the "subquery." The subquery executes to find the information you don’t know. The outer query uses that information to find out what you need to know.

Being able to combine two queries into one can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.
Subquery Example

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. A subquery executes once before the main query. The result of the subquery is used by the main or outer query.

Subqueries can be placed in a number of SQL clauses, including the WHERE clause, the HAVING clause, and the FROM clause.

The subquery syntax is:

```
SELECT select_list
FROM table
WHERE expression operator
(SELECT select_list
FROM table);
```

The SELECT statement in parentheses is the inner query or ‘subquery’. It executes first, before the outer query.
Tell Me / Show Me

Guidelines for using Subqueries

- The subquery is enclosed in parentheses.
- The subquery is placed on the right side of the comparison condition.
- The outer and inner queries can get data from different tables.
- Only one ORDER BY clause can be used for a SELECT statement; and, if used, it must be the last clause in the outer query. A subquery cannot have its own ORDER BY clause.
- The only limit on the number of subqueries is the buffer size the query uses.
Tell Me / Show Me

Two types of Subqueries

- **Single-row subqueries** that use single-row operators (>, =, >=, <, <>, <=) and return only one row from the inner query.

- **Multiple-row subqueries** that use multiple-row operators (IN, ANY, ALL) and return more than one row from the inner query.
### Tell Me / Show Me

#### Subquery Example

What if you wanted to find out the names of the Global Fast Foods staff members that were born after Monique Tuttle? What is the first thing you need to know? When was Monique born? Once you know her birth date, then you can select those staff members whose birth dates are after hers.

```sql
SELECT staff_id, first_name, last_name, birth_date
FROM f_staffs
WHERE birth_date >=
    (SELECT birth_date
     FROM f_staffs
     WHERE last_name = 'Tuttle');
```

<table>
<thead>
<tr>
<th>STAFF_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>BIRTH_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Roger</td>
<td>Morgan</td>
<td>17-JUN-87</td>
</tr>
<tr>
<td>1189</td>
<td>Nancy</td>
<td>Vickers</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>1007</td>
<td>Monique</td>
<td>Tuttle</td>
<td>13-JAN-87</td>
</tr>
<tr>
<td>1354</td>
<td>Alex</td>
<td>Hunter</td>
<td>03-JAN-90</td>
</tr>
<tr>
<td>1423</td>
<td>Kathryn</td>
<td>Bassman</td>
<td>08-AUG-91</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Subquery and Null
If a subquery returns a null value or no rows, the outer query takes the results of the subquery (null) and uses this result in its WHERE clause.

The outer query will then return no rows, because comparing a value with null always yields a null.

Who works in the same department as Grant? Grant’s department_id is null.

The outer query does not even return Grant’s row, because comparing a null with a null returns a null.
Tell Me / Show Me

Multiple-Column Subqueries

Subqueries can use one or more columns. If they use more than one column, they are called multiple-column subqueries. A multiple-column subquery can be either pair-wise comparisons or non-pair-wise comparisons.

The example on the right shows a multiple-column pair-wise subquery with the subquery highlighted in red and the result in the table below.

The query lists the employees whose manager and departments are the same as the manager and department of employees 149 or 174.

```
SELECT employee_id,
       manager_id,
       department_id
FROM employees
WHERE (manager_id,department_id) IN
      (SELECT manager_id,department_id
       FROM employees
       WHERE employee_id IN (149,174))
AND     employee_id NOT IN (149,174)
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>149</td>
<td>80</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Multiple-Column Subqueries (continued)

A non-pair-wise multiple-column subquery also uses more than one column in the subquery, but it compares them one at a time, so the comparisons take place in different subqueries. You will need to write one subquery per column you want to compare against when performing non-pair-wise multiple column subqueries. The example on the right shows a multiple-column non-pair-wise subquery with the subqueries highlighted in red. This query is listing the employees who have either a manager_id or a department_id in common with employees 174 or 199.

```
SELECT employee_id,
       manager_id,
       department_id
FROM employees
WHERE manager_id IN
    (SELECT manager_id
     FROM employees
     WHERE employee_id IN (174,199))
AND department_id IN
    (SELECT department_id
     FROM employees
     WHERE employee_id IN (174,199))
AND employee_id NOT IN (174,199);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>149</td>
<td>80</td>
</tr>
<tr>
<td>149</td>
<td>100</td>
<td>80</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

EXISTS & NOT EXISTS in Subqueries

EXISTS and its opposite, NOT EXISTS, are another two clauses that can be used when testing for matches in subqueries. EXISTS test for a TRUE, or a matching result result in the subquery.

If you want to see how many employees are employees and not managers, you could use NOT EXISTS:

```sql
SELECT count(*)
FROM   employees t1
WHERE  NOT EXISTS (SELECT
                    NULL
FROM   employees t2
WHERE  t2.manager_id = t1.employee_id );
```

In this example the subquery is selecting a NULL value to emphasize the fact that we are testing for an occurrence of rows in the subquery, we are not wanting to return anything but TRUE or FALSE from it.
EXISTS & NOT EXISTS in Subqueries (continued)

If the same query is executed with a NOT IN instead of NOT EXISTS, the result is very different. The result of this query suggests there are no employees who are also not managers, so all employees are managers, which we already know is not true. What is causing this result?

```
SELECT count(*)
FROM   employees t1
WHERE  t1.employee_id NOT IN (SELECT t2.manager_id
FROM   employees t2
);
```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
EXISTS & NOT EXISTS in Subqueries (continued)

The cause of the strange result is due to the NULL value returned by the subquery. One of the rows in the employees table does not have a manager, and this makes the entire result wrong. Subqueries can return three values: TRUE, FALSE and UNKNOWN. A NULL in the subquery result set will return an UNKNOWN, which Oracle cannot evaluate, so it doesn’t.

```sql
SELECT count(*)
FROM  employees t1
WHERE  t1.employee_id NOT IN (SELECT t2.manager_id
                                 FROM  employees t2);
```

BEWARE of NULLS in subqueries when using NOT IN. If you do not know if the subquery will return null values, always use NOT EXISTS. It is safer.
EXISTS & NOT EXISTS in Subqueries (continued)

The way EXISTS and NOT EXISTS are executed by Oracle is very different from the way IN and NOT IN are executed.

When the database is executing an IN with a Subquery, it evaluates the Subquery, typically Distincts that query and then joins the result to the outer query to get the result.

When an EXISTS is executed it performs a Full Table Scan of the outer table, and then Oracle loops through the Subquery result rows one by one to see if the condition is true. It executes like this:

```sql
FOR x IN ( SELECT * FROM employees t1 ) LOOP
    IF ( EXISTS ( SELECT NULL FROM employees t2 WHERE t1.employee_id = t2.manager_id ) ) THEN
        OUTPUT THE RECORD
    END IF
END IF
END LOOP
```
Tell Me / Show Me

EXISTS & NOT EXISTS in Subqueries (continued)

One final note about EXISTS and IN.

IN will generally execute faster, as it can use any existing indexes on the outer table. EXISTS cannot use indexes on the outer table. The size of the table is also an important factor and affects which of the expressions will run faster.

A small outer table joined to a very big inner table can still execute very fast using EXISTS, perhaps even faster than the same statement using IN.

And remember: be careful when handling NULLs.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Inner query
Multiple-row subquery
Outer subquery
Single row subquery
Subquery
Pair-wise multiple column subquery
Non-pair-wise multiple column subquery
EXIST and NOT EXIST
Summary

Objectives Summarized
In this lesson you have learned to:

• Define and explain the purpose of subqueries for retrieving data
• Construct and execute a single-row subquery in the WHERE clause
• Distinguish between single-row and multiple-row subqueries
• Distinguish between pair-wise and non-pair-wise subqueries
• Use the EXIST and NOT EXISTS operators in a query
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Single-Row Subqueries
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Construct and execute a single-row subquery in the WHERE clause or HAVING clause
• Construct and execute a SELECT statement using more than one subquery
• Construct and execute a SELECT statement using a group function in the subquery
Why Learn It?

Purpose

As you have probably realized, subqueries are a lot like Internet search engines. They are great at locating the information needed to accomplish another task.

In this lesson, you will learn how to create even more complicated tasks for subqueries to do for you. Keep in mind that subqueries save time in that you can accomplish two tasks in one statement.
Tell Me / Show Me

Facts about single-row subqueries

They:

• Return only one row
• Use single-row comparison operators (=, >, >=, <, <=, <>)

Always:

• Enclose the subquery in parentheses
• Place the subquery on the right hand side of the comparison condition.
Tell Me / Show Me

Additional Subquery Facts

• The outer and inner queries can get data from different tables.
• Only one ORDER BY clause can be used for a SELECT statement, and if specified, it must be the last clause in the main SELECT statement.
• The only limit on the number of subqueries is the buffer size that the query uses.
Tell Me / Show Me

Subqueries from Different Tables

The outer and inner queries can get data from different tables.

Who works in the Marketing department?

```sql
SELECT last_name, job_id, department_id
FROM employees
WHERE department_id =
    (SELECT department_id
     FROM departments
     WHERE department_name = 'Marketing')
ORDER BY job_id;
```
Tell Me / Show Me

Subqueries from Different Tables (continued)

There can be more than one subquery returning information to the outer query.

```sql
SELECT last_name, job_id, salary, department_id
FROM employees
WHERE job_id =
    (SELECT job_id
     FROM employees
     WHERE employee_id = 141)
AND department_id =
    (SELECT department_id
     FROM departments
     WHERE location_id = 150);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>50</td>
</tr>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>1400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>3500</td>
<td>50</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
<td>3100</td>
<td>50</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>2600</td>
<td>50</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Group Functions in Subqueries

Group functions can be used in subqueries. A group function without a GROUP BY clause in the subquery returns a single row.

Which Global Fast Foods staff earn less than the maximum salary?

```
SELECT last_name, first_name, salary
FROM f_staffs
WHERE salary <
(SELECT MAX(salary)
 FROM f_staffs);
```

<table>
<thead>
<tr>
<th>MAX (SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doe</td>
<td>Sue</td>
<td>6.75</td>
</tr>
<tr>
<td>Miller</td>
<td>Bob</td>
<td>10</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Subqueries in the HAVING Clause
Subqueries can also be placed in the HAVING clause.

Remember that the HAVING clause is similar to the WHERE clause, except that the HAVING clause is used to restrict groups and always has a group condition (such as MIN, MAX, AVG) stated.

Because the HAVING clause always has a group condition, the subquery will nearly always have a group condition as well.
**Tell Me / Show Me**

**Subquery Example**
Which departments have a lowest salary that is greater than the lowest salary in department 50?

In this example, the subquery selects and returns the lowest salary in department 50.

The outer query uses this value to select the department ID and lowest salaries of all the departments whose lowest salary is greater than that number.

The HAVING clause eliminated those departments whose MIN salary was less than department 50’s MIN salary.

---

**Employees**

<table>
<thead>
<tr>
<th>MIN(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
</tr>
</tbody>
</table>

**Department MIN(Salary)**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MIN(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>80</td>
<td>8600</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>110</td>
<td>8300</td>
</tr>
</tbody>
</table>

**Query Code**

```
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) >
    (SELECT MIN(salary)
     FROM employees
     WHERE department_id = 50);
```
Summary

Objectives Summarized
In this lesson you have learned to:

• Construct and execute a single-row subquery in the WHERE clause or HAVING clause
• Construct and execute a SELECT statement using more than one subquery
• Construct and execute a SELECT statement using a group function in the subquery
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Multiple-Row Subqueries
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Use the comparison operators IN, ANY and ALL correctly in multiple-row subqueries
• Construct and execute a multiple-row subquery in the WHERE clause or HAVING clause
• Describe what happens if a multiple-row subquery returns a null value
• Understand when multiple-row subqueries should be used, and when it is safe to use a single-row subquery.
• Create a query using the EXIST and NOT EXISTS operators to test for returned rows from the subquery.
Why Learn It?

Purpose
A subquery is designed to find information you don't know so you can find information you want to know.

However, single-row subqueries can return only one row. What if you need to find information based on several rows and several values? The subquery will need to return several rows.

We achieve this using multiple-row subqueries, and three comparison operators: IN, ANY, and ALL.
Tell Me / Show Me

Query Comparison
Whose salary is equal to the salary of an employee in department 20?

Why does this example not work? Because there is more than one employee in department 20, so the subquery returns multiple rows. We call this a multiple-row subquery.

The problem is the equals sign (=) in the WHERE condition. How can one value be equal to (or not equal to) more than one value? It’s a silly question, isn’t it?

SELECT first_name, last_name
FROM employees
WHERE salary =
(SELECT salary
FROM employees
WHERE department_id = 20);

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>6000</td>
</tr>
</tbody>
</table>

ORA-01427: single-row subquery returns more than one row
Tell Me / Show Me

IN, ANY, and ALL
Subqueries that return more than one value are called multiple-row subqueries.

Because we cannot use the single-row comparison operators (=, < and so on), we need different comparison operators for multiple-row subqueries.

The multiple-row operators are: IN, ANY, and ALL. The NOT operator can be used with any of these three operators.

SELECT title, year
FROM d_cds
WHERE year IN
  (SELECT year
   FROM d_cds);

<table>
<thead>
<tr>
<th>D_CDS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE</td>
<td>YEAR</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------</td>
</tr>
<tr>
<td>The Celebrants Live in Concert</td>
<td>1997</td>
</tr>
<tr>
<td>Songs from My Childhood</td>
<td>1999</td>
</tr>
<tr>
<td>Party Music for All Occasions</td>
<td>2000</td>
</tr>
<tr>
<td>Carpe Diem</td>
<td>2000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2002</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

IN

The IN operator is used when the outer query WHERE clause is designed to select only those rows which are equal to one of the list of values returned from the inner query.

For example, we are interested in all the CD titles that have the same year as the CD numbers less than 93. Since we are not sure what the years are for the CDs numbered below 93, the inner query will return a list of years.

The outer query will then return any title that has the same year as any year in the inner query list.

```
SELECT title, year
FROM d_cds
WHERE year IN
  (SELECT year
   FROM d_cds
   WHERE cd_number < 93);
```

<table>
<thead>
<tr>
<th>TITLE</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Celebrants Live in Concert</td>
<td>1997</td>
</tr>
<tr>
<td>Party Music for All Occasions</td>
<td>2000</td>
</tr>
<tr>
<td>Back to the Shire</td>
<td>2002</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

ANY

The ANY operator is used when we want the outer-query WHERE clause to select the rows which are equal to, less than or greater than at least one value in the subquery result set.

The example shown will return any CD title whose year is less than at least one CD title year produced by "The Music Man."

```
SELECT title, producer
FROM d_cds
WHERE year < ANY
(SELECT year
     FROM d_cds
     WHERE producer = 'The Music Man');
```
ALL
The ALL operator is used when we want the outer-query WHERE clause to select the rows which are equal to, less than or greater than all the values in the subquery result set.

The example shown will return any CD title whose year is greater than all the CD title years produced by "The Music Man."

The ALL operator compares a value to every value returned by the inner query.

```
SELECT title, producer, year
FROM d_cds
WHERE year > ALL
(SELECT year
FROM d_cds
WHERE producer = 'The Music Man');
```

<table>
<thead>
<tr>
<th>TITLE</th>
<th>PRODUCER</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back to the Shire</td>
<td>Middle Earth Records</td>
<td>2002</td>
</tr>
<tr>
<td>Whirled Peas</td>
<td>Old Town Records</td>
<td>2004</td>
</tr>
</tbody>
</table>

D_CDS

<table>
<thead>
<tr>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2001</td>
</tr>
</tbody>
</table>

D_CDS
Tell Me / Show Me

NULL VALUES
Suppose that one of the values returned by a multiple-row subquery is null, but other values are not.

• if IN or ANY are used, the outer query will return rows which match the non-null values

• if ALL is used, the outer query returns no rows. This is because ALL compares the outer query row with every value returned by the subquery, including the null. And comparing anything with null results in null not true.

The example lists those employees who are managers.

```
SELECT last_name, employee_id
FROM employees
WHERE employee_id IN
(SELECT manager_id
FROM employees);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>EMPLOYEE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Kochhar</td>
<td>101</td>
</tr>
<tr>
<td>De Haan</td>
<td>102</td>
</tr>
<tr>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Mourgos</td>
<td>124</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null)</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>103</td>
</tr>
<tr>
<td>103</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>124</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

NULL Values in Subqueries

SELECT emp.last_name
FROM employees emp
WHERE emp.employee_id NOT IN
(SELECT mgr.manager_id
FROM employees mgr
WHERE mgr.manager_id IS NOT NULL);

Now, none of the values returned by the inner query is a null value, thus it works.
Multiple-Row Subqueries

Tell Me / Show Me

GROUP BY and HAVING

As you might suspect, the GROUP BY clause, and the HAVING clause can also be used with multiple-row subqueries.

What if you wanted to find the departments whose minimum salary is less than the salary of any employee who works in department 10 or 20?

We need a multiple-row subquery which returns the salaries of employees in departments 10 and 20. The outer query will use a group function (MIN) so we need to GROUP the outer query BY department_id.

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>6000</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MIN(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>80</td>
<td>8600</td>
</tr>
<tr>
<td>110</td>
<td>8300</td>
</tr>
<tr>
<td>(null)</td>
<td>7000</td>
</tr>
</tbody>
</table>
```
Tell Me / Show Me

GROUP BY and HAVING (continued)
Here is the needed SQL statement:

```sql
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) < ANY
(SELECT salary
    FROM employees
    WHERE department_id IN (10, 20));
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>6000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MIN(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>80</td>
<td>8600</td>
</tr>
<tr>
<td>110</td>
<td>8300</td>
</tr>
<tr>
<td>(null)</td>
<td>7000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

GROUP BY and HAVING (continued)

You can even have a GROUP BY clause in the subquery!

Which departments have a minimum salary which is greater than the minimum salaries in departments less than 50? Here is the needed SQL statement:

```
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) > ALL
    (SELECT MIN(salary)
     FROM employees
     WHERE department_id < 50
     GROUP BY department_id);
```
Tell Me / Show Me

One Last Point About Subqueries
Some subqueries may return a single row or multiple rows, depending on the data values in the rows. If there is even a possibility of multiple rows, make sure you write a multiple-row subquery.

For example: who does the same job as Ernst? This single-row subquery works correctly because there is only one Ernst in the table.

But what if later, the business hires a new employee called Susan Ernst?

```sql
SELECT first_name, last_name, job_id
FROM employees
WHERE job_id =
  (SELECT job_id
   FROM employees
   WHERE last_name = 'Ernst');
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bruce</td>
<td>Ernst</td>
<td>IT_PROG</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexander</td>
<td>Hunold</td>
<td>IT_PROG</td>
</tr>
<tr>
<td>Diana</td>
<td>Lorentz</td>
<td>IT_PROG</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

One Last Point About Subqueries (continued)
It would be better to write a multiple-row subquery.

The multiple-row subquery syntax will still work even if the subquery returns a single row.

If in doubt, write a multiple-row subquery!

```
SELECT first_name, last_name, job_id
FROM employees
WHERE job_id IN
  (SELECT job_id
   FROM employees
   WHERE last_name = 'Ernst');
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bruce</td>
<td>Ernst</td>
<td>IT_PROG</td>
</tr>
<tr>
<td>Susan</td>
<td>Ernst</td>
<td>SA_MAN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bruce</td>
<td>Ernst</td>
<td>IT_PROG</td>
</tr>
<tr>
<td>Alexander</td>
<td>Hunold</td>
<td>IT_PROG</td>
</tr>
<tr>
<td>Diana</td>
<td>Lorentz</td>
<td>IT_PROG</td>
</tr>
<tr>
<td>Susan</td>
<td>Ernst</td>
<td>SA_MAN</td>
</tr>
<tr>
<td>Eleni</td>
<td>Zlotkey</td>
<td>SA_MAN</td>
</tr>
</tbody>
</table>
Summary

Objectives Summarized
In this lesson you have learned to:

• Use the comparison operators IN, ANY and ALL correctly in multiple-row subqueries
• Construct and execute a multiple-row subquery in the WHERE clause or HAVING clause
• Describe what happens if a multiple-row subquery returns a null value
• Understand when multiple-row subqueries should be used, and when it is safe to use a single-row subquery.
• Create a query using the EXIST and NOT EXISTS operators to test for returned rows from the subquery.
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Correlated Subqueries
What Will I Learn?

Objectives

In this lesson, you will learn:

- Identify when correlated subqueries are needed.
- Construct and execute correlated subqueries.
- Construct and execute named subqueries using the WITH clause.
Why Learn It?

Purpose

Sometimes you have to answer more than one question in one sentence. Your friend might ask you if you have enough money for a cinema ticket, popcorn, and a Coca Cola. Before you can answer your friend, you need to know what the prices are of the ticket, the popcorn, and the Coca Cola. You also need to see how much money you have in your pocket. So actually, what seemed like an easy question, turns into 4 questions you need answers to, before you can say Yes or No.

In business, you might get asked to produce a report of all employees earning more than the average salary for their department. So here you first have to calculate the average salary per department, and then compare the salary for each employee to the average salary of that employee’s department.
Correlated Subqueries

Tell Me / Show Me

Correlated Subqueries
The Oracle server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement.
A correlated subquery is evaluated once for each row processed by the parent statement.
The parent statement can be a SELECT, UPDATE or DELETE statement.

Diagram:
- GET candidate row from outer query
- EXECUTE inner query using candidate row value
- USE values from inner query to qualify or disqualify candidate row
Tell Me / Show Me

Correlated Subquery Example
Whose salary is higher than the average salary of their department?
To answer that question we need to write a correlated subquery. Correlated subqueries are used for row-by-row processing.
Each subquery is executed once for every row of the outer query.
With a normal subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each row considered by the outer query. In other words, the inner query is driven by the outer query. The example correlated subquery is marked in red.

```sql
SELECT o.first_name, o.last_name, o.salary
FROM employees o
WHERE o.salary >
  (SELECT AVG(i.salary)
   FROM employees i
   WHERE i.department_id = o.department_id);
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steven</td>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Alexander</td>
<td>Hunold</td>
<td>9000</td>
</tr>
<tr>
<td>Kevin</td>
<td>Mourgos</td>
<td>5800</td>
</tr>
<tr>
<td>Eleni</td>
<td>Zlotkey</td>
<td>10500</td>
</tr>
<tr>
<td>Ellen</td>
<td>Abel</td>
<td>11000</td>
</tr>
<tr>
<td>Michael</td>
<td>Hartstein</td>
<td>13000</td>
</tr>
<tr>
<td>Shelley</td>
<td>Higgins</td>
<td>12000</td>
</tr>
</tbody>
</table>
Correlated Subqueries

Tell Me / Show Me

WITH clause

If you have to write a very complex query with joins and aggregations used many times, you can write the different parts of the statement as query blocks and then use those same query block in a SELECT statement. Oracle allows you to write named subqueries in one single statement, as long as you start your statement with the keyword WITH.

• The WITH clause retrieves the results of one or more query blocks and stores those results for the user who runs the query.
• The WITH clause improves performance.
• The WITH clause makes the query easier to read.
Tell Me / Show Me

WITH clause (continued)
The syntax for the WITH clause is as follows:

WITH subquery-name AS (subquery),
    subquery-name AS (subquery)
SELECT  column-list
FROM     {table | subquery-name | view}
WHERE  condition is true;
Correlated Subqueries

Tell Me / Show Me

WITH clause (continued)

Write the query for the following requirement:

Display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.

To solve this query you will need to first get the total salaries per department, then the average salary per department and then you can list just the ones with total salary greater than the average of all departments.
Tell Me / Show Me

WITH clause (continued)

Let’s examine an example of a WITH clause. Let’s start by creating two subqueries, one called `dept_costs` and a second called `avg_cost`. `Avg_cost` uses the result of `dept_cost` and once these two subqueries have been run the actual query itself is executed. The query itself selects from both `dept_cost` and `avg_cost`. By creating the subqueries first, you do not need to create two temporary tables to hold the results of the SUM of salary per department and the AVG of department salaries.
WITH

department_costs AS (  
    SELECT d.department_name, SUM(e.salary) AS dept_total  
    FROM employees e JOIN departments d  
    ON e.department_id = d.department_id  
    GROUP BY d.department_name),

avg_cost AS (  
    SELECT SUM(dept_total)/COUNT(*) AS dept_avg  
    FROM department_costs  
)

SELECT *  
FROM department_costs  
WHERE dept_total >  
    (SELECT dept_avg  
    FROM avg_cost)  
ORDER BY department_name;
Summary

Objectives
In this lesson you have learned:

• Identify when correlated subqueries are needed.
• Construct and execute correlated subqueries.
• Construct and execute named subqueries using the WITH clause.
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
INSERT Statements
What Will I Learn?

Objectives

In this lesson, you will learn to:

- Give examples of why it is important to be able to alter the data in a database
- Construct and execute INSERT statements which insert a single row using a VALUES clause
- Construct and execute INSERT statements that use special values, null values, and date values
- Construct and execute INSERT statements that copy rows from one table to another using a subquery
Why Learn It?

Purpose
Up to now you have been learning how to extract data from a database. It's time to learn how to make changes to the database.

In business, databases are dynamic. They are constantly in the process of having data inserted, updated, and deleted. Think how many times the school's student database changes from day to day and year to year. Unless changes are made, the database would quickly lose its usefulness.

In this lesson, you will begin to use data manipulation language (DML) statements to make changes to a database.
Tell Me / Show Me

Copy Tables Before Inserting

You will be responsible for altering tables in your schema. You will also be responsible for restoring them just as a real Database Administrator would assume that responsibility.

- To keep your schema tables in their original state you will make a copy of each table to complete the practice activities in this and later lessons.
- If you inadvertently alter a copy table you can restore a correct version of it from the original table.
- You should name each table copy_tablename.
- The table copies will not inherit the associated primary-to-foreign-key integrity rules (relationship constraints) of the original tables. The column data types, however, are inherited in the copied tables.
Tell Me / Show Me

Syntax to Create a Copy of a Table

CREATE TABLE copy_tablename
    AS (SELECT * FROM tablename);

For example:

CREATE TABLE copy_f_customers
    AS (SELECT * FROM f_customers);

To verify and view the copy of the table, use the following DESCRIBE and SELECT statements:

DESCRIBE copy_f_customers

SELECT * FROM copy_f_customers;
Tell Me / Show Me

INSERT
The INSERT statement is used to add new rows to a table. The statement requires three values:
- the name of the table
- the names of the columns in the table to populate
- corresponding values for the column

How can we INSERT the data below to create a new customer in the copy_f_customers table?

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
<th>PHONE_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>Katie</td>
<td>Hernandez</td>
<td>92 Chico Way</td>
<td>LA</td>
<td>CA</td>
<td>98008</td>
<td>8586667641</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

INSERT (continued)

The syntax shown uses INSERT to add a new customer to a Global Fast Foods table. This statement explicitly lists each column as it appears in the table. The values for each column are listed in the same order. Note that number values are not enclosed in single quotation marks.

```
INSERT INTO copy_f_customers
    (id, first_name, last_name, address, city, state, zip, phone_number)
VALUES
    (145, 'Katie', 'Hernandez', '92 Chico Way', 'Los Angeles', 'CA', 98008, 8586667641);
```

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
<th>PHONE_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>Katie</td>
<td>Hernandez</td>
<td>92 Chico Way</td>
<td>LA</td>
<td>CA</td>
<td>98008</td>
<td>8586667641</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

INSERT (continued)

Another way to insert values in a table is to implicitly add them by omitting the column names.

One precaution: the values for each column must match exactly the default order in which they appear in the table (as shown in a DESCRIBE statement), and a value must be provided for each column.
Tell Me / Show Me

INSERT (continued)

The INSERT statement in this example was written without explicitly naming the columns. For clarity, however, it is best to use the column names in an INSERT clause.

```
INSERT INTO copy_f_customers
VALUES (475, 'Angelina', 'Wright', '7425 Redwood St', 'San Francisco', 'CA', 94162, '4159982010');
```

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
<th>PHONE_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>475</td>
<td>Angelina</td>
<td>Wright</td>
<td>7425 Redwood St</td>
<td>San Francisco</td>
<td>CA</td>
<td>94162</td>
<td>4159982010</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Check The Table First

Before inserting data into a table, you must check several table details. The DESCRIBE tablenam syntax will return a table summary.

COPY_F_STAFFS TABLE SUMMARY

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>Name</th>
<th>Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Nullable</th>
<th>De-fault</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ID</td>
<td>Number</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FIRST_NAME</td>
<td>Varchar2</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LAST_NAME</td>
<td>Varchar2</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIRTHDATE</td>
<td>Date</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SALARY</td>
<td>Number</td>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OVERTIME_RATE</td>
<td>Number</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRAINING</td>
<td>Varchar2</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Table Summary
As shown in the example, the table summary provides the following information about the columns in the table:

- columns that can have a NULL value
- columns that cannot have duplicate values
- allowable data type
- amount of data that can be entered in a column

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>Name</th>
<th>Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Nullable</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ID</td>
<td>Number</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FIRST_NAME</td>
<td>Varchar2</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LAST_NAME</td>
<td>Varchar2</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIRTHDATE</td>
<td>Date</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SALARY</td>
<td>Number</td>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OVERTIME_RATE</td>
<td>Number</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRAINING</td>
<td>Varchar2</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Table Summary (continued)

Notice the Length column for characters and dates, and the Precision and Scale column for numbers. Precision is the total number of digits, and the scale is the number of digits to the right of the decimal place.

The OVERTIME_RATE number column has precision 5 and scale 2. The maximum value that can be inserted into this column is 999.99.

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>Name</th>
<th>Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Nullable</th>
<th>De-fault</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ID</td>
<td>Number</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>FIRST_NAME</td>
<td>Varchar2</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>LAST_NAME</td>
<td>Varchar2</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>BIRTHDATE</td>
<td>Date</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>SALARY</td>
<td>Number</td>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>OVERTIME_RATE</td>
<td>Number</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>TRAINING</td>
<td>Varchar2</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Copyright © 2011, Oracle. All rights reserved.
Tell Me / Show Me

Inserting Rows With Null Values

The INSERT statement need not specify every column, as long as all the not null columns are named in the INSERT clause, a new row will be inserted.

In our example, the SALARY column is defined as a NOT NULL column. An implicit attempt to add values to the table as shown would generate an error.

```
INSERT INTO copy_f_staffs
  ( id, first_name, last_name, birthdate, overtime_rate)
VALUES
  (15, 'Gregorz', 'Polanski', '25-SEP-88', 17.50);

ORA-01400: cannot insert NULL into
("USQA_JOHN_SQL01_S01"."COPY_F_STAFFS"."SALARY")
```
Tell Me / Show Me

Inserting Rows With Null Values (continued)

If a column can hold null values, it can be omitted from the `INSERT` clause. An implicit insert will automatically insert a null value in that column. To explicitly add null values to a column, use the `NULL` keyword in the `VALUES` list for those columns that can hold null values.

To specify empty strings and/or missing dates, use empty single quotation marks, with not even a blank between them (" "') for missing data.

```sql
INSERT INTO copy_f_staffs
( id, first_name, last_name, birthdate, salary, overtime_rate)
VALUES
(15, 'Gregorz', 'Polanski', '25-SEP-88', 224.25, null);
```

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>BIRTHDATE</th>
<th>SALARY</th>
<th>OVERTIME_RATE</th>
<th>TRAINING</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Gregorz</td>
<td>Polanski</td>
<td>25-SEP-88</td>
<td>224.25</td>
<td>(null)</td>
<td>(null)</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Inserting Special Values

Special values such as SYSDATE and USER can be entered in the VALUES list of an INSERT statement.

SYSDATE will put the current date and time in a column.

USER will insert the current session’s username, which in Oracle Application Express, will be OAE_PUBLIC_USER.

```
INSERT INTO copy_employees
    (employee_id, last_name, email, hire_date, job_id)
VALUES
    (1001, USER, 'Test', SYSDATE,'IT_PROG');
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>(null)</td>
<td>HTMLDB_PUBLIC_USER</td>
<td>Test</td>
<td>(null)</td>
<td>12-MAR-06</td>
<td>IT_PROG</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Inserting Special Values (continued)
In addition, functions and calculated expressions can also be used in the VALUES clause.

The example below illustrates two types of insertions into the Global Fast Foods copy_f_orders table. SYSDATE is used to insert the order_date and a calculated expression is used to enter the order_total.

```sql
INSERT INTO copy_f_orders
( order_number, order_date, order_total, cust_id, staff_id )
VALUES
( 1889, SYSDATE, 87.92*1.08, 123, 19 )
```
Tell Me / Show Me

Inserting Specific Date Values

The default format model for date is DD-MON-RR. With this format, recall that the century defaults to the nearest century (nearest to SYSDATE) with the default time of midnight (00:00:00).

In an earlier section we learned how to use the TO_CHAR function to convert a date to a character string when we want to retrieve and display a date value in a non-default format. Here is a reminder of TO_CHAR:

```
SELECT first_name, TO_CHAR(birthdate, 'Month fmDD, RRRR')
FROM f_staffs
WHERE id = 12;
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>TO_CHAR(BIRTHDATE, ‘MONTHFMDD’, RRRR’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>July 1, 1980</td>
</tr>
</tbody>
</table>
Similarly, if we want to `INSERT` a row with a non-default format for a date column, we must use the `TO_DATE` function to convert the date value (a character string) to a date.

```sql
INSERT INTO f_staffs
  (first_name, TO_DATE(birthdate, 'Month fmDD, RRRR'))
VALUES
  ('Sue', 'July 1, 1980');
```
Inserting Specific Date Values (continued)
A second example of TO_DATE allows the insertion of a specific time of day, avoiding the default time of midnight.

```
INSERT INTO f_staffs
  (first_name, TO_DATE(birthdate, 'Month fmDD, RRRR HH24:MI')
VALUES
  ('Sue', 'July 1, 1980 17:20');
```
Using A Subquery To Copy Rows
Each INSERT statement we have seen so far adds only one row to the table. But suppose we want to copy 100 rows from one table to another?

We do not want to have to write and execute 100 INSERT statements, one after the other. That would be very time-consuming.

Fortunately, SQL allows us to use a subquery within an INSERT statement. All the results from the subquery are inserted into the table. So we can copy 100 rows – or 1000 rows – with one multiple-row subquery within the INSERT.

As you would expect, you don’t need a VALUES clause when using a subquery to copy rows, because the inserted values will be exactly the values returned by the subquery.
Using A Subquery To Copy Rows (continued)

In the example shown, a new table called SALES_REPS is being populated with copies of some of the rows and columns from the EMPLOYEES table.

The WHERE clause is selecting those employees that have job IDs like '%REP%'.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
  SELECT employee_id, last_name, salary, commission_pct
  FROM   employees
  WHERE  job_id LIKE '%REP%';
```

The number of columns and their data types in the column list of the INSERT clause must match the number of columns and their data types in the subquery.

The subquery is not enclosed in parentheses as is done with subqueries in a WHERE clause of a SELECT statement.
Using A Subquery To Copy Rows (continued)
If we want to copy all the data – all rows and all columns – the syntax is even simpler.

To select all rows from the EMPLOYEES table and insert them into the SALES_REPS table, the statement would be written as shown:

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

Again, this will work only if both tables have the same number of columns, in the same order, with the same data types.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

INSERT INTO
USER
Transaction
Explicit
Objectives Summarized
In this lesson you have learned to:

• Give examples of why it is important to be able to alter the data in a database
• Construct and execute INSERT statements which insert a single row using a VALUES clause
• Construct and execute INSERT statements that use special values, null values, and date values
• Construct and execute INSERT statements that copy rows from one table to another using a subquery
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Updating Column Values and Deleting Rows
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Construct and execute an UPDATE statement
• Construct and execute a DELETE statement
• Construct and execute a query that uses a subquery to update and delete data from a table
• Construct and execute a query that uses a correlated subquery to update and delete from a table
• Explain how foreign-key and primary-key integrity constraints affect UPDATE and DELETE statements
Why Learn It?

Purpose
Wouldn't it be a wonderful world if, once you got something done, it never needed to be changed or redone? Your bed would stay made, your clothes would stay clean, and you'd always be getting passing grades. Unfortunately, in databases, as in life, "There is nothing permanent except change."

Updating, inserting, deleting, and managing data is a Database Administrator's (DBA's) job. In this lesson, you will become the DBA of your own schema and learn to manage your database.
Tell Me / Show Me

UPDATE
The UPDATE statement is used to modify existing rows in a table. It requires four values:

• the name of the table
• the name of the column(s) whose values will be modified
• a new value for each of the column(s) being modified
• a condition that identifies which rows in the table will be modified.

The new value for a column can be the result of a single-row subquery.
Tell Me / Show Me

UPDATE (continued)
The example shown uses an UPDATE statement to change the phone number of one customer in the Global Fast Foods database. Note that the copy_f_customers table is used in this transaction.

```
UPDATE copy_f_customers
SET phone_number='4475582344'
WHERE id=123;
```
**Tell Me / Show Me**

**UPDATE (continued)**

We can change several columns and/or several rows in one UPDATE statement. This example changes both the phone number and the city for two Global Fast Foods customers.

```
UPDATE copy_f_customers
SET phone_number='4475582344',
city = 'Chicago'
WHERE id < 200;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
<th>PHONE_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Cole</td>
<td>Bee</td>
<td>123 Main Street</td>
<td>Chicago</td>
<td>FL</td>
<td>32838</td>
<td>4475582344</td>
</tr>
<tr>
<td>456</td>
<td>Zoe</td>
<td>Twee</td>
<td>1009 Oliver Avenue</td>
<td>Boston</td>
<td>MA</td>
<td>12889</td>
<td>7098675309</td>
</tr>
<tr>
<td>145</td>
<td>Katie</td>
<td>Hernandez</td>
<td>92 Chico Way</td>
<td>Chicago</td>
<td>CA</td>
<td>98008</td>
<td>4475582344</td>
</tr>
</tbody>
</table>

Copyright © 2011, Oracle. All rights reserved.
Tell Me / Show Me

UPDATE (continued)
Which rows would be updated in the following transaction?

UPDATE copy_f_customers
SET phone_number='9876543210';

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
<th>PHONE_NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Cole</td>
<td>Bee</td>
<td>123 Main Street</td>
<td>Orlando</td>
<td>FL</td>
<td>32838</td>
<td>4475582344</td>
</tr>
<tr>
<td>456</td>
<td>Zoe</td>
<td>Twee</td>
<td>1009 Oliver Avenue</td>
<td>Boston</td>
<td>MA</td>
<td>12889</td>
<td>7098675309</td>
</tr>
<tr>
<td>145</td>
<td>Katie</td>
<td>Hernandez</td>
<td>92 Chico Way</td>
<td>Los Angeles</td>
<td>CA</td>
<td>98008</td>
<td>8586667641</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Updating a Column with a value from a Subquery
We can use the result of a single-row subquery to provide the new value for an updated column.

```
UPDATE copy_f_staffs
SET salary = (SELECT salary
              FROM copy_f_staffs
              WHERE id = 9)
WHERE id = 12;
```

This example changes the salary of one employee (id = 12) to the same salary as another employee (id = 9). As usual, the subquery executes first and retrieves the salary for employee id=12. This salary value is then used to update the salary for employee id=9.

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Bob</td>
<td>Miller</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>Sue</td>
<td>Doe</td>
<td>10</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Updating Two Columns with Two Subquery Statements

To update several columns in one UPDATE statement, it is possible to write multiple single-row subqueries, one for each column.

In this example the UPDATE statement changes the salary and staff type of one employee (id = 12) to the same values as another employee (id = 9).

```
UPDATE copy_f_staffs
SET salary = (SELECT salary
              FROM copy_f_staffs
              WHERE id = 9),
    staff_type = (SELECT staff_type
                  FROM copy_f_staffs
                  WHERE id = 9)
WHERE id = 12;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>STAFF_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Bob</td>
<td>Miller</td>
<td>10</td>
<td>Cook</td>
</tr>
<tr>
<td>12</td>
<td>Sue</td>
<td>Doe</td>
<td>10</td>
<td>Cook</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Updating Rows Based On Another Table
As you may have expected, the subquery can retrieve information from one table which is then used to update another table.

In this example, a copy of the f_staffs table was created. Then data from the original f_staffs table was retrieved, copied, and used to populate the copy of the f_staffs table.

UPDATE copy_f_staffs
SET salary = (SELECT salary
              FROM f_staffs
              WHERE id = 9)
WHERE id = 9;
Tell Me / Show Me

Updating Rows Based On The Same Table

As you already know subqueries can be either stand alone or correlated. In a correlated subquery you are updating a row in a table based on a select from that same table.

In the example below, a copy of the department name column was created in the employees table. Then data from the original departments table was retrieved, copied and used to populate the copy of the column in the employees table.

```
UPDATE employees e
SET e.department_name = (SELECT d.department_name
                         FROM departments d
                         WHERE e.department_id = d.department_id);
```
Tell Me / Show Me

DELETE

The DELETE statement is used to remove existing rows in a table. The statement requires two values:

- the name of the table
- the condition that identifies the rows to be deleted
Tell Me / Show Me

DELETE (continued)
The example shown uses the Global Fast Foods database to delete one row, the customer with ID number 123.

DELETE FROM copy_f_customers
WHERE ID= 123;

What do you predict will be deleted if the WHERE clause is eliminated in a DELETE statement?

All rows in the table are deleted if you omit the WHERE clause.
Tell Me / Show Me

Subquery DELETE
Subqueries can also be used in DELETE statements.

The example shown deletes rows from the employees table for all employees who work in the Shipping department. Maybe this department has been renamed or eliminated.

```
DELETE FROM employees
WHERE  department_id =
  (SELECT department_id
   FROM departments
   WHERE  department_name = 'Shipping');
```
Tell Me / Show Me

Correlated Subquery DELETE
The example below deletes rows of all employees who work for a manager that manages more than 2 departments.

```
DELETE FROM employees e
WHERE e.manager_id =
    (SELECT d.manager_id
     FROM departments d
     WHERE e.department_id = d.department_id
     HAVING count(d.department_id) > 2
     GROUP BY d.manager_id);
```
Tell Me / Show Me

Integrity Constraint Errors

Integrity constraints ensure that the data conforms to a needed set of rules. The constraints are automatically checked whenever a DML statement which could break the rules is executed. If the any rule would be broken, the table is not updated and an error is returned.

This example violates a NOT NULL constraint, because first_name has a not null constraint and id=123 does not exist, so the subquery returns a null result.

```
UPDATE copy_f_staffs
SET first_name = (SELECT first_name
                 FROM copy_f_staffs
                 WHERE id = 123);
```

ORA-01407: cannot update
(“USQA_JOHN_SQL01_S01”.“COPY_F_STAFFS”.“FIRST_NAME”) to NULL
Tell Me / Show Me

Integrity Constraint Errors (continued)
When will primary key - foreign key constraints be checked?

The EMPLOYEES table has a foreign key constraint on department_id, which references the department_id of the DEPARTMENTS table. This ensures that every employee belongs to a valid department. In the DEPARTMENTS table, department_ids 10 and 20 exist but 15 does not.

Which of the following statements will return an error?

1. UPDATE employees SET department_id = 15  
   WHERE employee_id = 100;

2. DELETE FROM departments WHERE department_id = 10;

3. UPDATE employees SET department_id = 10  
   WHERE department_id = 20;
Tell Me / Show Me

Integrity Constraint Errors (continued)

When modifying your copy tables (for example copy_f_customers) you may see not null constraint errors, but you will not see any primary key – foreign key constraint errors.

This is because the CREATE TABLE …. AS (SELECT …) statement used to create the copy tables copies the rows to the copy table and copies the not null constraints, but does not copy primary key – foreign key constraints.

Therefore at present there are no primary key – foreign key constraints on the copy tables.

Adding constraints is covered in another lesson.
Tell Me / Show Me

Terminology

Key terms used in this lesson include:

DELETE
Integrity constraint
UPDATE
Correlated subquery UPDATE
Correlated subquery DELETE
Summary

Objectives Summarized
In this lesson you have learned to:

- Construct and execute an UPDATE statement
- Construct and execute a DELETE statement
- Construct and execute a query that uses a subquery to update and delete data from a table
- Construct and execute a query that uses a correlated subquery to update and delete from a table
- Explain how foreign-key and primary-key integrity constraints affect UPDATE and DELETE statements
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
DEFAULT Values, MERGE, and Multi-Table Inserts
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Understand when to specify a DEFAULT value
• Construct and execute a MERGE statement
• Construct and execute DML statements using subqueries
• Construct and execute multi-table inserts
Why Learn It?

Purpose
Up to now, you have been updating data using a single INSERT statement. It has been relatively easy when adding records one at a time, but what if your company is very large and utilizes a data warehouse to store sales records and customer, payroll, accounting, and personal data?

In this case, data is probably coming in from multiple sources and being managed by multiple people. Managing data one record at a time could be very confusing and very time consuming.
Why Learn It?

Purpose (continued)
How do you determine what has been newly inserted or what has been recently changed?

In this lesson, you will learn a more efficient method to update and insert data using a sequence of conditional INSERT and UPDATE commands in a single atomic statement.

You will also learn how to retrieve data from one single subquery and INSERT the rows returned into more than one target table.

As you extend your knowledge in SQL, you'll appreciate effective ways to accomplish your work.
Tell Me / Show Me

DEFAULT
A column in a table can be given a default value. This option prevents null values from entering the columns if a row is inserted without a specified value for the column.

Using default values also allows you to control where and when the default value should be applied. The default value can be a literal value, an expression, or a SQL function, such as SYSDATE and USER, but the value cannot be the name of another column.

The default value must match the data type of the column. DEFAULT can be specified for a column when the table is created or altered.
**Tell Me / Show Me**

**DEFAULT Example**

The example below shows a default value being specified at the time the table is created:

```
CREATE TABLE my_employees (  
hire_date DATE DEFAULT SYSDATE,  
first_name VARCHAR2(15),  
last_name VARCHAR2(15));
```

When rows are added to this table, SYSDATE will be added to any row that does not explicitly specify a hire_date value.
Explicit DEFAULT with INSERT

Explicit defaults can be used in INSERT and UPDATE statements. The INSERT example using the DEPARTMENTS table shows the explicit use of DEFAULT.

```
INSERT INTO departments
    (department_id, department_name, manager_id)
VALUES
    (300, 'Engineering', DEFAULT);
```

If a default value was set for the manager_id column, Oracle sets the column to the default value. However, if no default value was set when the column was created, Oracle inserts a null value.
Tell Me / Show Me

Explicit DEFAULT with UPDATE

Explicit defaults can be used in INSERT and UPDATE statements. The UPDATE example using the Oracle DEPARTMENTS table shows explicit use of DEFAULT.

```
UPDATE departments
SET manager_id = DEFAULT
WHERE department_id = 10;
```

If a default value was set for the department_id column, Oracle sets the column to the default value. However, if no default value was set when the column was created, Oracle inserts a null value.
Tell Me / Show Me

**MERGE**
Using the MERGE statement accomplishes two tasks at the same time. MERGE will INSERT and UPDATE simultaneously. If a value is missing, a new one is inserted. If a value exists, but needs to be changed, MERGE will update it.

To perform these kinds of changes to database tables, you need to have INSERT and UPDATE privileges on the target table and SELECT privileges on the source table. Aliases can be used with the MERGE statement.
Tell Me / Show Me

MERGE Syntax
MERGE INTO destination-table USING source-table
ON matching-condition
WHEN MATCHED THEN UPDATE
SET ……
WHEN NOT MATCHED THEN INSERT
VALUES (……);

One row at a time is read from the source table, and its column values are compared with rows in the destination table using the matching condition. If a matching row exists in the destination table, the source row is used to update column(s) in the matching destination row. If a matching row does not exist, values from the source row are used to insert a new row into the destination table.
Tell Me / Show Me

MERGE Example
This example uses the EMPLOYEES table (alias e) as a data source to insert and update rows in a copy of the table named COPY_EMP (alias c).

```
MERGE INTO copy_emp  c USING employees e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN  UPDATE
  SET
    c.last_name = e.last_name,
    c.department_id = e.department_id
WHEN NOT MATCHED THEN INSERT
  VALUES  (e.employee_id, e.last_name, e.department_id);
```
**Tell Me / Show Me**

### MERGE Example (continued)

MERGE INTO `copy_emp` c USING `employees` e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN UPDATE
  SET
    c.last_name  = e.last_name,
    c.department_id  = e.department_id
WHEN NOT MATCHED THEN INSERT
  VALUES  (e.employee_id, e.last_name, e.department_id);

EMPLOYEES rows 100 and 103 have matching rows in COPY_EMP, and so the matching COPY_EMP rows were updated.

EMPLOYEE 142 had no matching row, and so was inserted into COPY_EMP.

#### EMPLOYEES (source table)

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>60</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>50</td>
</tr>
</tbody>
</table>

#### COPY_EMP before the MERGE is executed

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Smith</td>
<td>40</td>
</tr>
<tr>
<td>103</td>
<td>Chang</td>
<td>30</td>
</tr>
</tbody>
</table>

#### COPY_EMP after the MERGE has executed

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>60</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>50</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Multi-Table Inserts

Multi-table inserts are used when the same source data should be inserted into more than one target table. This functionality is useful when you are working in a data warehouse environment, where it is common to regularly move data from the operational systems into a data warehouse for analytical reporting and analysis. Creating and managing data warehouses is one way of managing the sometimes very high number of rows inserted into operational systems during a normal working day.

Imagine, for instance, how many rows your mobile/cell telephone provider must create daily. At least one for each time you use your mobile/cell phone, and how many calls do you make and receive a day?
Tell Me / Show Me

Multi-Table Inserts (continued)
Then add the number of SMS’s you send and receive.
Add to that your mobile surfing and downloads of ringtones, wallpapers, games and other mobile applications.
Multiply that number by the number of customers.
That might give you an idea of the amount of data the telecommunication companies have to manage.
These rows may have to be inserted into more than one table in the data warehouse, so if we can just SELECT them once and then replicate them, that will improve the performance.
Multi-Table Inserts (continued)

Multi-table inserts can be unconditional or conditional. In an unconditional multi-table insert Oracle will insert all rows returned by the subquery into all table insert clauses found in the statement.

In a conditional multi-table insert you can specify either ALL or FIRST.

**ALL**

If you specify ALL, the default value, then the database evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause.

For each WHEN clause whose condition evaluates to true, the database executes the corresponding INTO clause list.
Tell Me / Show Me

Multi-Table Inserts (continued)

**FIRST**

If you specify FIRST, then the database evaluates each WHEN clause in the order in which it appears in the statement. For the first WHEN clause that evaluates to true, the database executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

**ELSE clause**

For a given row, if no WHEN clause evaluates to true, then:
- If you have specified an ELSE clause, then the database executes the INTO clause list associated with the ELSE clause.
- If you did not specify an else clause, then the database takes no action for that row.
Tell Me / Show Me

Multi-Table Inserts Syntax
Multi-table insert statement syntax is as follows:

```
INSERT ALL INTO clause VALUES clause SUBQUERY

INSERT ALL
  INTO all_calls VALUES (caller_id, call_timestamp, call_duration, call_format)
  INTO police_record_calls VALUES (caller_id, call_timestamp, recipient_caller)
SELECT caller_id, call_timestamp, call_duration, call_format, recipient_caller
FROM calls
WHERE TRUNC(call_timestamp) = TRUNC(SYSDATE)
```
Tell Me / Show Me

Multi-Table Inserts Conditional

```sql
INSERT ALL
    WHEN call_format IN ('tlk', 'txt', 'pic') THEN
        INTO all_calls VALUES (caller_id, call_timestamp, call_duration, call_format)
    WHEN call_format IN ('tlk', 'txt') THEN
        INTO police_record_calls VALUES (caller_id, call_timestamp, recipient_caller)
    WHEN call_duration < 50 AND call_type = 'tlk' THEN
        INTO short_calls VALUES (caller_id, call_timestamp, call_duration)
    WHEN call_duration >= 50 AND call_type = 'tlk' THEN
        INTO long_calls VALUES (caller_id, call_timestamp, call_duration)
SELECT caller_id, call_timestamp, call_duration, call_format, recipient_caller
FROM calls
WHERE TRUNC(call_timestamp) = TRUNC(SYSDATE)
```
Summary

Objectives Summarized
In this lesson you have learned to:

• Understand when to specify a DEFAULT value
• Construct and execute a MERGE statement
• Construct and execute DML statements using subqueries
• Construct and execute multi-table inserts
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Creating Tables
What Will I Learn?

Objective
In this lesson, you will learn to:

• List and categorize the main database objects
• Review a table structure
• Describe how schema objects are used by the Oracle database
• List and provide an example of each of the number, character, and date data types
• Create a table using the appropriate data type for each column
• Explain the use of external tables
• Query the Data Dictionary to obtain the names and other attributes of database objects
Creating Tables

Why Learn It?

Purpose
Up until now, you have selected, added, and deleted information in tables in an existing database. If you have a job as a Database Administrator (DBA) you will be expected to know how to create tables as well.

In this lesson, you will learn what database objects are most frequently used, how to look at the table structure, and how to create new tables. Your tables will be small compared to tables that hold millions of rows and hundreds of columns, but creating a small table involves the same SQL statements and syntax as creating a very large one.

You will also learn about external tables, which are similar in structure to the normal Oracle database tables, but the actual data rows are not held inside the database. Instead they are held in a flat file stored externally to the database, and are accessed only when required.
Database Schema Objects

An Oracle Database can contain many different types of objects. This section introduces the most commonly used objects, and also describes how the Oracle Server uses the information stored in the Data Dictionary when it is performing work as a result of the SQL statements you issue.

The main database object types are:

- Table
- Index
- Constraint
- View
- Sequence
- Synonym

Some of these object types can exist independently and others can not.
Database Schema Objects (continued)

Some of the object types take up space in the database, known as Storage, and others do not. Database objects taking up Storage are known as Segments. Tables and Indexes are examples of Segments, as the rows stored in the tables and the column values take up physical space on disk in the database.

Views, Constraints, Sequences, and Synonyms are just objects, the only space they take up in the database, is the definition of those objects, none of them have any actual data rows associated with them.

The database stores the definitions of all database objects in the Data Dictionary, and these definitions are accessible to all users of the database as well as the database itself.
Database Schema Objects (continued)

Have you ever wondered, how Oracle knows which columns to return from a Query? For example, if you specify SELECT * FROM d_cds instead of SELECT cd_number, title FROM d_cds how does Oracle know which columns to return?

The database looks up the definition of the table used in the query, translates the ‘*’ into the full list of columns and returns the result to you.

The database uses the Data Dictionary for all statements you issue, even if you list the columns instead of using ‘*’. It checks that the tables you are referencing in your statement exist in the database, it checks that the column names are correct, it checks if you have the correct privileges to perform the action you are requesting, and finally it uses the Data Dictionary to decide on what is called the Execution Plan – how it will actually perform the statement.
Database Schema Objects (continued)

The Data Dictionary itself can be queried by all database users. In Application Express it can be accessed both via SQL statements in the SQL Workshop> SQL Commands interface and also from the SQL Workshop> Object Browser interface.

In the SQL Commands window you have to know the names of the views you are querying and in the Object Browser interface you simply click on the listed objects to see the details of them. So if you wanted to see the details of the D_CDS table you could simply click in it in the table listing:
Database Schema Objects (continued)

In this example, using the Object Browser, you can see the details of the D_CDS table, as well as all the options available to you to view the data, see the indexes, constraints and grants on the table.
Database Schema Objects (continued)

In this example, using the SQL Commands window, you must ask for a DESCRIPTION of the table. All the extra options offered by Object Browser are not available in this interface.

```
DESC d_cds
```

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Data Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Primary Key</th>
<th>Nullable</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_cds</td>
<td>CD_NUMBER</td>
<td>Number</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>d_cds</td>
<td>TITLE</td>
<td>Varchar2</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>d_cds</td>
<td>PRODUCER</td>
<td>Varchar2</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>d_cds</td>
<td>YEAR</td>
<td>Varchar2</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Table Creation
All data in a relational database is stored in tables. When creating a new table, use the following rules for table names and column names:

- Must begin with a letter
- Must be 1 to 30 characters long
- Must contain only A - Z, a - z, 0 - 9, _ (underscore), $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle Server reserved word
Tell Me / Show Me

Naming Conventions

It is best to use descriptive names for tables and other database objects. If a table will store information about students, name it STUDENTS, not PEOPLE or CHILDREN.

Also, names are not case sensitive. For example, STUDENTS is treated the same as STuDents or students.

Creating tables is part of SQL's data definition language (DDL). Other DDL statements used to set up, change, and remove data structures from tables include ALTER, DROP, RENAME, and TRUNCATE.
CREATE TABLE
To create a new table, you must have the CREATE TABLE privilege and a storage area for it. The database administrator uses data control language (DCL) statements to grant this privilege to users and assign a storage area.

Tables belonging to other users are not in your schema. If you want to use a table that is not in your schema, use the table owner's name as a prefix to the table name:

SELECT * FROM mary.students;

You must be granted access to a table to be able to select from it.
Tell Me / Show Me

CREATE TABLE Syntax

To create a new table consider the following syntax details:

- **table** is the name of the table
- **column** is the name of the column
- **datatype** is the column's data type and length
- **DEFAULT expression** specifies a default value if a value is omitted in the INSERT statement

```
CREATE TABLE table
(column datatype [DEFAULT expression],
(column datatype [DEFAULT expression],
(......[]);
```

For example:

```
CREATE TABLE cd_collection
(cd_number NUMBER(2),
title VARCHAR2(14),
artist VARCHAR2(13),
purchase_date DATE DEFAULT SYSDATE);
```
Tell Me / Show Me

Creating A Table Using A Subquery

A second method for creating a table is to apply the AS subquery clause, which both creates the table and inserts rows returned from the subquery.

This is an easy way to create a copy of a table to practice SQL statements. Note that you need to create a column alias for those columns in the subquery that contain expressions.

Only datatype definitions and NOT NULL constraints are passed on to a new table created from a subquery. This is because the new table could be used in a different context, in which existing PK-FK relationships may not be relevant.

```
CREATE TABLE tablename [(column, column, ...)]
AS subquery;
```

Two examples:

```
CREATE TABLE copy_mymytable
AS
(SELECT code, name, start_date, end_date, give_away
FROM f_promotional_menus);

CREATE TABLE dept80
AS
SELECT employee_id, last_name, salary*12 ANNSAL, hire_date
FROM employees
WHERE department_id = 80;
```
Creating Tables

Tell Me / Show Me

Creating A Table Using A Subquery (continued)

When a copy of a table is made using a subquery, the following rules are important:

• The column names in the new table will be identical to those in the original table, unless column aliases are used
• The column datatypes in the new table will be identical to those in the original table
• Only Not Null constraints are copied to the table, no other constraint types will exist on the new table
External Tables

Oracle also supports another table type: External table.

In an external table the data rows are not held inside the database files, but are instead found in a flat file, stored externally to the database.
Creating Tables

Tell Me / Show Me

External Tables (continued)

Typically an external table is used to store data migrated from older versions of the databases used by a company.

When a company is implementing a new application and database, they typically need to import most of the data from the old systems to the new system for normal read and write access, but there may be some data that is not used frequently and therefore would only need to be accessed for read access. This kind of data could be held in an external table.

One of the many benefits for Oracle is that data held in external tables only has to be backed up once, and then never again unless the contents of the file change.
External Tables (continued)
The syntax to create an external table is very similar to that of creating a standard table, except that it has extra syntax at the end. Please note the new syntax, not used in standard SQL statements for table creation.

The new syntax is found in red on the next slide.

**ORGANIZATION EXTERNAL** -- tells Oracle to create an external table
**TYPE ORACLE_LOADER** -- of type Oracle Loader (an Oracle Product)
**DEFAULT DIRECTORY def_dir1** -- what is the name of the directory where the file exists
**ACCESS PARAMETERS** -- how to read the file
**RECORDS DELIMITED BY NEWLINE** -- how to identify a new row starts
**FIELDS** – the start of the external file field name and datatype specification
**LOCATION** – name of the actual file containing the data
External Tables Syntax Example

CREATE TABLE emp_load
   (employee_number CHAR(5),
    employee_dob CHAR(20),
    employee_last_name CHAR(20),
    employee_first_name CHAR(15),
    employee_middle_name CHAR(15),
    employee_hire_date DATE)
ORGANIZATION EXTERNAL
   (TYPE ORACLE_LOADER
    DEFAULT DIRECTORY def_dir1
    ACCESS PARAMETERS
    (RECORDS DELIMITED BY NEWLINE
     FIELDS (employee_number CHAR(2),
              employee_dob CHAR(20),
              employee_last_name CHAR(18),
              employee_first_name CHAR(11),
              employee_middle_name CHAR(11),
              employee_hire_date CHAR(10) date_format DATE mask "mm/dd/yyyy")
    LOCATION ('info.dat'));
Data Dictionary

Two kinds of tables exist in an Oracle Database: User tables and Data Dictionary tables. You can issue SQL statements to access both kinds of tables, and you can select, insert, update, and delete data in the user tables and you can select data in the Data Dictionary tables.

User tables created by you containing your data: D_CDS, D_SONGS, D_EVENTS etc.

Data Dictionary tables: DICTIONARY, USER_OBJECTS, USER_TABLES, USER_SEGMENTS, USER_INDEXES etc.
Tell Me / Show Me

Data Dictionary (continued)

The Data Dictionary tables are all owned by a special Oracle user called SYS and only SELECT statements should be used when working with any of these tables. To make these tables safe from accidental user access, they all have views created which is how the Data Dictionary is accessed by database users. If any Oracle user attempts to do inserts, updates or deletes against any of the Data Dictionary tables, then this operation would be disallowed, as it might compromise the integrity of the entire database.
Data Dictionary (continued)

When you are using the Data Dictionary views in the SQL Commands interface, you need to know the names of the Dictionary views you are working with. In Oracle, this is quite simple: prefix the object type you are looking for with a USER_xxx or an ALL_xxx, where xxx is the object type. So if you want to investigate indexes, then simply select from USER_INDEXES, if you want information about sequences, then the table is USER SEQUENCES and so on.

Two very important views to remember are DICTIONARY (DICT) and DICT_COLUMNS.
Tell Me / Show Me

Data Dictionary (continued)

Dictionary (DICT) has a listing of all the dictionary views in the database.

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Data Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Primary Key</th>
<th>Nullable</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DICT</td>
<td>TABLE_NAME</td>
<td>Varchar2</td>
<td>30</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>Name of the object</td>
</tr>
<tr>
<td>COMMS</td>
<td>Varchar2</td>
<td>4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>Text comment on the object</td>
</tr>
</tbody>
</table>

DICT_COLUMNS lists all of the columns of all the tables in the Data Dictionary.

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Data Type</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Primary Key</th>
<th>Nullable</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DICT_COLUMNS</td>
<td>TABLE_NAME</td>
<td>Varchar2</td>
<td>30</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>Name of the object that contains the column</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>Varchar2</td>
<td>30</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>Name of the column</td>
</tr>
<tr>
<td>COMMS</td>
<td>Varchar2</td>
<td>4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>Text comment on the object</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

CREATE TABLE
Data dictionary
Table
Schema
DEFAULT
Summary

Objectives Summarized
In this lesson you have learned to:

• Categorize the main database objects
• Review a table structure
• Describe how schema objects are used
• List and provide an example of each of the number, character, and date data types
• Create a table using the appropriate data type for each column
• Explain the use of external tables
• Use the Data Dictionary to obtain the names and other attributes of database objects
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Using Data Types
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Create a table using TIMESTAMP and TIMESTAMP WITH TIME ZONE column data types
• Create a table using INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND column data types
• Give examples of organizations and personal situations where it is important to know to which time zone a date-time value refers
Why Learn It?

Purpose
If you ever travel to another country, you'll quickly find out that the money in your pocket may not be that of the local currency. If you want to buy something, it will be necessary to convert your money into the currency of the local country.

This conversion process is a lot like dealing with data types in SQL. Different types of data have different types of characteristics, the purpose of which is to efficiently store data. In this lesson, you will learn more about data types and their uses.
Tell Me / Show Me

Data Type Overview
Each value manipulated by Oracle has a data type. A value's data type associates a fixed set of properties with the value. These properties cause the database to treat values of one data type differently from values of another. Different data types offer several advantages:

- Columns of a single type produce consistent results. For example, DATE data type columns always produce date values.
- You cannot insert the wrong type of data into a column. For example, columns of data type DATE will prevent NUMBER type data from being inserted.

For these reasons, each column in a relational database can hold only one type of data. You cannot mix data types within a column.
Tell Me / Show Me

Common Data Types
The most commonly used column data types are:

• For character values: CHAR (fixed size, maximum 2000 characters); VARCHAR2 (variable size, maximum 4000 characters); CLOB (variable size, maximum 128 terabytes)
• For number values: NUMBER (variable size, maximum precision 38 digits)
• For date and time values: DATE, TIMESTAMP ...., INTERVAL
• For binary values (eg multimedia: JPG, WAV, MP3 and so on): RAW (variable size, maximum 2000 bytes); BLOB (variable size, maximum 128 terabytes).
Common Data Types (continued)
The most commonly used column data types are:

• For character values, it is usually better to use VARCHAR2 or CLOB than CHAR, because it saves space.

• For example, an employee’s last name is ‘Chang’. In a VARCHAR2(30) column, only the 5 significant characters are stored: C h a n g. But in a CHAR(30) column, 25 trailing spaces would be stored as well, to make a fixed size of 30 characters.

• Number values can be negative as well as positive. For example, NUMBER(6,2) can store any value from +9999.99 down to –9999.99.
DATE-TIME Data Types

The DATE data type stores a value of centuries down to whole seconds, but cannot store fractions of a second. ’21-AUG-2003 17:25:30’ is a valid value, but ’21-AUG-2003 17:25:30.255’ is not.

The TIMESTAMP data type is an extension of the DATE data type which allows fractions of a second.

For example, TIMESTAMP(3) allows 3 digits after the whole seconds, allowing values down to milliseconds to be stored.
Tell Me / Show Me

TIMESTAMP …. With [LOCAL] Time Zone
Think about the time value ‘17:30’. Of course it means “half past five in the afternoon”. But where in the world? Is it half past five New York City time or Beijing time or Istanbul time or …. ?

In today’s globalized organizations which operate in many different countries, it is important to know to which time zone a date-time value refers.
Tell Me / Show Me

TIMESTAMP .... With [LOCAL] Time Zone (continued)
TIMESTAMP WITH TIME ZONE stores a time zone value as a displacement from Universal Coordinated Time or UCT (previously known as Greenwich Mean Time or GMT).

A value of ’21-AUG-03 08:00:00 –5:00’ means 8:00 am 5 hours behind UTC.
This is US Eastern Standard Time (EST).

TIMESTAMP WITH LOCAL TIME ZONE is the same, but with one difference: when this column is SELECTed in a SQL statement, the time is automatically converted to the selecting user’s time zone.
Tell Me / Show Me

TIMESTAMP With … TIME ZONE Example

CREATE TABLE time_example
(first_column    TIMESTAMP WITH TIME ZONE,
 second_column  TIMESTAMP WITH LOCAL TIME ZONE);

INSERT INTO time_example (first_column, second_column)
VALUES ('15-NOV-2007 08:00:00 AM -05:00',
 '15-NOV-2007 08:00:00');
Both values are stored with a time displacement of –05:00 hours (EST).
But now a user in Istanbul executes:

SELECT * FROM time_example;

<table>
<thead>
<tr>
<th>FIRST_COLUMN</th>
<th>SECOND_COLUMN</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-NOV-2007 08:00:00 AM -05:00</td>
<td>15-NOV-2007 08:00:00</td>
</tr>
</tbody>
</table>

Istanbul time is 7 hours ahead of EST; when it’s 8am in New York City, it’s 3pm in Istanbul.
INTERVAL Data Types

These store the elapsed time, or interval of time, between two date-time values.

- INTERVAL YEAR TO MONTH stores a period of time measured in years and months.

- INTERVAL DAY TO SECOND stores a period of time measured in days, hours, minutes and seconds.

```
CREATE TABLE time_example1
(loan_duration INTERVAL YEAR(3) TO MONTH,
day_duration   INTERVAL DAY(3) TO SECOND);
```
Tell Me / Show Me

**INTERVAL YEAR ... TO MONTH**

The data type syntax is:

```
INTERVAL YEAR
[(year_precision)] TO MONTH
```

`year_precision` is the maximum number of digits in the YEAR element. The default value of `year_precision` is 2.

This example shows an interval of 120 months,

```
CREATE TABLE time_example2
(loan_duration INTERVAL YEAR(3) TO MONTH);

INSERT INTO time_example2 (loan_duration)
VALUES (INTERVAL '120' MONTH(3));

SELECT TO_CHAR(sysdate+loan_duration,
               'dd-mon-yyyy')
FROM time_example2;

-- assume today's date is 26-Sep-2005
```

<table>
<thead>
<tr>
<th>TO_CHAR(SYSDATE+LOAN_DUR, 'DD-MON-YYYYY')</th>
</tr>
</thead>
<tbody>
<tr>
<td>26-SEP-2015</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

INTERVAL DAY … TO SECOND
Use this when you need a more precise difference between two date-time values.
The data type syntax is:
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]

day_precision is the maximum number of digits in the DAY date-time field. The default is 2.

fractional_seconds_precision is the number of digits in the fractional part of the SECOND date-time field. The default is 6.

CREATE TABLE time_example3
    (day_duration INTERVAL DAY(3) TO SECOND);

INSERT INTO time_example3 (day_duration)
    VALUES (INTERVAL '25' DAY(2));

SELECT sysdate + day_duration "25 Days"
    FROM time_example3;

-- assume today's date is 06-Oct-2007

<table>
<thead>
<tr>
<th>25 Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-OCT-07</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

CLOB
BLOB
TIMESTAMP
TIMESTAMP WITH TIMEZONE
TIMESTAMP WITH LOCAL TIMEZONE
INTERVAL DAY TO SECOND
INTERVAL DAY TO MONTH
Summary

Objectives Summarized

In this lesson you have learned to:

- Create a table using TIMESTAMP and TIMESTAMP WITH TIME ZONE column data types
- Create a table using INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND column data types
- Give examples of organizations and personal situations where it is important to know to which time zone a date-time value refers
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Modifying a Table
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Explain why it is important to be able to modify a table

• Explain and provide an example for each of the DDL statements ALTER, DROP, RENAME, and TRUNCATE and the effect each has on tables and columns

• Construct a query and execute the ALTER TABLE commands ADD, MODIFY, and DROP

• Explain and perform FLASHBACK QUERY on a table

• Explain and perform FLASHBACK table operations
What Will I Learn?

Objectives (continued)

In this lesson, you will learn to:

• Track the changes to data over a period of time
• Explain the rationale for using TRUNCATE versus DELETE for tables
• Add a comment to a table using the COMMENT ON TABLE command
• Name the changes that can and cannot be made to modify a column
• Explain when and why the SET UNUSED statement is advantageous
Why Learn It?

Purpose
Remember the statement, "There is nothing permanent except change"? Wouldn't it be nice if we never made mistakes or needed to change anything? As you know by now, databases are dynamic entities. They probably wouldn't be very useful if they couldn't be changed.

Up to now you've created tables and made changes to the row data inside tables, but how do you make changes to the tables themselves? This lesson presents the DDL commands that are used to alter, rename, empty, or simply eliminate a table altogether.
Tell Me / Show Me

ALTER TABLE
ALTER TABLE statements are used to:
- Add a new column
- Modify an existing column
- Define a DEFAULT value for a column
- Drop a column.

You can add or modify a column in a table, but you cannot specify where the column appears. A newly added column always becomes the last column of the table.

Also, if a table already has rows of data and you add a new column to the table, the new column is initially null for all the rows.
Tell Me / Show Me

ALTER TABLE: Adding A Column
To add a new column, use the SQL syntax shown.

ALTER TABLE tablename
ADD (column name datatype [DEFAULT expression],
column name datatype [DEFAULT expression], ...)

For example:

ALTER TABLE copy_f_staffs
ADD (hire_date DATE DEFAULT SYSDATE);

ALTER TABLE copy_f_staffs
ADD (e_mail_address VARCHAR2(80));
Tell Me / Show Me

ALTER TABLE: Modifying A Column

Modifying a column can include changes to a column's data type, size, and DEFAULT value. Rules and restrictions when modifying a column are:

- You can increase the width or precision of a numeric column.
- You can increase the width of a character column.
- You can decrease the width of a column if you are decreasing the column size to a size that is no less than the largest width of the existing data, or, if the column contains only null values, or if the table has no rows.
ALTER TABLE: Modifying A Column (continued)

- You can change the data type only if the column contains null values.
- You can convert a CHAR column to VARCHAR2 or convert a VARCHAR2 column to CHAR only if the column contains null values or if you do not change the size.
- A change to the DEFAULT value of a column affects only later insertions to the table.
ALTER TABLE: Modifying A Column Example
Example: a table has been created with two columns:

CREATE TABLE mod_emp
  (last_name       VARCHAR2(20),
   salary               NUMBER(8,2));

Which of these modifications would be allowed, and which would not?
1. ALTER TABLE mod_emp MODIFY (last_name VARCHAR2(30));
2. ALTER TABLE mod_emp MODIFY (last_name VARCHAR2(10));
3. ALTER TABLE mod_emp MODIFY (salary NUMBER(10,2));
4. ALTER TABLE mod_emp MODIFY (salary NUMBER(8,2) DEFAULT 50);
Tell Me / Show Me

ALTER TABLE: Dropping A Column
When dropping a column the following rules apply:

- A column to be dropped may or may not contain data.
- Only one column can be dropped at a time.
- You can't drop all of the columns in a table; at least one column must remain.
- Once a column is dropped, the data values in it cannot be recovered.

SQL syntax:
ALTER TABLE tablename DROP COLUMN column name;
For example:
ALTER TABLE copy_f_staffs DROP COLUMN manager_target;
Tell Me / Show Me

SET UNUSED Columns

Dropping a column from a large table can take a long time. A quicker alternative is to mark the column as unusable. The column values remain in the database but cannot be accessed in any way, so the effect is the same as dropping the column.

In fact, you could add a new column to the database with the same name as the unused column. The unused columns are there, but invisible!

Syntax:
ALTER TABLE tablename SET UNUSED (column name);
Tell Me / Show Me

SET UNUSED Columns Example
Example:
ALTER TABLE copy_f_staffs
       SET UNUSED (manager_budget);

DROP UNUSED COLUMNS removes all columns currently marked as unused. You use this statement when you want to reclaim the extra disk space from unused columns in a table.
ALTER TABLE tablename DROP UNUSED COLUMNS;

Example:
ALTER TABLE copy_f_staffs DROP UNUSED COLUMNS;
### Tell Me / Show Me

**ALTER TABLE Summarized**

This chart summarizes the uses of the ALTER TABLE command.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Outcomes</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER TABLE tablename ADD (column name datatype [DEFAULT expression],</td>
<td>Adds a new column to a table</td>
<td>You cannot specify where the column is to appear in the table. It becomes the last column.</td>
</tr>
<tr>
<td>column name datatype [DEFAULT expression], …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTER TABLE tablename MODIFY (column name datatype [DEFAULT expression],</td>
<td>Used to change a column’s datatype, size, and default value</td>
<td>A change to the default value of a column affects only subsequent insertions to the table.</td>
</tr>
<tr>
<td>column name datatype, …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTER TABLE tablename DROP COLUMN column name;</td>
<td>Used to drop a column from a table</td>
<td>The table must have at least one column remaining in it after it is altered. Once dropped, the column cannot be recovered.</td>
</tr>
<tr>
<td>ALTER TABLE tablename SET UNUSED (column name);</td>
<td>Used to mark one or more columns so they can be dropped later</td>
<td>Does not restore disk space. Columns are treated as if they were dropped.</td>
</tr>
<tr>
<td>ALTER TABLE tablename DROP UNUSED COLUMNS</td>
<td>Removes from the table all columns currently marked as unused</td>
<td>Once set unused, there is no access to the columns; no data displayed using DESCRIBE. Permanent removal; no rollback.</td>
</tr>
</tbody>
</table>

Copyright © 2011, Oracle. All rights reserved.
DROP TABLE

The DROP TABLE statement removes the definition of an Oracle table. The database loses all the data in the table and all the indexes associated with it. When a DROP TABLE statement is issued:

- All data is deleted from the table.
- The table’s description is removed from the Data Dictionary.
- It may be irreversible! The Oracle Server does not question your decision.
Tell Me / Show Me

DROP TABLE (continued)

Only the creator of the table or a user with DROP ANY TABLE privilege (usually only the DBA) can remove a table.

DROP TABLE tablename;

DROP TABLE copy_f_staffs;
Tell Me / Show Me

FLASHBACK TABLE
If you drop a table by mistake there is a way of bringing that table and its data back.

Each database user has their own recyclebin into which dropped objects are now moved, and they can be recovered from here with the FLASHBACK TABLE command.

This command can be used to restore a table, a view, or an index that was dropped in error. The Syntax is::

FLASHBACK TABLE tablename TO BEFORE DROP;
Tell Me / Show Me

FLASHBACK TABLE (continued)
For example, if you drop the EMPLOYEES table in error, you can restore it by simply issuing the command:

FLASHBACK TABLE employees TO BEFORE DROP;

As the owner of a table you can issue the flashback command, and if the table that you are restoring had any indexes, then these are also restored.

It is possible to see what objects you have that you can restore, by querying the data dictionary view USER_RECYCLEBIN.
**Tell Me / Show Me**

**FLASHBACK TABLE (continued)**

The USER_RECYCLEBIN view can be queried like all other data dictionary views:

```
SELECT original_name, operation, droptime
FROM user_recyclebin
```

<table>
<thead>
<tr>
<th>ORIGINAL_NAME</th>
<th>OPERATION</th>
<th>DROPTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEES</td>
<td>DROP</td>
<td>2007-12-05 05:12:34</td>
</tr>
<tr>
<td>EMP_PK</td>
<td>DROP</td>
<td>2007-12-05 05:12:34</td>
</tr>
</tbody>
</table>

Once a table has been restored by the FLASHBACK TABLE command, it is no longer visible in the USER_RECYCLEBIN view.

Any indexes that were dropped along with the original table, will also have been restored.
Tell Me / Show Me

RENAME
To change the name of a table, use the RENAME statement. This can be done only by the owner of the object or by the DBA.

RENAME old_name to new_name;

Example:

RENAME copy_f_staffs to copy_fastfood_staffs;

We will see later that we can rename other types of object, such as views, sequences, and synonyms.
Tell Me / Show Me

TRUNCATE
Truncating a table removes all rows from a table and releases the storage space used by that table. When using the TRUNCATE TABLE statement:

• You cannot roll back row removal.
• You must be the owner of the table or have been given DROP ANY TABLE system privileges.

Syntax: TRUNCATE TABLE tablename;

The DELETE statement can also remove rows from a table, but it does not release storage space. TRUNCATE is faster than DELETE because it does not generate rollback information.
**Tell Me / Show Me**

**COMMENT ON TABLE**

You can add a comment of up to 2,000 characters about a column, table, or view by using the COMMENT statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

Comments on columns:
- ALL_COL_COMMENTS
- USER_COL_COMMENTS

Comments on tables:
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS
Tell Me / Show Me

COMMENT ON TABLE Examples

Syntax:
COMMENT ON TABLE tablename | COLUMN table.column IS 'place your comment here';

Example:
COMMENT ON TABLE employees IS 'Western Region only';
SELECT table_name, comments
FROM user_tab_comments;

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEES</td>
<td>Western Region only</td>
</tr>
</tbody>
</table>

If you want to drop a comment previously made on a table or column, use the empty string (''):

COMMENT ON TABLE employees IS '' ;
Tell Me / Show Me

Flashback Query
You may discover that somehow data in a table has been inappropriately changed. Luckily, Oracle has a facility that allows you to view row data at specific points in time, so you can compare different versions of a row over time.

This facility is very useful. Imagine for instance that someone accidently performs some DML on a table, and then executes a COMMIT on those changes. Oracle Application Express commits automatically, so mistakes are easily made.

You can use the FLASHBACK QUERY facility to examine what the rows looked like BEFORE those changes were applied.
Tell Me / Show Me

Flashback Query (continued)

When Oracle changes data, it always keeps a copy of what the amended data looked like before any changes were made. So it keeps a copy of the old column value for a column update, it keeps the entire row for a delete and it keeps nothing for an insert statement.

These old copies are held in a special place called the UNDO tablespace. Users can access this special area of the Database using a flashback query.
Tell Me / Show Me

Flashback Query (continued)
You can look at older versions of data by using the VERSIONS clause in a SELECT statement. For example:

```sql
SELECT employee_id,
       first_name || ' ' || last_name AS "NAME",
       versions_operation AS "OPERATION",
       versions_starttime AS "START_DATE",
       versions_endtime AS "END_DATE",
       salary
FROM employees
   VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE employee_id = 1;
```

The SCN number referred to in the above query means the System Change Number, and is a precise identification of time in the database. It is a sequential number incremented and maintained by the database itself.
Tell Me / Show Me

Flashback Query (continued)
The best way to demonstrate FLASHBACK QUERY is with an example.
The contents of the employees table is as follows for employee_id 1.

```sql
SELECT employee_id,
       first_name || ' ' || last_name AS "NAME",
       versions_operation AS "OPERATION",
       versions_starttime AS "START_DATE",
       versions_endtime AS "END_DATE",
       salary
FROM employees
VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE employee_id = 1;
```

<table>
<thead>
<tr>
<th>Results</th>
<th>Explain</th>
<th>Describe</th>
<th>Saved SQL</th>
<th>History</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no data found</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Then we create the employee:

```sql
INSERT INTO employees
VALUES (1, 'Natacha', 'Hansen', 'N.Hansen@Mail.com', '+44123 1234 1234', '07-SEP-1998', 'AD_VP', 12000, null, 100, 90);
```

<table>
<thead>
<tr>
<th>Results</th>
<th>Explain</th>
<th>Describe</th>
<th>Saved SQL</th>
<th>History</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 row(s) inserted.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flashback Query (continued)

SELECT employee_id,
    first_name || ' ' || last_name AS "NAME",
    versions_operation AS "OPERATION",
    versions_starttime AS "START_DATE",
    versions_endtime AS "END_DATE",
    salary
FROM employees
    VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 1;

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>NAME</th>
<th>OPERATION</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Natacha Hansen</td>
<td>I</td>
<td>23-JUN-09 06.51.58 AM</td>
<td>-</td>
<td>12000</td>
</tr>
</tbody>
</table>
Flashback Query (continued)
Then you can update the row:

```sql
UPDATE employees
SET salary = 1
WHERE employee_id = 1;
```

```
1 row(s) updated.
```

```
<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>NAME</th>
<th>OPERATION</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Natacha Hansen</td>
<td>U</td>
<td>23-JUN-09 06.57.01 AM</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>I</td>
<td>23-JUN-09 06.51.58 AM</td>
<td>23-JUN-09 06.57.01 AM</td>
<td>12000</td>
</tr>
</tbody>
</table>
```
Flashback Query (continued)

Then you can delete the row:

```sql
DELETE employees
WHERE employee_id = 1;
```

1 row(s) deleted.

```sql
SELECT employee_id,
  first_name || ' ' || last_name AS "NAME",
  versions_operation AS "OPERATION",
  versions_starttime AS "START_DATE",
  versions_endtime AS "END_DATE",
  salary
FROM employees
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 1;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>NAME</th>
<th>OPERATION</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Natacha Hansen</td>
<td>D</td>
<td>23-JUN-09 07.00.10 AM</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Natacha Hansen</td>
<td>U</td>
<td>23-JUN-09 06.57.01 AM</td>
<td>23-JUN-09 07.00.10 AM</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Natacha Hansen</td>
<td>I</td>
<td>23-JUN-09 06.51.58 AM</td>
<td>23-JUN-09 06.57.01 AM</td>
<td>12000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Flashback Query (continued)

The result from the last query on the previous slide is only available when using Flashback query, i.e. the VERSIONS clause. If you attempt a normal search from employee_id = 1 following the delete statement, you would have received the normal error, No Data Found.

```
SELECT employee_id, salary
FROM employees
WHERE employee_id = 1
```

no data found
Summary

Objectives Summarized
In this lesson you have learned to:

• Explain why it is important to be able to modify a table
• Explain and provide an example for each of the DDL statements ALTER, DROP, RENAME, and TRUNCATE and the effect each has on tables and columns
• Construct a query and execute the ALTER TABLE commands ADD, MODIFY, and DROP
• Explain and perform FLASHBACK QUERY on a table
• Explain and perform FLASHBACK table operations
Objectives Summarized (continued)
In this lesson you have learned to:

• Track the changes to data over a period of time
• Explain the rationale for using TRUNCATE versus DELETE for tables
• Add a comment to a table using the COMMENT ON TABLE command
• Name the changes that can and cannot be made to modify a column
• Explain when and why the SET UNUSED statement is advantageous
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Ensuring Quality Query Results
What Will I Learn?

In this lesson, you will learn to:

• Create a query to produce specified data
• Modify a query to produce specified data
Why Learn It?

Purpose

You’ve learned the syntax rules for generating a SQL query but are you sure you are producing the desired data? Looking at the desired output and then figuring out the query to generate that output helps you to gain confidence that your query results are what you expect.
Tell Me / Show Me

Write the Query

**Problem:**
Produce a listing of all tables whose first two characters in the name of the table is D_. The tables must be owned by the current Oracle User.

**Tables Used:**
User_tables

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_COS</td>
</tr>
<tr>
<td>D_CLIENTS</td>
</tr>
<tr>
<td>D_EVENTS</td>
</tr>
<tr>
<td>D_JOB_ASSIGNMENTS</td>
</tr>
<tr>
<td>D_PACKAGES</td>
</tr>
<tr>
<td>D_PARTNERS</td>
</tr>
<tr>
<td>D_PLAY_LIST_ITEMS</td>
</tr>
<tr>
<td>D_SONGS</td>
</tr>
<tr>
<td>D_THEMES</td>
</tr>
<tr>
<td>D_TRACK_LISTINGS</td>
</tr>
<tr>
<td>D_TYPES</td>
</tr>
<tr>
<td>D_VENUES</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Create a listing of the first character of the first name, a space and the last name of all employees.

Tables Used:
Employees

Query Result:
Tell Me / Show Me
Write the Query

Problem:
Produce a listing of employee first names concatenated with a space and then the last name and the email of employees where the email address contains the string ‘IN’.

Tables Used:
Employees

Query Result:

<table>
<thead>
<tr>
<th>Employee Names</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shelley Higgins</td>
<td>SHIGGINS</td>
</tr>
<tr>
<td>Steven King</td>
<td>SKING</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Produce a list of ‘smallest’ last name and the ‘highest’ last name from the employees table.

Tables Used:
Employees

Query Result:

<table>
<thead>
<tr>
<th>First Last Name</th>
<th>Last Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>Zlotkey</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Produce a list of weekly salaries from the employees table where the weekly salary is between 700 and 3000. The salaries should be formatted to include a $-sign and have two decimal points like: $9999.99.

Tables Used:
Employees
Tell Me / Show Me

Write the Query

Problem:
Produce a listing of employees and their related job titles sorted by job_title.

Tables Used:
Employees, Jobs

Query Result:

<table>
<thead>
<tr>
<th>Employee Name</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>S Higgins</td>
<td>Accounting Manager</td>
</tr>
<tr>
<td>J Whalen</td>
<td>Administration Assistant</td>
</tr>
<tr>
<td>L De Haan</td>
<td>Administration Vice President</td>
</tr>
<tr>
<td>N Kochhar</td>
<td>Administration Vice President</td>
</tr>
<tr>
<td>M Hartstein</td>
<td>Marketing Manager</td>
</tr>
<tr>
<td>P Fay</td>
<td>Marketing Representative</td>
</tr>
<tr>
<td>S King</td>
<td>President</td>
</tr>
<tr>
<td>A Hunold</td>
<td>Programmer</td>
</tr>
<tr>
<td>B Ernst</td>
<td>Programmer</td>
</tr>
<tr>
<td>D Lorentz</td>
<td>Programmer</td>
</tr>
<tr>
<td>W Gietz</td>
<td>Public Accountant</td>
</tr>
<tr>
<td>E Zlotkey</td>
<td>Sales Manager</td>
</tr>
<tr>
<td>J Taylor</td>
<td>Sales Representative</td>
</tr>
<tr>
<td>K Grant</td>
<td>Sales Representative</td>
</tr>
<tr>
<td>E Abel</td>
<td>Sales Representative</td>
</tr>
<tr>
<td>T Rajs</td>
<td>Stock Clerk</td>
</tr>
<tr>
<td>C Davies</td>
<td>Stock Clerk</td>
</tr>
<tr>
<td>P Vargas</td>
<td>Stock Clerk</td>
</tr>
<tr>
<td>R Matos</td>
<td>Stock Clerk</td>
</tr>
<tr>
<td>K Mourgos</td>
<td>Stock Manager</td>
</tr>
</tbody>
</table>
Tell Me / Show Me
Write the Query

Problem:
Create a list of employees’ jobs and salary ranges within the jobs. Join the employees and jobs tables. List the lowest and highest salary range within each job with a dash to separate the salaries like this: 100 – 200.

Tables Used:
Employees, Jobs

Query Result:
Tell Me / Show Me

Write the Query

Problem:
Using an ANSI join method, produce a list of employees’ Initial and last name and department names. Make sure the tables are joined on all of the foreign keys declared between the two tables.

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>Employee Name</th>
<th>Department Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Kochhar</td>
<td>Executive</td>
</tr>
<tr>
<td>L De Haan</td>
<td>Executive</td>
</tr>
<tr>
<td>W Gietz</td>
<td>Accounting</td>
</tr>
<tr>
<td>E Abel</td>
<td>Sales</td>
</tr>
<tr>
<td>J Taylor</td>
<td>Sales</td>
</tr>
<tr>
<td>T Rajs</td>
<td>Shipping</td>
</tr>
<tr>
<td>C Davies</td>
<td>Shipping</td>
</tr>
<tr>
<td>R Matos</td>
<td>Shipping</td>
</tr>
<tr>
<td>P Vargas</td>
<td>Shipping</td>
</tr>
<tr>
<td>B Ernst</td>
<td>IT</td>
</tr>
<tr>
<td>D Lorentz</td>
<td>IT</td>
</tr>
<tr>
<td>P Fay</td>
<td>Marketing</td>
</tr>
</tbody>
</table>
Problem:
Change the previous listing to join only on the department_id column.

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>Employee Name</th>
<th>Department Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>J Whalen</td>
<td>Administration</td>
</tr>
<tr>
<td>M Hartstein</td>
<td>Marketing</td>
</tr>
<tr>
<td>P Fay</td>
<td>Marketing</td>
</tr>
<tr>
<td>C Davos</td>
<td>Shipping</td>
</tr>
<tr>
<td>P Vargas</td>
<td>Shipping</td>
</tr>
<tr>
<td>T Rajs</td>
<td>Shipping</td>
</tr>
<tr>
<td>K Mourgos</td>
<td>Shipping</td>
</tr>
<tr>
<td>R Matos</td>
<td>Shipping</td>
</tr>
<tr>
<td>A Hunold</td>
<td>IT</td>
</tr>
<tr>
<td>B Ernst</td>
<td>IT</td>
</tr>
<tr>
<td>D Lorentz</td>
<td>IT</td>
</tr>
<tr>
<td>J Taylor</td>
<td>Sales</td>
</tr>
<tr>
<td>E Zbtky</td>
<td>Sales</td>
</tr>
<tr>
<td>E Abel</td>
<td>Sales</td>
</tr>
<tr>
<td>L De Haan</td>
<td>Executive</td>
</tr>
<tr>
<td>S King</td>
<td>Executive</td>
</tr>
<tr>
<td>N Kochhar</td>
<td>Executive</td>
</tr>
<tr>
<td>S Higgins</td>
<td>Accounting</td>
</tr>
<tr>
<td>W Gietz</td>
<td>Accounting</td>
</tr>
</tbody>
</table>
**Problem:**
Produce a listing of employee last names and the word nobody or somebody depending on if the employee has a manager or not. Use the Oracle DECODE function to create the listing.

**Tables Used:**
Employees

**Query Result:**

<table>
<thead>
<tr>
<th>Works for</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nobody</td>
<td>King</td>
</tr>
<tr>
<td>Somebody</td>
<td>Kochhar</td>
</tr>
<tr>
<td>Somebody</td>
<td>De Haan</td>
</tr>
<tr>
<td>Somebody</td>
<td>Whalen</td>
</tr>
<tr>
<td>Somebody</td>
<td>Higgins</td>
</tr>
<tr>
<td>Somebody</td>
<td>Gietz</td>
</tr>
<tr>
<td>Somebody</td>
<td>Zlotkey</td>
</tr>
<tr>
<td>Somebody</td>
<td>Abel</td>
</tr>
<tr>
<td>Somebody</td>
<td>Taylor</td>
</tr>
<tr>
<td>Somebody</td>
<td>Grant</td>
</tr>
<tr>
<td>Somebody</td>
<td>Mourgos</td>
</tr>
<tr>
<td>Somebody</td>
<td>Rajs</td>
</tr>
<tr>
<td>Somebody</td>
<td>Davies</td>
</tr>
<tr>
<td>Somebody</td>
<td>Matos</td>
</tr>
<tr>
<td>Somebody</td>
<td>Vargas</td>
</tr>
<tr>
<td>Somebody</td>
<td>Hurnold</td>
</tr>
<tr>
<td>Somebody</td>
<td>Ernst</td>
</tr>
<tr>
<td>Somebody</td>
<td>Lorentz</td>
</tr>
<tr>
<td>Somebody</td>
<td>Hartstein</td>
</tr>
<tr>
<td>Somebody</td>
<td>Fay</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Fix the Query

Problem:
Create a list of employee first initial, last name, salary and a yes or no to show if an employee makes a commission or not. Fix the query to produce the result.

QUERY:
SELECT SUBSTR(first_name,1,1)||' '||last_name, "Employee Name",
salary "Salary",
DEC(commission_pct NULL, 'No', 'Yes') 'Commission'
FROM   employees;

Query Result:
Tell Me / Show Me

Write the Query

Problem:
Produce a listing of employees’ last name, their department name and city and state_province. Include departments without employees. An outer join is required.

Tables Used:
Employees, Departments, Locations

Query Result:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_NAME</th>
<th>CITY</th>
<th>STATE_PROVINCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>Sales</td>
<td>Oxford</td>
<td>Oxford</td>
</tr>
<tr>
<td>Davies</td>
<td>Shipping</td>
<td>South San Francisco</td>
<td>California</td>
</tr>
<tr>
<td>De Haan</td>
<td>Executive</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
<tr>
<td>Ernst</td>
<td>IT</td>
<td>Southlake</td>
<td>Texas</td>
</tr>
<tr>
<td>Fay</td>
<td>Marketing</td>
<td>Toronto</td>
<td>Ontario</td>
</tr>
<tr>
<td>Gietz</td>
<td>Accounting</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
<tr>
<td>Hartstein</td>
<td>Marketing</td>
<td>Toronto</td>
<td>Ontario</td>
</tr>
<tr>
<td>Higgins</td>
<td>Accounting</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
<tr>
<td>Hunold</td>
<td>IT</td>
<td>Southlake</td>
<td>Texas</td>
</tr>
<tr>
<td>King</td>
<td>Executive</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
<tr>
<td>Kochhar</td>
<td>Executive</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
<tr>
<td>Lorentz</td>
<td>IT</td>
<td>Southlake</td>
<td>Texas</td>
</tr>
<tr>
<td>Matos</td>
<td>Shipping</td>
<td>South San Francisco</td>
<td>California</td>
</tr>
<tr>
<td>Murgos</td>
<td>Shipping</td>
<td>South San Francisco</td>
<td>California</td>
</tr>
<tr>
<td>Rajs</td>
<td>Shipping</td>
<td>South San Francisco</td>
<td>California</td>
</tr>
<tr>
<td>Taylor</td>
<td>Sales</td>
<td>Oxford</td>
<td>Oxford</td>
</tr>
<tr>
<td>Vargos</td>
<td>Shipping</td>
<td>South San Francisco</td>
<td>California</td>
</tr>
<tr>
<td>Whalen</td>
<td>Administration</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>Sales</td>
<td>Oxford</td>
<td>Oxford</td>
</tr>
<tr>
<td></td>
<td>Contracting</td>
<td>Seattle</td>
<td>Washington</td>
</tr>
</tbody>
</table>
Tell Me / Show Me
Write the Query

Problem:
Create a listing of employee first and last names, and the first occurrence of: commission_pct, manager_id or -1. So if an employee gets commission, display the commission_pct column, if no commission then display his manager_id and if he has neither commission nor manager then the number -1.

Tables Used:
Employees

Query Result:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Which function???</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steven</td>
<td>King</td>
<td>-1</td>
</tr>
<tr>
<td>Neena</td>
<td>Kochhar</td>
<td>100</td>
</tr>
<tr>
<td>Lex</td>
<td>De Haan</td>
<td>100</td>
</tr>
<tr>
<td>Jennifer</td>
<td>Whalen</td>
<td>101</td>
</tr>
<tr>
<td>Shelley</td>
<td>Higgins</td>
<td>101</td>
</tr>
<tr>
<td>William</td>
<td>Gietz</td>
<td>205</td>
</tr>
<tr>
<td>Eleni</td>
<td>Zlotkey</td>
<td>.2</td>
</tr>
<tr>
<td>Elliot</td>
<td>Abel</td>
<td>.3</td>
</tr>
<tr>
<td>Jonathon</td>
<td>Taylor</td>
<td>.2</td>
</tr>
<tr>
<td>Kimberely</td>
<td>Grant</td>
<td>.15</td>
</tr>
<tr>
<td>Kevin</td>
<td>Mourgos</td>
<td>100</td>
</tr>
<tr>
<td>Trenna</td>
<td>Rajs</td>
<td>124</td>
</tr>
<tr>
<td>Curtis</td>
<td>Davies</td>
<td>124</td>
</tr>
<tr>
<td>Randall</td>
<td>Matos</td>
<td>124</td>
</tr>
<tr>
<td>Peter</td>
<td>Vargas</td>
<td>124</td>
</tr>
<tr>
<td>Alexander</td>
<td>Hunold</td>
<td>102</td>
</tr>
<tr>
<td>Bruce</td>
<td>Ernst</td>
<td>103</td>
</tr>
<tr>
<td>Diana</td>
<td>Lorentz</td>
<td>103</td>
</tr>
<tr>
<td>Michael</td>
<td>Hartstein</td>
<td>100</td>
</tr>
<tr>
<td>Pat</td>
<td>Fay</td>
<td>201</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Fix the Query

Problem:
Produce a listing of employees’ last names, their salaries and their related job_grade for all employees working in departments with a department_id greater than 50.

QUERY:
SELECT e.last_name, e.salary, j.grade_level
FROM employees AS e JOIN job_grades AS j
ON e.salary BETWEEN j.highest_sal AND j.lowest_sal
AND e.department_id < 50

Query Result:
Tell Me / Show Me

Write the Query

Problem:
Produce a list of all employee last names and all department names. Include both employees without departments and departments without employees.

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>Administration</td>
</tr>
<tr>
<td>Fay</td>
<td>Marketing</td>
</tr>
<tr>
<td>Hartstein</td>
<td>Marketing</td>
</tr>
<tr>
<td>Vargas</td>
<td>Shipping</td>
</tr>
<tr>
<td>Matos</td>
<td>Shipping</td>
</tr>
<tr>
<td>Davios</td>
<td>Shipping</td>
</tr>
<tr>
<td>Rajs</td>
<td>Shipping</td>
</tr>
<tr>
<td>Mourgos</td>
<td>Shipping</td>
</tr>
<tr>
<td>Lorenz</td>
<td>IT</td>
</tr>
<tr>
<td>Ernst</td>
<td>IT</td>
</tr>
<tr>
<td>Hunold</td>
<td>IT</td>
</tr>
<tr>
<td>Taylor</td>
<td>Sales</td>
</tr>
<tr>
<td>Abel</td>
<td>Sales</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>Sales</td>
</tr>
<tr>
<td>De Haan</td>
<td>Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td>Executive</td>
</tr>
<tr>
<td>King</td>
<td>Executive</td>
</tr>
<tr>
<td>Gietz</td>
<td>Accounting</td>
</tr>
<tr>
<td>Higgins</td>
<td>Accounting</td>
</tr>
<tr>
<td>Grant</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Contracting</td>
</tr>
</tbody>
</table>
Problem:
Create a treewalking listing of employees’ last names, their manager’s last names and their position in the company, ie, the top level manager has position 1, this manager’s subordinates position 2, their subordinates position 3 and so on. Start the listing with employee number 100.

Tables Used:
Employees

<table>
<thead>
<tr>
<th>POSITION</th>
<th>LAST_NAME</th>
<th>MANAGER_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>King</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Kochhar</td>
<td>King</td>
</tr>
<tr>
<td>3</td>
<td>Whalen</td>
<td>Kochhar</td>
</tr>
<tr>
<td>3</td>
<td>Higgins</td>
<td>Kochhar</td>
</tr>
<tr>
<td>4</td>
<td>Gietz</td>
<td>Higgins</td>
</tr>
<tr>
<td>2</td>
<td>De Haan</td>
<td>King</td>
</tr>
<tr>
<td>3</td>
<td>Hunold</td>
<td>De Haan</td>
</tr>
<tr>
<td>4</td>
<td>Ernst</td>
<td>Hunold</td>
</tr>
<tr>
<td>4</td>
<td>Lorentz</td>
<td>Hunold</td>
</tr>
<tr>
<td>2</td>
<td>Mourgos</td>
<td>King</td>
</tr>
<tr>
<td>3</td>
<td>Rajs</td>
<td>Mourgos</td>
</tr>
<tr>
<td>3</td>
<td>Davias</td>
<td>Mourgos</td>
</tr>
<tr>
<td>3</td>
<td>Matos</td>
<td>Mourgos</td>
</tr>
<tr>
<td>3</td>
<td>Vargas</td>
<td>Mourgos</td>
</tr>
<tr>
<td>2</td>
<td>Zlotkey</td>
<td>King</td>
</tr>
<tr>
<td>3</td>
<td>Abel</td>
<td>Zlotkey</td>
</tr>
<tr>
<td>3</td>
<td>Taylor</td>
<td>Zlotkey</td>
</tr>
<tr>
<td>3</td>
<td>Grant</td>
<td>Zlotkey</td>
</tr>
<tr>
<td>2</td>
<td>Hartstein</td>
<td>King</td>
</tr>
<tr>
<td>3</td>
<td>Fay</td>
<td>Hartstein</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Produce a listing of the earliest hire date, the latest hire date and the number of employees from the employees table.

Tables Used:
Employees

Query Result:

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
<th>No of Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>29-JAN-00</td>
<td>17-JUN-87</td>
<td>20</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Produce a list of department names and the departmental costs (salaries added up). Include only departments whose salary costs are between 15000 and 31000, and sort the listing by the cost.

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>DEPARTMENT_NAME</th>
<th>SALARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shipping</td>
<td>17500</td>
</tr>
<tr>
<td>Marketing</td>
<td>19000</td>
</tr>
<tr>
<td>IT</td>
<td>19200</td>
</tr>
<tr>
<td>Accounting</td>
<td>20300</td>
</tr>
<tr>
<td>Sales</td>
<td>30100</td>
</tr>
</tbody>
</table>
Tell Me / Show Me
Write the Query

Problem:
Produce a listing of department names, the manager id and manager name (employee last name) of that department and the average salary of each department.

Tables Used:
Employees, Departments, Employees

Query Result:

<table>
<thead>
<tr>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>MANAGER_NAME</th>
<th>AVG_DEPT_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>200</td>
<td>Whalen</td>
<td>4400</td>
</tr>
<tr>
<td>IT</td>
<td>103</td>
<td>Hunold</td>
<td>6400</td>
</tr>
<tr>
<td>Marketing</td>
<td>201</td>
<td>Hartstein</td>
<td>9600</td>
</tr>
<tr>
<td>Shipping</td>
<td>124</td>
<td>Maurgos</td>
<td>3500</td>
</tr>
<tr>
<td>Executive</td>
<td>100</td>
<td>King</td>
<td>19333</td>
</tr>
<tr>
<td>Accounting</td>
<td>205</td>
<td>Higgins</td>
<td>10150</td>
</tr>
<tr>
<td>Sales</td>
<td>149</td>
<td>Zlochay</td>
<td>10033</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Show the highest average salary for the departments in the employees table. Round the result to the nearest whole number.

Tables Used:
Employees

Query Result:

<table>
<thead>
<tr>
<th>Highest Avg Sal for Depts</th>
</tr>
</thead>
<tbody>
<tr>
<td>19333</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Create a listing of department names and their monthly costs (salaries added up).

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>Department Name</th>
<th>Monthly Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>4400</td>
</tr>
<tr>
<td>Accounting</td>
<td>20300</td>
</tr>
<tr>
<td>IT</td>
<td>19200</td>
</tr>
<tr>
<td>Executive</td>
<td>58000</td>
</tr>
<tr>
<td>Shipping</td>
<td>17500</td>
</tr>
<tr>
<td>Sales</td>
<td>30100</td>
</tr>
<tr>
<td>Marketing</td>
<td>19000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Create a listing of department names, job_ids, the monthly salary cost for the departments and also the sum of salaries per department, independent of the job_ids and a grand total at the bottom of the listing. (Hint: Rollup)

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>Department Name</th>
<th>Job Title</th>
<th>Monthly Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>Accounting</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>Accounting</td>
<td>-</td>
<td>20300</td>
</tr>
<tr>
<td>Administration</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>Administration</td>
<td>-</td>
<td>4400</td>
</tr>
<tr>
<td>Executive</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>Executive</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>Executive</td>
<td>-</td>
<td>58000</td>
</tr>
<tr>
<td>IT</td>
<td>IT_PROG</td>
<td>19200</td>
</tr>
<tr>
<td>IT</td>
<td>-</td>
<td>19200</td>
</tr>
<tr>
<td>Marketing</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>Marketing</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>Marketing</td>
<td>-</td>
<td>19000</td>
</tr>
<tr>
<td>Sales</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>Sales</td>
<td>SA_REP</td>
<td>19600</td>
</tr>
<tr>
<td>Sales</td>
<td>-</td>
<td>30100</td>
</tr>
<tr>
<td>Shipping</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>Shipping</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>Shipping</td>
<td>-</td>
<td>17500</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>168500</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Create a listing of department names, job_ids, the monthly salary cost for the departments and also the sum of salaries per department, per job_id and a grand total sum of salaries at the bottom of the listing.
(Hint: Cube)

Tables Used:
Employees, Departments

Query Result:

<table>
<thead>
<tr>
<th>Department Name</th>
<th>Job Title</th>
<th>Monthly Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>Accounting</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>Accounting</td>
<td>-</td>
<td>20300</td>
</tr>
<tr>
<td>Administration</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>Administration</td>
<td>-</td>
<td>4400</td>
</tr>
<tr>
<td>Executive</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>Executive</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>Executive</td>
<td>-</td>
<td>68000</td>
</tr>
<tr>
<td>IT</td>
<td>IT_PROG</td>
<td>15200</td>
</tr>
<tr>
<td>IT</td>
<td>-</td>
<td>15200</td>
</tr>
<tr>
<td>Marketing</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>Marketing</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>Marketing</td>
<td>-</td>
<td>10000</td>
</tr>
<tr>
<td>Sales</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>Sales</td>
<td>SA_REP</td>
<td>19600</td>
</tr>
<tr>
<td>Sales</td>
<td>-</td>
<td>30100</td>
</tr>
<tr>
<td>Shipping</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>Shipping</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>Shipping</td>
<td>-</td>
<td>17500</td>
</tr>
<tr>
<td>-</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>-</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>-</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>-</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>-</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>-</td>
<td>IT_PROG</td>
<td>15200</td>
</tr>
<tr>
<td>-</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>-</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>-</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>-</td>
<td>SA_REP</td>
<td>19600</td>
</tr>
<tr>
<td>-</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>-</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>166500</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Expand the previous listing to also show if the department_id or job_id was used to create the subtotals shown in the output.
(Hint: Cube, Grouping)

Tables Used:
Employees, Departments

Query Result:
Tell Me / Show Me

Write the Query

Problem:
Generate a report that lists the monthly salary costs for the department name and job titles as one result and also, in the same query a monthly salary cost per city. (Hint: Grouping Sets)

Tables Used:
Employees, Departments, Locations

Query Result:

<table>
<thead>
<tr>
<th>Department Name</th>
<th>Job Title</th>
<th>Location City</th>
<th>Monthly Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>AC_ACCOUNT</td>
<td>-</td>
<td>8300</td>
</tr>
<tr>
<td>Accounting</td>
<td>AC_MGR</td>
<td>-</td>
<td>12000</td>
</tr>
<tr>
<td>Administration</td>
<td>AD_ASST</td>
<td>-</td>
<td>4400</td>
</tr>
<tr>
<td>Executive</td>
<td>AD_PRES</td>
<td>-</td>
<td>24000</td>
</tr>
<tr>
<td>Executive</td>
<td>AD_VP</td>
<td>-</td>
<td>34000</td>
</tr>
<tr>
<td>IT</td>
<td>IT_PROG</td>
<td>-</td>
<td>15200</td>
</tr>
<tr>
<td>Marketing</td>
<td>MK_MAN</td>
<td>-</td>
<td>13000</td>
</tr>
<tr>
<td>Marketing</td>
<td>MKREP</td>
<td>-</td>
<td>6000</td>
</tr>
<tr>
<td>Sales</td>
<td>SA_MAN</td>
<td>-</td>
<td>10500</td>
</tr>
<tr>
<td>Sales</td>
<td>SA_REP</td>
<td>-</td>
<td>19600</td>
</tr>
<tr>
<td>Shipping</td>
<td>ST_CLERK</td>
<td>-</td>
<td>11700</td>
</tr>
<tr>
<td>Shipping</td>
<td>ST_MAN</td>
<td>-</td>
<td>5800</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Oxford</td>
<td>30100</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Seattle</td>
<td>82700</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>South San Francisco</td>
<td>17500</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Southlake</td>
<td>19200</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Toronto</td>
<td>19000</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Query

Problem:
Create a report that lists employee names as shown and department ids. In the same report, list the department ids and department names and finally the cities. The rows should not be joined, just all listed in the same report.
(Hint: Union)

Tables Used:
Employees, Departments, Locations
Tell Me / Show Me

Write the Query

Problem:
Create a report that lists the code, the name, the hours served, start date, end date and gift of both the menus and promotional menus. Do not join the tables, but list in one report output from both tables.

Tables Used:
f_regularMenus, f_promotionalMenus

Query Result:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>HOURS_SERVED</th>
<th>Start Date</th>
<th>End Date</th>
<th>Give Away</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Breakfast</td>
<td>6-11am</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>Back to School</td>
<td>-</td>
<td>01-SEP-04</td>
<td>30-SEP-04</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>Valentines Special</td>
<td>-</td>
<td>16-FEB-04</td>
<td>15-FEB-04</td>
<td>small box of chocolates</td>
</tr>
<tr>
<td>20</td>
<td>Lunch/Dinner</td>
<td>11-9pm</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Tell Me / Show Me
Write the Query

Problem:
Create a report that lists first initial and last name, salaries and department names of employees earning more than the average for their departments.

Tables Used:
Departments, Employees

Query Result:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
<th>Department Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>M Hartstein</td>
<td>13000</td>
<td>Marketing</td>
</tr>
<tr>
<td>K Mourgos</td>
<td>5800</td>
<td>Shipping</td>
</tr>
<tr>
<td>A Hunold</td>
<td>9000</td>
<td>IT</td>
</tr>
<tr>
<td>E Zlotkey</td>
<td>10500</td>
<td>Sales</td>
</tr>
<tr>
<td>E Abel</td>
<td>11000</td>
<td>Sales</td>
</tr>
<tr>
<td>S King</td>
<td>24000</td>
<td>Executive</td>
</tr>
<tr>
<td>S Higgins</td>
<td>12000</td>
<td>Accounting</td>
</tr>
</tbody>
</table>
Summary

Objectives Summarized
In this lesson you have learned to:

• Create a query to produce specified data
• Modify a query to produce specified data
Summary

Practice Guide
There is no lesson practice.
Defining NOT NULL and UNIQUE Constraints
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Define the term "constraint" as it relates to data integrity
• State when it is possible to define a constraint at the column level, and when it is possible at the table level
• State why it is important to give meaningful names to constraints
• State which data integrity rules are enforced by NOT NULL and UNIQUE constraints
• Write a CREATE TABLE statement which includes NOT NULL and UNIQUE constraints at the table and column levels
• Explain how constraints are created at the time of table creation
Why Learn It?

Purpose
If you think about it, how would society function without rules? It is a rule to stop at a red traffic light. Would it be safe to drive without this rule? For databases, it is a rule that a foreign-key value cannot be entered without first entering a primary-key value. What do you think would happen to a database if this rule wasn't enforced?

A database is only as reliable as the data that is in it. Constraints are used to prevent invalid data entry into tables. Would it make sense to have negative salary values or six students with the same student ID or two tables that no longer reference each other? Without rules, how could you trust the integrity of the database? In the next three lessons, you will study how to create the constraints that enforce the "rules." You will also learn how to manage them and view constraints definitions in the data dictionary.
Tell Me / Show Me

Constraints In General

So, what exactly is a constraint? Think of constraints as database rules. All constraint definitions are stored in the data dictionary. Constraints prevent the deletion of a table if there are dependencies from other tables. Constraints enforce rules on the data whenever a row is inserted, updated, or deleted from a table. Constraints are important and naming them is also important. Although you could name a constraint "bubbles" or "squeak," you'd soon find it difficult to distinguish one constraint from another and would end up redoing a lot of work.
Creating Constraints
Recall the SQL syntax for creating a table. In the CREATE TABLE statement shown below, each column and its data type is defined. You use the CREATE TABLE statement to establish constraints for each column in the table.

There are two different places in the CREATE TABLE statement that you can specify the constraint details:
• At the column level next to the name and data type
• At the table level after all the column names are listed

CREATE TABLE clients
(client_number NUMBER(4),
first_name VARCHAR2(14),
last_name VARCHAR2(13));
Tell Me / Show Me

Constraints at the Column Level
A column-level constraint references a single column. To establish a column-level constraint the constraint must be defined in the CREATE TABLE statement as part of the column definition. Examine the following SQL statement that establishes a column-level constraint.

```sql
CREATE TABLE clients
(client_number NUMBER(4) CONSTRAINT clients_client_num_pk PRIMARY KEY,
first_name VARCHAR2(14),
last_name VARCHAR2(13));
```

The name of the constraint is clients_client_num_pk. It enforces the business rule that the client_number is the primary key of the clients table.
Defining NOT NULL and UNIQUE Constraints

Tell Me / Show Me

Naming Constraints

Every constraint in the database has a name. When a constraint is created, it can be given a name, such as clients_client_num_pk, or given no name, in which case the system gives the constraint a name, such as SYS_C00585417.

A naming convention can be the combination of the tablename abbreviated and a column name abbreviated followed by the constraint abbreviation:

table-name_column-name_constraint-type.

If the reserved word CONSTRAINT is used in the CREATE TABLE definition, you must give the constraint a name. Constraint names are limited to 30 characters.
Tell Me / Show Me

Naming Constraints at the Column Level
It is best to name constraints yourself because system-generated names are not easy to recognize. Look at this table definition:

CREATE TABLE clients
(client_number NUMBER(4),
 last_name VARCHAR2(13),
 email VARCHAR2(80));

According to our naming convention:

- A primary key constraint on client_number would be named clients_client_no_pk
- A not null constraint on last_name would be named clients_last_name_nn
- A unique constraint on e-mail_address would be named clients_email_uk
Defining NOT NULL and UNIQUE Constraints

Tell Me / Show Me

Constraint Naming Example
This example shows both a user-named constraint and a system-named constraint:

CREATE TABLE clients
(client_number   NUMBER(4)  CONSTRAINT
    clients_client_num_pk PRIMARY KEY,
last_name         VARCHAR2(13)  NOT NULL,
email                 VARCHAR2(80));

Two constraints have been created:
• a user-named constraint named clients_client_num_pk, to enforce the rule that client_number is the primary key
• a system-named constraint named SYS_Cn (where n is a unique integer) to enforce the rule that last_names cannot be null.
Tell Me / Show Me

Constraints at the Table Level
Table-level constraints are listed separately from the column definitions in the CREATE TABLE statement. Table-level constraint definitions are listed after all the table columns have been defined. In the example shown, the unique constraint is listed last in the CREATE TABLE statement.

CREATE TABLE clients (  
  client_number NUMBER(6) NOT NULL,  
  first_name VARCHAR2(20),  
  last_name VARCHAR2(20),  
  phone VARCHAR2(20),  
  email VARCHAR2(10) NOT NULL,  
  CONSTRAINT clients_phone_email_uk UNIQUE (email,phone));
Defining NOT NULL and UNIQUE Constraints

Tell Me / Show Me

Basic Rules For Constraints

• Constraints that refer to more than one column (a composite key) must be defined at the table level
• The NOT NULL constraint can be specified only at the column level, not the table level
• UNIQUE, PRIMARY KEY, FOREIGN KEY and CHECK constraints can be defined at either the column or table level
• If the word CONSTRAINT is used in a CREATE TABLE statement, you must give the constraint a name
Defining NOT NULL and UNIQUE Constraints

Tell Me / Show Me

Examine the Violations

```
CREATE TABLE clients(
    client_number  NUMBER(6),
    first_name     VARCHAR2(20),
    last_name      VARCHAR2(20),
    phone          VARCHAR2(20) CONSTRAINT phone_email_uk
                        UNIQUE(email,phone),
    email          VARCHAR2(10) CONSTRAINT NOT NULL,
    CONSTRAINT emailclients_email NOT NULL,
    CONSTRAINT clients_client_num_pk
                        PRIMARY KEY (client_number));
```

**COMPOSITE UNIQUE KEY VIOLATION**
Composite keys must be defined at the table level.

**NOT NULL VIOLATION**
NOT NULL constraints can only be defined at the column level.

**NAME VIOLATION**
When using the term, CONSTRAINT, it must be followed by a constraint name.
Tell Me / Show Me

Five Types Of Constraint

There can be five types of constraint within an Oracle database. Each type enforces a different kind of rule.

The types are:

- NOT NULL constraints
- UNIQUE constraints
- PRIMARY KEY constraints
- FOREIGN KEY constraints
- CHECK constraints

In the rest of this lesson you will learn about NOT NULL and UNIQUE constraints. The next lesson will teach you about the other three types.
Tell Me / Show Me

NOT NULL Constraint

A column defined with a NOT NULL constraint requires that for every row entered into the table, there must be a value for that column. For example, if the email column in an employees table was defined as NOT NULL, every employee entered into the table MUST have a value in the email column.

When defining NOT NULL columns, it is customary to use the suffix _nn in the constraint name. For example, the constraint name for the NOT NULL email column in the employees table could be emp_email_nn.
Tell Me / Show Me

UNIQUE Constraint
A UNIQUE constraint requires that every value in a column or set of columns (a composite key) be unique; that is, no two rows of a table can have duplicate values. For example, it may be important for a business to ensure that no two people have the same email address. The email column could be defined using a UNIQUE constraint. The column or set of columns that is defined as UNIQUE is called a unique key. If the combination of two columns must not be the same for any entry, the constraint is said to be a composite unique key. Stating that all combinations of email and last name must be UNIQUE is an example of a composite unique key. The word "key" refers to the columns, not constraint names.
**Tell Me / Show Me**

**Unique Constraint Example**

An example of UNIQUE:

If the email column in the table is defined with a UNIQUE constraint, no other client entry can have an identical email. What if two clients live in the same household and share an email address?

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

```
INSERT INTO copy_d_clients (client_number, first_name, last_name, phone, email)
VALUES (7234, 'Lonny', 'Vigil', 4072220091, 'lbv@lbv.net');

ORA-00001: unique constraint (USWA_SKHS_SQL01_T01.CLIENT_EMAIL_UK) violated
```
Defining UNIQUE Constraints

When defining UNIQUE constraints, it is customary to use the suffix _uk in the constraint name. For example, the constraint name for the UNIQUE email column in the employees table could be emp_email_uk.

To define a composite unique key, you must define the constraint at the table level rather than the column level. An example of a composite unique-key constraint name is:

CONSTRAINT clients_phone_email_uk UNIQUE(email,phone)
Defining NOT NULL and UNIQUE Constraints

Tell Me / Show Me

Composite Unique Key
UNIQUE constraints allow the input of nulls unless the column also has a NOT NULL constraint defined. A null in a column (or in all columns of a composite unique key) always satisfies a UNIQUE constraint because nulls are not considered equal to anything.

To satisfy a constraint that designates a composite unique key, no two rows in the table can have the same combination of values in the key columns.

Also, any row that contains nulls in all key columns automatically satisfies the constraint.

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
<tr>
<td>7234</td>
<td>Lonny</td>
<td>Vigil</td>
<td>4072220091</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

This combination of columns must be **UNIQUE**
When you add a NOT NULL constraint as part of a table creation statement, the Oracle database will create a Check Constraint in the database to enforce a value in the NOT NULL column. This constraint creation can be almost invisible to you when you create your table, Oracle just does it. At the end of your table creation statement you are returned the message: “Table created”, and no details about how many NOT NULL check constraints were created at the same time.
Tell Me / Show Me

Terminology

Key terms used in this lesson include:

Constraint
Column level constraint
NOT NULL constraints
UNIQUE constraints
REFERENCES
Table level constraint
UNIQUE KEY
FOREIGN KEY
PRIMARY KEY
CHECK constraint
Summary

Objectives Summarized
In this lesson you have learned to:

• Define the term "constraint" as it relates to data integrity
• State when it is possible to define a constraint at the column level, and when it is possible at the table level
• State why it is important to give meaningful names to constraints
• State which data integrity rules are enforced by NOT NULL and UNIQUE constraints
• Write a CREATE TABLE statement which includes NOT NULL and UNIQUE constraints at the table and column levels
• Explain how constraints are created at the time of table creation
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
PRIMARY KEY, FOREIGN KEY and CHECK Constraints
Objectives

In this lesson, you will learn to:

- Define and give an example of a PRIMARY KEY, FOREIGN KEY and CHECK constraint
- Explain the purpose of defining PRIMARY KEY, FOREIGN KEY and CHECK constraints
- Demonstrate the creation of constraints at the column level and table level in a CREATE TABLE statement
- Evaluate a business problem requiring the addition of a PRIMARY KEY and FOREIGN KEY constraint and write the code to execute the change
- Query the data dictionary for USER_CONSTRAINTS and interpret the information returned
Why Learn It?

Purpose
As discussed in the last section, constraints are used to prevent invalid data entry into database tables. What would happen if, surreptitiously or just through a careless mistake, your personal unique identification was given to another person? What if tomorrow at school someone else was credited with your classes for graduation or was able to eat lunch using your lunch-card number?

Ensuring data integrity is what constraints are all about. After all, you're unique!
Tell Me / Show Me

PRIMARY KEY Constraints

A PRIMARY KEY constraint is a column or set of columns that uniquely identifies each row in a table. No primary-key value can appear in more than one row in the table. To satisfy a PRIMARY KEY constraint, both of the following conditions must be true:

- No column that is part of the primary key can contain a null.
- A table can have only one primary key.
PRIMARY KEY, FOREIGN KEY and CHECK Constraints

Tell Me / Show Me

PRIMARY KEY Constraints (continued)

PRIMARY KEY constraints can be defined at the column or the table level. However, if a composite PRIMARY KEY is created, it must be defined at the table level.

When defining PRIMARY KEY columns, it is a good practice to use the suffix _pk in the constraint name. For example, the constraint name for the PRIMARY KEY column named id in the DJ on Demand d_events table could be d_events_id_pk.
PRIMARY KEY Constraints (continued)
In a CREATE TABLE statement, the column-level PRIMARY KEY constraint syntax is stated:

```sql
CREATE TABLE clients
(client_number NUMBER(4) CONSTRAINT client_client_num_pk PRIMARY KEY,
first_name VARCHAR2(14),
last_name VARCHAR2(13));
```

Note that the column-level simply refers to the area in the CREATE TABLE statement where the columns are defined. The table level refers to the last lines in the statement below where the individual columns are defined.
Tell Me / Show Me

PRIMARY KEY Constraints (continued)
To define a composite PRIMARY KEY, you must define the constraint at the table level rather than the column level. An example of a composite unique-key constraint name is:

CONSTRAINT id_venue_id_pk PRIMARY KEY (id, venue_id)
FOREIGN KEY (REFERENTIAL INTEGRITY) Constraints

FOREIGN KEY constraints are also called "referential integrity" constraints.

CREATE TABLE clients
(client_number NUMBER(4) CONSTRAINT client_client_num_pk PRIMARY KEY, first_name VARCHAR2(14),
last_name VARCHAR2(13), department_id VARCHAR2(4,0),
CONSTRAINT clients_dept_id_fk FOREIGN KEY(department_id) REFERENCES departments(department_id));

These constraints designate a column or combination of columns as a foreign key. It establishes a relationship between a primary key or unique key in the same table or a different table with the foreign key.
Tell Me / Show Me

Stating a Foreign Key
To state a FOREIGN KEY constraints use statements such as:

“The child table column named __________ with a data type of __________ has a CONSTRAINT named __________ which references its parent table called __________ which has a column called __________.”

To state a table-level FOREIGN KEY constraint use statements such as:

“There is a table-level CONSTRAINT named __________ which is a FOREIGN KEY (in the __________ table); it REFERENCES the parent __________ table (which has a column named __________).”
Tell Me / Show Me

Viewing a Foreign Key

The table containing the foreign key is called the "child" table and the table containing the referenced key is called the "parent" table. In the tables shown, D_CLIENTS primary-key client_number also appears in D_EVENTS as a foreign-key column.

D_CLIENTS - Parents

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

D_EVENTS - Child

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>EVENT_DATE</th>
<th>DESCRIPTION</th>
<th>COST</th>
<th>VENUE_ID</th>
<th>PACKAGE_CODE</th>
<th>THEME_CODE</th>
<th>CLIENT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Party for 200, red, white, blue motif</td>
<td>8000</td>
<td>100</td>
<td>112</td>
<td>200</td>
<td>5922</td>
</tr>
<tr>
<td>105</td>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Black tie at Four Seasons Hotel</td>
<td>10000</td>
<td>220</td>
<td>200</td>
<td>200</td>
<td>6133</td>
</tr>
</tbody>
</table>
**Tell Me / Show Me**

**Referential-integrity Constraint**
To satisfy a referential-integrity constraint, a foreign-key value must match an existing value in the parent table or be NULL. In the example, note that a primary-key value can exist without a corresponding foreign-key value; however, a foreign-key must have a corresponding primary key.

**D_CLIENTS - Parents**

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

**D_EVENTS - Child**

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>EVENT_DATE</th>
<th>DESCRIPTION</th>
<th>COST</th>
<th>VENUE_ID</th>
<th>PACKAGE_CODE</th>
<th>THEME_CODE</th>
<th>CLIENT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Peters</td>
<td>14-MAY-04</td>
<td>Party for 200, red, white, blue motif</td>
<td>8000</td>
<td>100</td>
<td>112</td>
<td>200</td>
<td>5922</td>
</tr>
<tr>
<td>105</td>
<td>Vigil</td>
<td>28-APR-04</td>
<td>Black tie at Four Seasons Hotel</td>
<td>10000</td>
<td>220</td>
<td>200</td>
<td>200</td>
<td>6133</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Referential-Integrity Constraint Rule

The rule is: before you define a referential-integrity constraint in the child table, the referenced UNIQUE or PRIMARY KEY constraint on the parent table must already be defined. In other words, you must first have a parent primary key defined before you can create a foreign key in a child table.

D_CLIENTS - Parents

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

D_EVENTS - Child

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>EVENT_ DATE</th>
<th>DESCRIPTION</th>
<th>COST</th>
<th>VENUE_ID</th>
<th>PACKAGE_CODE</th>
<th>THEME_CODE</th>
<th>CLIENT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Party for 200, red, white, blue motif</td>
<td>8000</td>
<td>100</td>
<td>112</td>
<td>200</td>
<td>5922</td>
</tr>
<tr>
<td>105</td>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Black tie at Four Seasons Hotel</td>
<td>10000</td>
<td>220</td>
<td>200</td>
<td>200</td>
<td>6133</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

FOREIGN KEY Constraint
To define a FOREIGN KEY constraint, it is good practice to use the suffix _fk in the constraint name.

For example, the constraint name for the FOREIGN KEY column song_id in the DJ on Demand table named d_track_listings could be named d_track_list_song_id_fk.
FOREIGN KEY Constraint Syntax
The syntax for defining a FOREIGN KEY constraint requires a reference to the table and column in the parent table. A FOREIGN KEY constraint in a CREATE TABLE statement can be defined as follows.

**Column-level syntax:**

```sql
song_id NUMBER(5) CONSTRAINT d_track_list_song_id_fk REFERENCES d_songs(id)
```

**Table-level syntax:**

```sql
CONSTRAINT d_track_list_song_id_fk FOREIGN KEY (song_id)
REFERENCES d_songs(id)
```
**Tell Me / Show Me**

**ON DELETE CASCADE - Maintaining Referential Integrity**

Using the ON DELETE CASCADE option when defining a foreign key enables the dependent rows in the child table to be deleted when a row in the parent table is deleted. If the foreign key does not have an ON DELETE CASCADE option, referenced rows in the parent table cannot be deleted. In other words, the child table FOREIGN KEY constraint includes the ON DELETE CASCADE permission allowing its parent to delete rows that it refers to.

**D_CLIENTS - Parents**

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

**D_EVENTS - Child**

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>EVENT_DATE</th>
<th>DESCRIPTION</th>
<th>COST</th>
<th>VENUE_ID</th>
<th>PACKAGE_CODE</th>
<th>THEME_CODE</th>
<th>CLIENT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Peters</td>
<td>14-MAY-04</td>
<td>Party for 200, red, white, blue motif</td>
<td>8000</td>
<td>100</td>
<td>112</td>
<td>200</td>
<td>5922</td>
</tr>
<tr>
<td>105</td>
<td>Vigil</td>
<td>28-APR-04</td>
<td>Black tie at Four Seasons Hotel</td>
<td>10000</td>
<td>220</td>
<td>200</td>
<td>200</td>
<td>6133</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

ON DELETE CASCADE

If the song_id column in D_TRACK_LISTINGS was created with the ON DELETE CASCADE option specified, the DELETE statement issued on the D_SONGS table will execute. If the ON DELETE CASCADE option was not specified when the song_id column in D_TRACK_LISTINGS was created, the attempt to delete song_id = 47 will fail.
Tell Me / Show Me

Column-level ON DELETE CASCADE Syntax
song_id NUMBER(5) CONSTRAINT d_track_list_song_id_fk REFERENCES d_songs(id) ON DELETE CASCADE

Table-level ON DELETE CASCADE syntax:
CONSTRAINT d_track_list_song_id_fk FOREIGN KEY (song_id) REFERENCES d_songs(id) ON DELETE CASCADE

D_TRACK_LISTINGS

DELETE from D_SONGS WHERE song_id = 47

<table>
<thead>
<tr>
<th>SONG_ID</th>
<th>CD_NUMBER</th>
<th>TRACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>92</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>93</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>91</td>
<td>2</td>
</tr>
<tr>
<td>48</td>
<td>95</td>
<td>5</td>
</tr>
<tr>
<td>49</td>
<td>91</td>
<td>3</td>
</tr>
<tr>
<td>50</td>
<td>93</td>
<td>4</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

ON DELETE SET NULL

Rather than having the rows in the child table deleted when using an ON DELETE CASCADE option, the child rows can be filled with null values using the ON DELETE SET NULL option.

When do you choose whether to delete a row or simply set the values to null? An example might be when the parent table value is being changed to a new number such as converting inventory numbers to bar-code numbers. You would not want to delete the rows in the child table. When the new bar-code numbers are entered into the parent table, they would then be able to be inserted into the child table without having to totally re-create each child table row.
Tell Me / Show Me

CHECK Constraints

The CHECK constraint explicitly defines a condition that must be met. To satisfy the constraint, each row in the table must make the condition either True or unknown (due to a null). The condition of a CHECK constraint can refer to any column in the specified table, but not to columns of other tables.
CHECK Constraint Example

CREATE d_cds (cd_number NUMBER CONSTRAINT d_cds_cd_num_range
    CHECK (cd_number BETWEEN 10 AND 999) ,
    year NUMBER(4) CONSTRAINT d_cds_year_min
    CHECK (year > 1996) ,
    producer VARCHAR2(10) CONSTRAINT d_cds_prod_list
    CHECK (producer IN ('Old Town Records','The Music Man',
    'Middle Earth Records','R&B Inc','Tunes Are US')) ;

What is each constraint limiting? The cd_numbers must be
between 10 and 999; year must be greater than 1996; the producer
must be in the list shown.
Tell Me / Show Me

CHECK Constraint Conditions

- A CHECK constraint must only be on the row where the constraint is defined.
- A CHECK constraint cannot be used in queries that refer to values in other rows.
- The CHECK constraint cannot contain calls to the functions SYSDATE, UID, USER, or USERENV. The statement CHECK(SYSDATE > '05-MAY-99') is not allowed.
- The CHECK constraint cannot use the pseudocolumns CURRVAL, NEXTVAL, LEVEL, or ROWNUM. The statement CHECK(NEXTVAL > 0) is not allowed.
- A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.
Tell Me / Show Me

CHECK Constraint Syntax
CHECK constraints can be defined at the column level or the table level.

The syntax to define a CHECK constraint is:

**Column-level syntax:**
```
salary NUMBER(8,2) CONSTRAINT f_staffs_min_salary CHECK (salary > 0)
```

**Table-level syntax:**
```
CONSTRAINT f_staffs_min_salary CHECK (salary > 0)
```
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

- PRIMARY KEY constraint
- FOREIGN KEY constraint
- ON DELETE CASCADE
- ON DELETE SET NULL
- CHECK constraint
- NOT NULL
Summary

Objectives Summarized
In this lesson you have learned to:

• Provide an example of a PRIMARY KEY, FOREIGN KEY and CHECK constraint
• Explain the purpose of defining PRIMARY KEY, FOREIGN KEY and CHECK constraints
• Demonstrate the creation of constraints at the column level and table level in a CREATE TABLE statement
• Evaluate a business problem requiring the addition of a PRIMARY KEY and FOREIGN KEY constraint and writing the code to execute the change
• Query the data dictionary for USER_CONSTRAINTS and interpret the information returned
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Managing Constraints
What Will I Learn?

Objectives Summarized
In this lesson, you will learn to:

• List four different functions that the ALTER statement can perform on constraints
• Write ALTER TABLE statements to add, drop, disable and enable constraints
• Name a business function that would require a DBA to drop, enable and/or disable a constraint or use the CASCADE syntax
• Query the data dictionary for USER_CONSTRAINTS and interpret the information returned
Why Learn It?

Purpose
Would it make any difference if a new student ID number was entered into the school's database when no actual student enrolled?

Is it likely that a credit-card company would issue the same credit-card number to more than one account or that a business would hire an employee for a department that didn't exist?

What do you predict would happen if a business could not trust the reliability of the information in its database?
Why Learn It?

Purpose (continued)
A database system needs to be able to enforce business rules and at the same time prevent adding, modifying, or deleting data that violates the referential integrity of the database.

In this section, you will learn how to make changes to table constraints so that referential integrity and, in turn, database reliability are maintained when data needs to be changed.
Tell Me / Show Me

Managing Constraints

The ALTER TABLE statement is used to make changes to constraints in existing tables. These changes can include adding or dropping constraints, enabling or disabling constraints, and adding a NOT NULL constraint to a column. The guidelines for making changes to constraints are:

- You can add, drop, enable or disable a constraint, but you cannot modify its structure.
- You can add a NOT NULL constraint to an existing column by using the MODIFY clause of the ALTER TABLE statement. MODIFY is used because NOT NULL is a column-level change.
- You can define a NOT NULL constraint only if the table is empty or if the column has a value for every row.
Tell Me / Show Me

The ALTER statement requires:

- name of the table
- name of the constraint
- type of constraint
- name of the column affected by the constraint

In the code example shown below, using the DJs on Demand database, the primary-key constraint could have been added after the D_CLIENTS table was originally created. In this case, the primary-key constraint is added to the D_CLIENTS table.

```
ALTER TABLE d_clients
ADD CONSTRAINT clients_client_num_pk PRIMARY KEY(client_number)
```
Tell Me / Show Me

Adding Constraints
To add a constraint to an existing table, use the following SQL syntax:

```
ALTER TABLE table_name
ADD [CONSTRAINT constraint_name] type of constraint (column_name);
```
Adding Constraints (continued)
If the constraint is a FOREIGN KEY constraint, the REFERENCES keyword must be included in the statement:

```
ADD CONSTRAINT constraint_name FOREIGN KEY(column_name) REFERENCES tablename(column_name);
```
Adding Constraints Example

Consider the DJs on Demand database. The primary key from the D_CLIENTS table is entered in the D_EVENTS table as a foreign key. The following example demonstrates the syntax to add this foreign key to the D_EVENTS table:

```
ALTER TABLE d_events
ADD CONSTRAINT d_events_client_num_fk FOREIGN KEY (client_number) REFERENCES d_clients(client_number) ON DELETE CASCADE;
```

### D_CLIENTS

<table>
<thead>
<tr>
<th>CLIENT_NUMBER</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>PHONE</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5922</td>
<td>Hiram</td>
<td>Peters</td>
<td>3715832249</td>
<td><a href="mailto:hpeters@yahoo.com">hpeters@yahoo.com</a></td>
</tr>
<tr>
<td>5857</td>
<td>Serena</td>
<td>Jones</td>
<td>7035335900</td>
<td><a href="mailto:serena.jones@jones.com">serena.jones@jones.com</a></td>
</tr>
<tr>
<td>6133</td>
<td>Lauren</td>
<td>Vigil</td>
<td>4072220090</td>
<td><a href="mailto:lbv@lbv.net">lbv@lbv.net</a></td>
</tr>
</tbody>
</table>

### D_EVENTS

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>EVENT_DATE</th>
<th>DESCRIPTION</th>
<th>COST</th>
<th>VENUE_ID</th>
<th>PACKAGE_CODE</th>
<th>THEME_CODE</th>
<th>CLIENT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Party for 200, red, white, blue motif</td>
<td>8000</td>
<td>100</td>
<td>112</td>
<td>200</td>
<td>5922</td>
</tr>
<tr>
<td>105</td>
<td>Vigil Wedding</td>
<td>28-APR-04</td>
<td>Black tie at Four Seasons Hotel</td>
<td>10000</td>
<td>220</td>
<td>200</td>
<td>200</td>
<td>6133</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Adding Constraints Conditions
If the constraint is a NOT NULL constraint, the ALTER TABLE statement uses MODIFY in place of ADD. NOT NULL constraints can be added only if the table is empty or if the column has a value for every row:

```
ALTER TABLE table_name
MODIFY (column_name CONSTRAINT constraint_name NOT NULL);
```

```
ALTER TABLE d_clients
MODIFY (email CONSTRAINT d_clients_email_nn NOT NULL);
```
Tell Me / Show Me

Why Enable and Disable Constraints?
To enforce the rules defined by integrity constraints, the constraints should always be enabled. In certain situations, however, it is desirable to temporarily disable the integrity constraints of a table temporarily for performance reasons, such as:

- When loading large amounts of data into a table
- When performing batch operations that make massive changes to a table (such as changing everyone's employee number by adding 1,000 to the existing number)
Tell Me / Show Me

Dropping Constraints
To drop a constraint, you need to know the name of the constraint. If you do not know it, you can find the constraint name from the USER_CONSTRAINTS and USER_CONS_COLUMNS in the data dictionary.

The CASCADE option of the DROP clause causes any dependent constraints also to be dropped. Note that when you drop an integrity constraint, that constraint is no longer enforced by the Oracle Server and is no longer available in the data dictionary.

No rows or any data in any of the affected tables are deleted when you drop a constraint

```
ALTER TABLE table_name
DROP CONSTRAINT name [CASCADE];
```
Disabling Constraints

By default, whenever an integrity constraint is defined in a CREATE or ALTER TABLE statement, the constraint is automatically enabled (enforced) by Oracle unless it is specifically created in a disabled state using the DISABLE clause.

You can disable a constraint without dropping it or re-creating it by using the ALTER TABLE option DISABLE. DISABLE allows incoming data, regardless of whether it conforms to the constraint. This function allows data to be added to a child table, without having corresponding values in the parent table.

This is a little bit like simply switching the constraint off.
Tell Me / Show Me

Using the DISABLE Clause
You can use the DISABLE clause in both the ALTER TABLE statement and the CREATE TABLE statement.

```
ALTER TABLE d_clients
DISABLE CONSTRAINT clients_client_num_pk

CREATE TABLE d_clients
(client_number NUMBER(5)
PRIMARY KEY DISABLE);
```

Disabling a unique or primary-key constraint removes the unique index.
Tell Me / Show Me

Using the CASCADE Clause
The CASCADE clause disables dependent integrity constraints. If the constraint is later enabled, the dependent constraints are not automatically enabled.

ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name [CASCADE];

ALTER TABLE d_clients
DISABLE CONSTRAINT clients_client_num_pk CASCADE;
Enabling Constraints
To activate an integrity constraint currently disabled, use the ENABLE clause in the ALTER TABLE statement. ENABLE ensures that all incoming data conforms to the constraint.

ALTER TABLE table_name
ENABLE CONSTRAINT constraint_name;

ALTER TABLE d_clients
ENABLE CONSTRAINT clients_client_num_pk;

You can use the ENABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.
Tell Me / Show Me

Enabling Constraint Considerations

If you enable a constraint, that constraint applies to all the data in the table.

All the data in the table must fit the constraint. If you enable a UNIQUE KEY or PRIMARY KEY constraint, a UNIQUE or PRIMARY KEY index is created automatically.

Enabling a PRIMARY KEY constraint that was disabled with the CASCADE option does not enable any foreign keys that are dependent on the primary key.

This is like switching the constraint back on, after you switched it off.
Tell Me / Show Me

Cascading Constraints

Cascading referential-integrity constraints allow you to define the actions the database server takes when a user attempts to delete or update a key to which existing foreign keys point. The CASCADE CONSTRAINTS clause is used along with the DROP COLUMN clause. It drops all referential-integrity constraints that refer to the primary and unique keys defined on the dropped columns. It also drops all multicolumn constraints defined on the dropped columns. If an ALTER TABLE statement does not include the CASCADE CONSTRAINTS option, any attempt to drop a primary key or multicolumn constraint will fail. Remember, you can’t delete a parent value if child values exist in other tables.

```
ALTER TABLE table_name
DROP(column name(s)) CASCADE CONSTRAINTS;
```
Tell Me / Show Me

When CASCADE is Not Required

If all columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to column PK, it is valid to submit the following statement without the CASCADE CONSTRAINTS clause:

```
ALTER TABLE tablename DROP
(pk_column_name(s));
```

However, if any constraint is referenced by columns from other tables or remaining columns in the target table, you must specify CASCADE CONSTRAINTS to avoid an error.
Tell Me / Show Me

Viewing Constraints

After creating a table, you can confirm its existence by issuing a DESCRIBE command. The only constraint that you can verify using DESCRIBE is the NOT NULL constraint.

The NOT NULL constraint will also appear in the data dictionary as a CHECK constraint.
Viewing Constraints (continued)
To view all constraints on your table, query the USER_CONSTRAINTS table.

```
SELECT constraint_name, constraint_type
FROM user_constraints
WHERE table_name = 'TABLE_NAME';
```

In constraint types listed in the data dictionary, C stands for CHECK, P for PRIMARY KEY, R for REFERENTIAL INTEGRITY, and U for UNIQUE.

```
<table>
<thead>
<tr>
<th>CONSTRAINT_NAME</th>
<th>CONSTRAINT_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANIMALS_ADMIT_NN</td>
<td>C Check Constraint</td>
</tr>
<tr>
<td>ANIMALS_VACC_NN</td>
<td>C Check Constraint</td>
</tr>
<tr>
<td>ANIMAL_ID_PK</td>
<td>P Primary key</td>
</tr>
<tr>
<td>LIC_TAG_NUM_UK</td>
<td>U Unique key</td>
</tr>
</tbody>
</table>
```
Tell Me / Show Me

Viewing Constraints With The Data Dictionary
You can browse the Data Dictionary using Oracle Application Express. Below is a summary of the steps used to browse the Data Dictionary.

1. In Application Express, choose Utilities.
2. Choose Object Reports.
3. Under All Object Reports, choose Data Dictionary.
4. In the Search box enter USER_CONSTRAINTS and click GO.
5. Click the USER_CONSTRAINTS link.
6. Select the information you want returned in the Dictionary Query By Example (QBE) Form. Select check boxes 1, 2, 3, and 4.
7. Click the Query button.
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

ALTER TABLE
DISABLE CONSTRAINT
CASCADE clause
ENABLE CONSTRAINT
DROP COLUMN
CASCADE RESTRAINTS clause
DROP CONSTRAINT
Summary

Objectives Summarized
In this lesson you have learned to:

• List four different functions that the ALTER statement can perform on constraints
• Write ALTER TABLE statements to add, drop, disable and enable constraints
• Name a business function that would require a DBA to drop, enable and/or disable a constraint or use the CASCADE syntax
• Query the data dictionary for USER_CONSTRAINTS and interpret the information returned
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Creating Views
What Will I Learn?

Objectives

In this lesson, you will learn to:

• List three uses for views from the standpoint of a database administrator

• Explain, from a business perspective, why it is important to be able to create and use logical subsets of data derived from one or more tables

• Create a view with and without column aliases in the subquery using a single base table

• Create a complex view that contains group functions to display values from two tables

• Retrieve data from a view
Why Learn It?

Purpose
Take a minute to look back at what you've learned so far as an Oracle Academy student. How easy would it be to explain what you know to someone who hadn't taken this class? You should pat yourself on the back!

The level of knowledge you have acquired is understood by only a select few. Now, imagine yourself as the Database Administrator of a business. What do you do when a manager asks you to make it possible for him/her to be able to retrieve and input data using the company's database? "Don't make it too complicated; I just want to be able to prepare reports about all our operations."
Why Learn It?

Purpose (continued)
Should these employees have access to all of the company's data?

How will they execute commands that require join conditions?

Is it wise to allow data input from anyone?

These are questions that you, as DBA, need to know how to answer. In this section, you will learn how to create "views" -- virtual representations of tables customized to meet specific user requirements.
Tell Me / Show Me

View

A view, like a table, is a database object. However, views are not “real” tables. They are logical representations of existing tables or of another view. Views contain no data of their own. They function as a window through which data from tables can be viewed or changed. The tables on which a view is based are called "base" tables. The view is a query stored as a SELECT statement in the data dictionary.

CREATE VIEW view_employees
AS SELECT first_name, last_name, email
FROM employees
WHERE employee_id BETWEEN 100 and 124;

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
</tr>
<tr>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOCHHAR</td>
</tr>
<tr>
<td>Lex</td>
<td>De Haan</td>
<td>LDEHAAN</td>
</tr>
<tr>
<td>Alexander</td>
<td>Hunold</td>
<td>AHUNOLD</td>
</tr>
<tr>
<td>Bruce</td>
<td>Ernst</td>
<td>BERNST</td>
</tr>
<tr>
<td>Diana</td>
<td>Lorentz</td>
<td>DLORENTZ</td>
</tr>
<tr>
<td>Kevin</td>
<td>Mourgos</td>
<td>KMOURGOS</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Why Use Views?

• Views restrict access to base table data because the view can display selective columns from the table.

• Views can be used to reduce the complexity of executing queries based on more complicated SELECT statements. For example, the creator of the view can construct join statements that retrieve data from multiple tables. The user of the view neither sees the underlying code nor how to create it. The user, through the view, interacts with the database using simple queries.

• Views can be used to retrieve data from several tables, providing data independence for users. Users can view the same data in different ways.

• Views provide groups of users with access to data according to their particular permissions or criteria.
Creating A View
To create a view, embed a subquery within the CREATE VIEW statement. The syntax of a view statement is as follows:

CREATE [OR REPLACE] [FORCE| NOFORCE] VIEW view [(alias [, alias]...)] AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];

Example
CREATE OR REPLACE VIEW view_of_animals
AS SELECT animal_name...;
Creating Views

Tell Me / Show Me

Creating A View (continued)

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

<table>
<thead>
<tr>
<th>OR REPLACE</th>
<th>re-creates the view if it already exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORCE</td>
<td>creates the view regardless of whether or not the base tables exists</td>
</tr>
<tr>
<td>NOFORCE</td>
<td>creates the view only if the base table exists (default)</td>
</tr>
<tr>
<td>view_name</td>
<td>name of view</td>
</tr>
<tr>
<td>alias</td>
<td>specifies a name for each expression selected by the view’s query</td>
</tr>
<tr>
<td>subquery</td>
<td>is a complete SELECT statement (you can use aliases for the columns in the SELECT list). The subquery can contain complex SELECT syntax</td>
</tr>
<tr>
<td>WITH CHECK OPTION</td>
<td>specifies that only rows accessible to the view can be inserted or updated</td>
</tr>
<tr>
<td>constraint</td>
<td>is the name assigned to the CHECK OPTION constraint</td>
</tr>
<tr>
<td>WITH READ ONLY</td>
<td>ensures that no DML operations can be performed on this view</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Guidelines for Creating A View

• The subquery that defines the view can contain complex SELECT syntax.
• The subquery that defines the view cannot contain an ORDER BY clause. The ORDER BY clause is specified when you retrieve data from the view.
• You can use the OR REPLACE option to change the definition of the view without having to drop it or regrant object privileges previously granted on it.
• Aliases can be used for the column names in the subquery.
Tell Me / Show Me

CREATE VIEW Features

There are two classifications for views: simple and complex. The table summarizes the features of each view.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Simple Views</th>
<th>Complex Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tables used to derive data</td>
<td>One</td>
<td>One or more</td>
</tr>
<tr>
<td>Can contain functions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can contain groups of data</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can perform DML operations</td>
<td>Yes</td>
<td>Not always</td>
</tr>
<tr>
<td>(INSERT, UPDATE, DELETE) through a view</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Simple View
The view shown below is an example of a simple view. The subquery derives data from only one table and it does not contain a join function or any group functions. Because it is a simple view, INSERT, UPDATE, DELETE and MERGE operations affecting the base table could possibly be performed through the view.

CREATE VIEW view_copy_d_cds
AS SELECT cd_number, title, producer, year
FROM d_cds;
Simple View (continued)
Column names in the SELECT statement can have aliases as shown below. Note that aliases can also be listed after the CREATE VIEW statement and before the SELECT subquery.

CREATE VIEW view_copy_d_cds  
AS SELECT cd_number AS "Number", title AS "Title", year AS "Year_Recorded"
FROM d_cds;

CREATE VIEW view_copy_d_cds(Number, Title, Year_Recorded)  
AS SELECT cd_number, title, year
FROM d_cds;
Simple View (continued)

It is possible to create a view regardless of whether or not the base tables exist. Adding the word FORCE to the CREATE VIEW statement creates the view.

As a DBA, this option could be useful during the development of a database, especially if you are waiting for the necessary privileges to the referenced object to be granted shortly. The FORCE option will create the view despite it being invalid.

The NOFORCE option is the default when creating a view.
Complex View

Complex views are views that can contain group functions and joins. The following example creates a view that derives data from two tables.

```
CREATE VIEW view_dj_on_demand (LAST_NAME, TELEPHONE, EVENT, DATE_HELD)
AS SELECT c.last_name, c.phone, e.name,
       TO_CHAR(e.event_date, 'Month dd, YYYY')
FROM d_clients c, d_events e
WHERE c.client_number = e.client_number;
```
Complex View (continued)

Group functions can also be added to complex-view statements.

CREATE VIEW view_dj_cds (TITLE, SONG, MIN_YEAR, MAX_YEAR)
AS SELECT c.title, t.song_id, MIN(c.year), MAX(c.year)
FROM d_cds c, d_track_listings t
WHERE c.cd_number = t.cd_number
GROUP BY c.cd_number, c.title, t.song_id;
Tell Me / Show Me

Modifying A View

To modify an existing view without having to re-create it, use the OR REPLACE option in the CREATE VIEW statement. The old view is replaced by the new version. For example:

```sql
CREATE OR REPLACE VIEW view_copy_d_cds
AS SELECT cd_number, title, producer, year
FROM d_cds;
```
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

- Alias
- Complex review
- CREATE VIEW
- FORCE
- NOFORCE
- REPLACE
- Simple view
- Subquery
- View
- VIEW_NAME
Summary

Objectives Summarized
In this lesson you have learned to:

• List three uses for views from the standpoint of a database administrator
• Explain, from a business perspective, why it is important to be able to create and use logical subsets of data derived from one or more tables
• Create a view with and without column aliases in the subquery using a single base table
• Create a complex view that contains group functions to display values from two tables
• Retrieve data from a view
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
DML Operations and Views
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Write and execute a query that performs DML operations on a simple view
• Name the conditions that restrict your ability to modify a view using DML operations
• Write and execute a query using the WITH CHECK OPTION clause
• Explain the use of WITH CHECK OPTION as it applies to integrity constraints and data validation
• Apply the WITH READ ONLY option to a view to restrict DML operations
Why Learn It?

Purpose
As you learned in the last lesson, views simplify queries that require data collected from multiple tables in the database.

However, views also allow users to make changes to the underlying tables.

As the DBA and the person whose job it is to maintain the integrity of the database, you will want to put constraints on these views of the data.

In this lesson, you will learn how to allow data access and at the same time ensure data security.
Why Learn It?

Purpose (continued)
Have you ever wondered what your classroom will look like in 20 years?

A new generation of students will be sitting in your chairs. Will they be staring at huge monitors or sitting comfortably using their own hand-held personal computer?

Maybe school will be something we do from home. Interesting thought, isn't it?

In this lesson, you will have an opportunity to look at what's next in computing.

You'll be part of shaping the future.
Tell Me / Show Me

DML Statements And Views
The DML operations INSERT, UPDATE and DELETE can be performed on simple views. These operations can be used to change the data in the underlying base tables. If you create a view that allows users to view restricted information using the WHERE clause, users can still perform DML operations on all of the view's columns.
Tell Me / Show Me

DML Statements And Views (Continued)

For example, the view shown at right was created for the managers of department 50 from the Oracle database. The intent of this view is to allow managers of department 50 to see information about their employees.

CREATE VIEW view_dept50 AS
SELECT department_id, employee_id, first_name, last_name, salary
FROM employees
WHERE department_id = 50;

SQL Statement (All DDL statements are auto committed.)
select * from view_dept50;

SQL Query Results

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>124</td>
<td>Kevin</td>
<td>Mourgos</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>141</td>
<td>Trenna</td>
<td>Rajs</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>142</td>
<td>Curtis</td>
<td>Davies</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>143</td>
<td>Randall</td>
<td>Matos</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>144</td>
<td>Peter</td>
<td>Vargas</td>
<td>2500</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Controlling Views
Using the view as stated, it is possible to INSERT, UPDATE, and DELETE information for all departments.

This may not be what the DBA intended when the view was created.

To control data access, two options can be added to the CREATE VIEW statement:

- WITH CHECK OPTION
- WITH READ ONLY
Tell Me / Show Me

Views With CHECK Option

The WITH CHECK OPTION ensures that DML operations performed on the view stay within the domain of the view. Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint. Notice in the example below that the WITH CHECK OPTION CONSTRAINT was given the name view_dept50_check.

CREATE OR REPLACE VIEW view_dept50 AS
SELECT department_id, employee_id, first_name, last_name, salary
FROM employees
WHERE department_id = 50
WITH CHECK OPTION CONSTRAINT view_dept50_check;
Tell Me / Show Me

Views With READ ONLY

The WITH READ ONLY option ensures that no DML operations occur through the view. Any attempt to execute an INSERT, UPDATE or DELETE statement will result in an Oracle server error.

CREATE OR REPLACE VIEW view_dept50 AS
SELECT department_id, employee_id, first_name, last_name, salary
FROM employees
WHERE department_id = 50;
WITH READ ONLY;
Tell Me / Show Me

DML Restrictions

Simple views and complex views differ in their ability to allow DML operations through a view.

For simple views, DML operations can be performed through the view.

For complex views, DML operations are not always allowed.

The following 3 rules must be considered when performing DML operations on views.

```
SELECT rownum, first_name
FROM employees
WHERE employee_id
BETWEEN 100 AND 105;
```

<table>
<thead>
<tr>
<th>ROWNUM</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Steven</td>
</tr>
<tr>
<td>2</td>
<td>Neena</td>
</tr>
<tr>
<td>3</td>
<td>Lex</td>
</tr>
<tr>
<td>4</td>
<td>Alexander</td>
</tr>
<tr>
<td>5</td>
<td>Bruce</td>
</tr>
</tbody>
</table>
DML Restrictions (continued)

(1) You cannot remove a row from an underlying base table if the view contains any of the following:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM Keyword

```sql
SELECT rownum, first_name
FROM employees
WHERE employee_id
BETWEEN 100 AND 105;
```

<table>
<thead>
<tr>
<th>ROWNUM</th>
<th>FIRST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Steven</td>
</tr>
<tr>
<td>2</td>
<td>Neena</td>
</tr>
<tr>
<td>3</td>
<td>Lex</td>
</tr>
<tr>
<td>4</td>
<td>Alexander</td>
</tr>
<tr>
<td>5</td>
<td>Bruce</td>
</tr>
</tbody>
</table>
(2) You cannot modify data through a view if the view contains:
   - Group functions
   - A GROUP BY clause
   - The DISTINCT keyword
   - The pseudocolumn ROWNUM keyword
   - Columns defined by expressions
DML Restrictions (continued)

(3) You cannot add data through a view if the view

- includes group functions
- includes a GROUP BY clause
- includes the DISTINCT keyword
- includes the pseudocolumn ROWNUM keyword
- includes columns defined by expressions
- does not include NOT NULL columns in the base tables
Tell Me / Show Me

What’s Next in Computing?
Moore's Law—which states that the number of transistors on a given chip can be doubled every two years--has been the guiding principle of progress in electronics and computing since Moore first formulated the famous prediction in 1965.

What will technology "look like" in 20 years, in a hundred years?

Many people are wondering the same thing and working to turn their dreams into reality.

Let’s look into the future of computing and technology. It's your future and you will be a part of it.
Tell Me / Show Me

Future Trends To Consider

Wireless technologies -- when can we pull the plug?
How big is big? What technologies are being developed to store large quantities of information?
How much is too much? What are the trends in storing personal data and what are the issues being addressed related to personal privacy?!
What is data mining? How can businesses target product advertising gleaned from data stored about your buying habits or Internet browsing preferences?
How can we make computers know how we see and feel?
What technologies are being developed to protect copyrighted material?
How small is small? What are the limits to miniaturizing computer technologies? Can a phone, computer, and camera be integrated into a wrist watch?
Tell Me / Show Me

Terminology
Key term used in this lesson include:

- ROWNUM
- WITH CHECK OPTION
- WITH READ ONLY
Summary

Objectives Summarized
In this lesson you have learned to:

• Write and execute a query that performs DML operations on a simple view
• Name the conditions that restrict your ability to modify a view using DML operations
• Write and execute a query using the WITH CHECK OPTION clause
• Explain the use of WITH CHECK OPTION as it applies to integrity constraints and data validation
• Apply the WITH READ ONLY option to a view to restrict DML operations
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Managing Views
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Create and execute a SQL statement that removes a view
• Create and execute a query to create an inline view
• Create and execute a top-n-analysis query
Why Learn It?

Purpose

Learning to create and replace views wouldn't be complete unless you also knew how to remove them.

Views are created for specific purposes. When the view is no longer needed or needs to be modified, there has to be a way to make the necessary changes.

If an employee who had access to financial information leaves the company, you probably don’t want his/her view to remain accessible.

In this lesson, you will learn how to delete a view, create an inline view and construct a \texttt{SELECT} statement to produce a sorted list of data.
Tell Me / Show Me

Deleting A View

Because a view contains no data of its own, removing it does not affect the data in the underlying tables. If the view was used to INSERT, UPDATE or DELETE data in the past, these changes to the base tables remain.

Deleting a view simply removes the view definition from the database. Remember, views are stored as SELECT statements in the data dictionary. Only the creator or users with the DROP ANY VIEW privilege can remove a view. The SQL syntax to remove a view is:

```
DROP VIEW viewname;
```
Tell Me / Show Me

Inline Views

Inline views are also referred to as queries in the FROM clause.

You insert a query in the FROM clause just as if the query was a table name.

Inline views are commonly used to simplify complex queries by removing join operations and condensing several queries into a single query.
Inline Views (continued)
As shown in the example below, the FROM clause contains a SELECT statement that retrieves data much like any SELECT statement. The data returned by the subquery is given an alias (p), which is then used in conjunction with the main query to return selected columns from both query sources.

```
SELECT e.name, e.description, p.maxrange, p.code
FROM d_events e, (SELECT code, max(high_range) maxrange
FROM d_packages
GROUP BY code) p
WHERE e.package_code = p.code
AND e.cost < p.maxrange;
```
Tell Me / Show Me

TOP-N-ANALYSIS

Top-n-analysis is a SQL operation used to rank results. The use of top-n-analysis is useful when you want to retrieve the top-n records, or top 4 records, of a result set returned by a query.

The top-n-analysis query uses an inline subquery to return a result set. You can use ROWNUM in your query to assign a row number to the result set. The main query then uses ROWNUM to order the data and return the top four.

```sql
SELECT ROWNUM as RANK, year, title
FROM (SELECT year, title
      FROM d_cds
      ORDER BY year)
WHERE ROWNUM <= 4;
```
TOP-N-ANALYSIS (continued)

SELECT ROWNUM as RANK, year, title
FROM (SELECT year, title
       FROM d_cds
       ORDER BY year)
WHERE ROWNUM <= 4;

In the example above, the inline subquery first selects the list of years and titles of the DJ on Demand's CDs:

(SELECT year, title FROM d_cds)

Then the inline subquery orders the years from oldest to newest.

(SELECT …… ORDER BY year)

The outer query WHERE clause is used to restrict the number of rows returned and must use a < or <= operator.

WHERE ROWNUM <= 4;
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

INLINE VIEW
DROP VIEW
TOP-N ANALYSIS
Summary

Objectives Summarized
In this lesson you have learned to:

• Create and execute a SQL statement that removes a view
• Create and execute a query to create an inline view
• Create and execute a top-n-analysis query
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Working With Sequences
What Will I Learn?

Objectives
In this lesson, you will learn to:

• List at least three useful characteristics of a sequence
• Write and execute a SQL statement that creates a sequence
• Query the data dictionary using USER_SEQUENCES to confirm a sequence definition
• Apply the rules for using NEXTVAL to generate sequential unique numbers in a table
• List the advantages and disadvantages of caching sequence values
• Name three reasons why gaps can occur in a sequence
Why Learn It?

Purpose
Can you image how tedious it would be to have to enter the names of the 30,000 people who enter the London Marathon into a database, while making sure that no one was given the same identification number?

What if you went to lunch and when you returned, someone else had entered some of the runners’ applications?

How would you know where to start again?

Fortunately, SQL has a process for automatically generating unique numbers that eliminates the worry about the details of duplicate numbers. The numbering process is handled through a database object called a SEQUENCE.
Tell Me / Show Me

The Sequence Object

You already know how to create two kinds of database objects, the TABLE and the VIEW.

A third database object is the SEQUENCE. A SEQUENCE is a shareable object used to automatically generate unique numbers.

Because it is a shareable object, multiple users can access it. Typically, sequences are used to create a primary-key value.

As you’ll recall, primary keys must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it reduces the amount of code you need to write.
Tell Me / Show Me

The Sequence Object (continued)
Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

To create a SEQUENCE:
CREATE SEQUENCE sequence
    [INCREMENT BY n]
    [START WITH n]
    [{MAXVALUE n | NOMAXVALUE}]
    [{MINVALUE n | NOMINVALUE}]
    [{CYCLE | NOCYCLE}]
    [{CACHE n | NOCACHE}];
**Sequence Syntax**

CREATE SEQUENCE  
`sequence`  

[INCREMENT BY `n`]  
[START WITH `n`]  
[{{MAXVALUE `n` | NOMAXVALUE}}]  
[{{MINVALUE `n` | NOMINVALUE}}]  
[{{CYCLE | NOCYCLE}}]  
[{{CACHE `n` | NOCACHE}}];

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>is the name of the sequence generator (object)</td>
</tr>
<tr>
<td>INCREMENT BY <code>n</code></td>
<td>specifies the interval between sequence numbers where <code>n</code> is an integer (If this clause is omitted, the sequence increments by 1.)</td>
</tr>
<tr>
<td>START WITH <code>n</code></td>
<td>specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)</td>
</tr>
<tr>
<td>MAXVALUE <code>n</code></td>
<td>specifies the maximum value the sequence can generate</td>
</tr>
<tr>
<td>NOMAXVALUE</td>
<td>specifies a maximum value of $10^{27}$ for an ascending sequence and -1 for a descending sequence (default)</td>
</tr>
<tr>
<td>MINVALUE <code>n</code></td>
<td>specifies the minimum sequence value</td>
</tr>
<tr>
<td>NOMINVALUE</td>
<td>specifies a minimum value of 1 for an ascending sequence and $-10^{26}$ for a descending sequence (default)</td>
</tr>
<tr>
<td>CYCLE</td>
<td>specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)</td>
</tr>
<tr>
<td>NOMINVALUE</td>
<td>specifies a minimum value of 1 for an ascending sequence and $-10^{26}$ for a descending sequence (default)</td>
</tr>
<tr>
<td>CACHE <code>n</code></td>
<td>specifies how many values the Oracle server pre-allocates and keep in memory. (By default, the Oracle server caches 20 values.) If the system crashes, the values are lost.</td>
</tr>
<tr>
<td>NOCACHE</td>
<td>specifies that the Oracle server does not pre-allocate values.</td>
</tr>
</tbody>
</table>

Tell Me / Show Me

- `sequence` is the name of the sequence generator (object).
- `INCREMENT BY `n` specifies the interval between sequence numbers where `n` is an integer (If this clause is omitted, the sequence increments by 1.).
- `START WITH `n` specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.).
- `MAXVALUE `n` specifies the maximum value the sequence can generate.
- `NOMAXVALUE` specifies a maximum value of $10^{27}$ for an ascending sequence and -1 for a descending sequence (default).
- `MINVALUE `n` specifies the minimum sequence value.
- `NOMINVALUE` specifies a minimum value of 1 for an ascending sequence and $-10^{26}$ for a descending sequence (default).
- `CYCLE` specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.).
- `CACHE `n` specifies how many values the Oracle server pre-allocates and keep in memory. (By default, the Oracle server caches 20 values.) If the system crashes, the values are lost. `NOCACHE` specifies that the Oracle server does not pre-allocate values.
Creating A Sequence

In the SEQUENCE created for the London Marathon runners, the numbers will increment by 1, starting with the number 1. In this case, beginning the sequence with 1 is probably the best starting point. It is a tradition that the best runner in the elite group wears number 1. For other situations, such as department IDs and employee IDs, the starting number may be assigned differently.

Because there will be at least 30,000 runners, the sequence's maximum value was set well above the expected number of runners.

```
CREATE SEQUENCE runner_id_seq
  INCREMENT BY 1
  START WITH 1
  MAXVALUE 50000
  NOCACHE
  NOCYCLE;
```
Creating A Sequence (continued)

The NOCACHE option prevents values in the SEQUENCE from being cached in memory, which in the event of system failure prevents numbers preallocated and held in memory from being lost.

The NOCYCLE option prevents the numbering from starting over at 1 if the value 50,000 is exceeded. Don't use the CYCLE option if the sequence is used to generate primary-key values unless there is a reliable mechanism that deletes old rows faster than new ones are added.

```
CREATE SEQUENCE runner_id_seq
    INCREMENT BY 1
    START WITH 1
    MAXVALUE 50000
    NOCACHE
    NOCYCLE;
```
Tell Me / Show Me

Confirming Sequences

To verify that a sequence was created, query the USER_OBJECTS data dictionary. To see all of the SEQUENCE settings, query the USER_SEQUENCES data dictionary as shown below. List the value names in the SELECT statement as shown below.

```
SELECT sequence_name, min_value, max_value, increment_by, last_number
FROM user_sequences;
```

If NOCACHE is specified, the last_number column in the above query displays the next available sequence number.

If CACHE is specified, the last_number column displays the next available number in the sequence which has not been cached into memory.
Tell Me / Show Me

NEXTVAL And CURRVAL Pseudocolumns

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name.

When you reference sequence.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.
Tell Me / Show Me

NEXTVAL And CURRVAL Pseudocolumns (continued)
The example below inserts a new department in the DEPARTMENTS table. It uses the DEPARTMENTS_SEQ sequence for generating a new department number as follows:

```
INSERT INTO departments
    (department_id, department_name, location_id)
VALUES  (departments_seq.NEXTVAL, 'Support', 2500);
```
NEXTVAL And CURRVAL Pseudocolumns (continued)

Suppose now you want to hire employees to staff the new department. The INSERT statement to be executed for all new employees can include the following code:

```sql
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq.CURRVAL, ...);
```

Note: The preceding example assumes that a sequence called EMPLOYEE_SEQ has already been created for generating new employee numbers.
NEXTVAL And CURRVAL Pseudocolumns (continued)

The CURRVAL pseudocolumn in the example below is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user’s session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When sequence.CURRVAL is referenced, the last value generated by that user’s process is returned.

```
INSERT INTO employees (employee_id,
    department_id, ...)
VALUES (employees_seq.NEXTVAL,
    dept_deptid_seq.CURRVAL, ...);
```
Using A Sequence
After you create a sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

You can use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement
Using A Sequence (continued)
You cannot use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement
Tell Me / Show Me

Using A Sequence (continued)

To continue our London Marathon example, the following syntax would be used to insert a new participant's information into the runners' table. The runner's identification number would be generated by retrieving the NEXTVAL from the sequence.

```
INSERT INTO runners(runner_id, first_name, last_name, address, city, state/province, country)
VALUES (runner_id_seq.NEXTVAL, 'Joanne', 'Everely', '1945 Brookside Landing', 'New York', 'NY', 'USA');
```
Tell Me / Show Me

Using A Sequence (continued)

To view the current value for the runners_id_seq, CURRVAL is used. Note the use of the DUAL table in this example. Oracle Application Developer will not execute this query, but you should understand how this works.

```sql
SELECT runner_id_seq.CURRVAL
FROM dual;
```

Cache sequences in memory provide faster access to sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory. 20 is the default number of sequence numbers cached.
Tell Me / Show Me

Nonsequential Numbers

Although sequence generators issue sequential numbers without gaps, this action occurs independent of a database commit or rollback. Gaps (nonsequential numbers) can be generated by:

- rolling back a statement containing a sequence, the number is lost.
- a system crash. If the sequence caches values into the memory and the system crashes, those values are lost.
- the same sequence being used for multiple tables. If you do so, each table can contain gaps in the sequential numbers.
Tell Me / Show Me

Viewing the Next Value

If the sequence was created with NOCACHE, it is possible to view the next available sequence value without incrementing it by querying the USER_SEQUENCES table.
Tell Me / Show Me

Modifying A Sequence

As with the other database objects you've created, a SEQUENCE can also be changed using the ALTER SEQUENCE statement. What if the London Marathon exceeded the 50,000 runner registrations and you needed to add more numbers? The sequence could be changed to increase the MAXVALUE without changing the existing number order.

```
ALTER SEQUENCE runner_id_seq
  INCREMENT BY 1
  MAXVALUE 999999
  NOCACHE
  NOCYCLE;
```
Modifying A Sequence (continued)
Some validation is performed when you alter a sequence. For example, a new MAXVALUE that is less than the current sequence number cannot be executed.

ALTER SEQUENCE runner_id_seq
  INCREMENT BY 1
  MAXVALUE 90
  NOCACHE
  NOCYCLE;

ERROR at line 1:
ORA-04009: MAXVALUE cannot be made to be less than the current value
ALTER SEQUENCE Guidelines

A few guidelines apply when executing an ALTER SEQUENCE statement. They are:

- You must be the owner or have the ALTER privilege for the sequence in order to modify it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order to restart the sequence at a different number.
Tell Me / Show Me

Removing A Sequence

To remove a sequence from the data dictionary, use the DROP SEQUENCE statement. You must be the owner of the sequence or have DROP ANY SEQUENCE privileges to remove it. Once removed, the sequence can no longer be referenced.

DROP SEQUENCE dept_deptid_seq;
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

NEXTVAL
CURRVAL
INCREMENT BY
Sequences
CACHE/ NOCACHE
MAXVALUE
NOMAXVALUE
CREATE SEQUENCE
STARTS WITH
CYCLE/ NOCYCLE
MINVALUE
NO MINVALUE
Summary

Objectives Summarized
In this lesson you have learned to:

• List at least three useful characteristics of a sequence
• Write and execute a SQL statement that creates a sequence
• Query the data dictionary using USER_SEQUENCES to confirm a sequence definition
• Apply the rules for using NEXTVAL to generate sequential unique numbers in a table
• List the advantages and disadvantages of caching sequence values
• Name three reasons why gaps can occur in a sequence
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Indexes and Synonyms
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Define an index and its use as a schema object
• Define ROWID and its use in locating information in a database
• Name the conditions that cause an index to be created automatically
• Create and execute a CREATE INDEX and DROP INDEX statement
• Create and execute a function-based index
• Create private and public synonym
Why Learn It?

Purpose

Can you imagine going to a book store or a library and finding all the books stacked on shelves row after row in no real order?

The only way to locate what you're looking for would be to examine every book in every row!

Not very efficient, wouldn't you agree?

You may be surprised to know that retrieval of data from a database is exactly like the rows of books.

For every query, a full table scan happens.
Why Learn It?

Purpose (continued)
Fortunately, there is a way to make finding data in a database more efficient than having to examine every row. Oracle uses an index to speed up the retrieval of rows. In this lesson, you will learn how and when to create an index as well as how to delete an index.

Also, in this lesson, you will learn how to create user-friendly names for database objects. Much like Internet Web addresses that eliminate having to know the Internet protocol address, synonyms provide a way to give an object a simpler name (sounds like aliases, doesn’t it?).
Indexes
An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer.
Indexes can be created explicitly or automatically. If you do not have an index on the column you’re selecting, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O (input/output) by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle Server. Once an index is created, no direct activity is required by the user.

A ROWID is a base 64 string representation of the row address containing block identifier, row location in the block and the database file identifier. Indexes use ROWID’s because they are the fastest way to access any particular row.
Indexes and Synonyms

Tell Me / Show Me

Indexes (continued)
Indexes are logically and physically independent of the table they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Note: When you drop a table, corresponding indexes are also dropped.
Tell Me / Show Me

Types Of Indexes
Two types of indexes can be created:

- **Unique index**: The Oracle Server automatically creates this index when you define a column in a table to have a PRIMARY KEY or a UNIQUE KEY constraint. The name of the index is the name given to the constraint. Although you can manually create a unique index, it is recommended that you create a unique constraint in the table, which implicitly creates a unique index.

- **Nonunique index**: This is an index that a user can create to speed up access to the rows. For example, to optimize joins, you can create an index on the FOREIGN KEY column, which speeds up the search to match rows to the PRIMARY KEY column.
Creating An Index

Create an index on one or more columns by issuing the CREATE INDEX statement:

CREATE INDEX index_name
ON table_name( column...,column)

To create an index in your schema, you must have the CREATE TABLE privilege. To create an index in any schema, you need the CREATE ANY INDEX privilege or the CREATE TABLE privilege on the table on which you are creating the index. Null values are not included in the index.
Creating An Index (continued)
For example, to improve the speed of query access to the TITLE column in the DJ on Demand D_CDS table:

CREATE INDEX d_cds_idx
ON d_cds(title);
Tell Me / Show Me

When To Create An Index
An index should be created only if:

- The column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a WHERE clause or a join condition
- The table is large and most queries are expected to retrieve less than 2-4% of the rows.
Tell Me / Show Me

When Not To Create An Index

When deciding whether or not to create an index, more is not always better. Each DML operation (INSERT, UPDATE, DELETE) that is performed on a table with indexes means that the indexes must be updated. The more indexes you have associated with a table, the more effort it takes to update all the indexes after the DML operation.
When Not To Create An Index (continued)

It is usually not worth creating an index if:

• The table is small
• The columns are not often used as a condition in the query
• Most queries are expected to retrieve more than 2-4 % of the rows in the table
• The table is updated frequently
• The indexed columns are referenced as part of an expression
Tell Me / Show Me

Composite Index

A composite index (also called a "concatenated" index) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index.
Tell Me / Show Me

Composite Index (continued)

Null values are not included in the composite index.

To optimize joins, you can create an index on the FOREIGN KEY column, which speeds up the search to match rows to the PRIMARY KEY column.

The optimizer does not use an index if the WHERE clause contains the IS NULL expression.
Tell Me / Show Me

Confirming Indexes

Confirm the existence of indexes from the USER_INDEXES data dictionary view. You can also check the columns involved in an index by querying the USER_IND_COLUMNS view.

The query shown on the next slide is a join between the USER_INDEXES table (names of the indexes and their uniqueness) and the USER_IND_COLUMNS (names of the indexes, table names, and column names) table.
Tell Me / Show Me

Confirming Indexes

SELECT ic.index_name,
ic.column_name,
ic.column_position col_pos,
ix.uniqueness
FROM user_indexes ix, user_ind_columns ic
WHERE ic.index_name = ix.index_name
AND ic.table_name = 'EMPLOYEES';
Tell Me / Show Me

Function-based Indexes

A function-based index stores the indexed values and uses the index based on a SELECT statement to retrieve the data.

A function-based index is an index based on expressions.

The index expression is built from table columns, constants, SQL functions and user-defined functions.
Function-based Indexes (continued)

Function-based indexes are useful when you don't know in what case the data was stored in the database. For example, you can create a function-based index that can be used with a SELECT statement using UPPER in the WHERE clause. The index will be used in this search.

```
CREATE INDEX upper_last_name_idx
ON employees (UPPER(last_name));

SELECT *
FROM employees
WHERE UPPER(last_name) = 'KING';
```
Function-based Indexes (continued)
Function-based indexes defined with the UPPER(column_name) or LOWER(column_name) keywords allow case-insensitive searches.

If you don't know how the employee last names were entered into the database, you could still use the index by entering uppercase in the SELECT statement.

When a query is modified using an expression in the WHERE clause, the index won't use it unless you create a function-based index to match the expression.
Tell Me / Show Me

Function-based Indexes (continued)

For example, the following statement allows for case-insensitive searches using the index:

```sql
CREATE INDEX upper_last_name_idx
ON d_partners (UPPER(last_name));
```

```sql
SELECT *
FROM d_partners
WHERE UPPER(last_name) = 'CHO';
```
Function-based Indexes (continued)

To ensure that the Oracle Server uses the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, the following statement is guaranteed to use the index, but without the WHERE clause the Oracle Server may perform a full table scan:

```sql
SELECT *
FROM d_partners
WHERE UPPER(last_name) IS NOT NULL
ORDER BY UPPER(last_name);
```

The Oracle Server treats indexes with columns marked DESC as function-based indexes. The columns marked DESC are sorted in descending order.
Tell Me / Show Me

Function-based Indexes (continued)

All these examples all used the UPPER and LOWER functions, but it is worth noticing that whilst these two are very frequently used in Function Based Indexes, the Oracle database is not limited to just them.

Any valid Oracle Build-in function can be used, as can Database Functions that you write yourself. There is only one extra rule you must remember: if you are writing your own functions to use in a Function Based Index, you must include the key word DETERMINISTIC in the function header.
Tell Me / Show Me

Function-based Indexes (continued)

In mathematics, a **deterministic system** is a system in which no randomness is involved in the development of future states of the system. Deterministic models therefore produce the same output for a given starting condition.

In Oracle, Deterministic declares that a function, when given the same inputs, will always return the exact same output. You must tell Oracle that the function is DETERMINISTIC and will return a consistent result given the same inputs.

The built-in SQL functions UPPER, LOWER and TO_CHAR are all defined as deterministic by Oracle so you can create an index on the UPPER of a column.
Function-based Indexes (continued)

Another example of Function Based Indexes is shown here. Using the d_events table querying this table to find any events planned for the Month of May.

```
SELECT *
FROM d_events
WHERE TO_CHAR(event_date,'mon') = 'may'
```

As you can see, this query results in a Full Table Scan, which can be a very expensive operation, if the table is big. Even though the event_date column is indexed, the index is not used, due to the TO_CHAR expression.
Tell Me / Show Me

Function-based Indexes (continued)

Once we create the following Function Based Index, we can run the same query, but this time avoid the expensive Full Table Scan.

CREATE INDEX d_evnt_dt_indx ON d_events (to_char(event_date,'mon'))

```
SELECT *
FROM d_events
WHERE TO_CHAR(event_date,'mon') = 'may'
```

Now, Oracle can use the index on the events_date column.
Tell Me / Show Me

Removing An Index

You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the DROP INDEX statement. To drop an index, you must be the owner of the index or have the DROP ANY INDEX privilege. If you drop a table, indexes and constraints are automatically dropped, but views and sequences remain.

**In the syntax:**

index is the name of the index

```
DROP INDEX upper_last_name_idx;
DROP INDEX index;
DROP INDEX d_cds_idx;
```
SYNONYM

In SQL, as in language, a synonym is a word or expression that is an accepted substitute for another word. Synonyms are used to simplify access to objects by creating another name for the object. Synonyms can make referring to a table owned by another user easier and shorten lengthy object names. For example, to refer to the amys_copy_d_track_listings table in your classmate's schema, you can prefix the table name with the name of the user who created it followed by a period and then the table name, as in USMA_SBHS_SQL01_S04.amy.
SYNONYM (continued)

Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views. The database administrator can create a public synonym accessible to all users and can specifically grant the CREATE PUBLIC SYNONYM privilege to any user, and that user can create public synonyms.
SYNONYM (continued)
CREATE [PUBLIC] SYNONYM synonym
FOR object;

In the syntax:
• PUBLIC: creates a synonym accessible to all users
• synonym: is the name of the synonym to be created
• object: identifies the object for which the synonym is created

CREATE SYNONYM dj_titles
FOR d_cds;
SYNONYM Guidelines

• The object cannot be contained in a package.
• A private synonym name must be distinct from all other objects owned by the same user.

To remove a synonym:

DROP [PUBLIC] SYNONYM name_of_synonym

DROP SYNONYM dj_titles;
Confirming A SYNONYM

The existence of synonyms can be confirmed by querying the USER_SYNONYMS data dictionary view.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synonym_name</td>
<td>Name of the synonym</td>
</tr>
<tr>
<td>Table_name</td>
<td>Owner of the object referenced by the synonym</td>
</tr>
<tr>
<td>Table_owner</td>
<td>Name of the object referenced by the synonym</td>
</tr>
<tr>
<td>Db_link</td>
<td>Database link referenced in a remote synonym</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

DROP INDEX
Synonym
CREATE PUBLIC SYNONYM
Non-unique index
Unique index
Composite index
Confirming index
Function-based index
Summary

Objectives Summarized
In this lesson you have learned to:

• Define an index and its use as a schema object
• Define ROWID and its use in locating information in a database
• Name the conditions that cause an index to be created automatically
• Create and execute a CREATE INDEX and DROP INDEX statement
• Create function-based indexes
• Create private and public synonyms
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Controlling User Access
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Compare the difference between object privileges and system privileges

• Construct the two commands required to enable a user to have access to a database

• Construct and execute a GRANT… ON …TO statement to assign privileges to objects in their schema to other users and/or PUBLIC

• Query the data dictionary to confirm privileges granted
Why Learn It?

Purpose
If you share a computer with others, whether at school or at home, you've probably had something you're working on or something you've saved either viewed, changed or deleted by someone else. Wouldn't it be nice to be able to control the privileges others have to your personal files? For databases, just as at school or home, data security is very important. In this lesson, you will learn how to grant or take away access to database objects as a means to control who can alter, delete, update, insert, index, or reference the database objects.
Controlling User Access

Tell Me / Show Me

Controlling User Access

In a multiple-user environment, you want to maintain security of the database access and use. With Oracle Server database security, you can do the following:

• Control database access
• Give access to specific objects in the database
• Confirm given and received privileges within the Oracle data dictionary
• Create synonyms for database objects
Database Security

Database security can be classified into two categories:

- system security
- data security

System security covers access and use of the database at the system level, such as creating users, usernames and passwords, allocating disk space to users, and granting the system privileges that users can perform such as creating tables, views and sequences. There are more than 100 distinct system privileges.

Data security (also known as object security) relates to object privileges which covers access and use of the database objects and the actions that those users can have on the objects. These privileges include being able to execute DML statements.
Tell Me / Show Me

Privileges and Schemas

Privileges are the right to execute particular SQL statements. The DBA is a high-level user with the ability to grant users access to the database and its objects. Users require system privileges to gain access to the database. They require object privileges to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to roles, which are named groups of related privileges.
Tell Me / Show Me

Privileges and Schemas (continued)
A schema is a collection of objects, such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.
In this course, your schema name is a combination of your city, state/country, your school name, course name and student number.

For example: uswa_skhs_sql01_s22
Tell Me / Show Me

System Security
This level of security covers access and use of the database at the system level. There are more than 100 distinct system privileges.

System privileges such as the ability to create or remove users, remove tables or backup tables are usually held only by the DBA.

The table on the right lists some of the system privileges which the DBA would not normally grant to other users. Would you want another user to be able to drop your tables?

<table>
<thead>
<tr>
<th>System Privilege</th>
<th>Operations Authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE USER</td>
<td>Grantee can create other Oracle users (a privilege required for a DBA role)</td>
</tr>
<tr>
<td>DROP USER</td>
<td>Grantee can drop another user</td>
</tr>
<tr>
<td>DROP ANY TABLE</td>
<td>Grantee can drop a table in any schema</td>
</tr>
<tr>
<td>BACKUP ANY TABLE</td>
<td>Grantee can backup any table in any schema with the export utility</td>
</tr>
<tr>
<td>SELECT ANY TABLE</td>
<td>Grantee can query tables, views, or snapshots in any schema</td>
</tr>
<tr>
<td>CREATE ANY TABLE</td>
<td>Grantee can create tables in any schema</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

System Privileges

The DBA creates the user by executing the CREATE USER statement. The user does not have any privileges at this point. The DBA can then grant required privileges to that user.

```
CREATE USER user
IDENTIFIED BY password;

CREATE USER scott
IDENTIFIED BY ur35scott;
```

<table>
<thead>
<tr>
<th>System Privilege</th>
<th>Operations Authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE USER</td>
<td>Grantee can create other Oracle users (a privilege required for a DBA role.)</td>
</tr>
<tr>
<td>DROP USER</td>
<td>Grantee can drop another user.</td>
</tr>
<tr>
<td>DROP ANY TABLE</td>
<td>Grantee can drop a table in any schema.</td>
</tr>
<tr>
<td>BACKUP ANY TABLE</td>
<td>Grantee can backup any table in any schema with the export utility.</td>
</tr>
<tr>
<td>SELECT ANY TABLE</td>
<td>Grantee can query tables, views, or snapshots in any schema.</td>
</tr>
<tr>
<td>CREATE ANY TABLE</td>
<td>Grantee can create tables in any schema.</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

System Privileges (continued)
Using the ALTER USER statement, a user can change their password.

ALTER USER scott
IDENTIFIED BY imscott35;
Tell Me / Show Me

User System Privileges

The DBA uses the GRANT statement to allocate system privileges to the user. System privileges determine what the user can do at the database level. Once the user has been granted the privileges, the user can immediately use those privileges.

GRANT privilege [, privilege...] TO user [, user| role, PUBLIC...];

GRANT create session, create table, create sequence, create view TO scott;

<table>
<thead>
<tr>
<th>System Privilege</th>
<th>Operations Authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SESSION</td>
<td>Connect to the database</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Create tables in the user’s schema.</td>
</tr>
<tr>
<td>CREATE SEQUENCE</td>
<td>Create a sequence in the user’s schema.</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>Create a view in the user’s schema.</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>Create a procedure, function, or package in the user’s schema.</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

User System Privileges (continued)
A user must have a CREATE SESSION privilege as well as having a user id if they are to be able to access a database.

You cannot issue the CREATE SESSION command in Oracle Application Express, this happens automatically behind the scenes.
Tell Me / Show Me

Object Security
This level of security covers access and use of the database objects and the actions users can have on those objects.
Tell Me / Show Me

Object Privileges

Each object has a particular set of grantable privileges. The table below lists the privileges for various objects. It is important to note the following four points regarding object privileges:

1. The only privileges that apply to a sequence are SELECT and ALTER.

Remember, a sequence uses ALTER to change the INCREMENT, MAXVALUE, CACHE/NOCACHE, or CYCLE/NOCYCLE options.

START WITH cannot be changed using ALTER.
Tell Me / Show Me

Object Privileges (continued)

2. You can grant UPDATE, REFERENCES, and INSERT on individual columns on a table. For example:
   GRANT UPDATE (auth_expense)
   ON d_partners TO allison_plumb

3. A SELECT privilege can be restricted by creating a view with a subset of columns and granting the SELECT privilege only on the view. You can't grant SELECT on individual columns.

<table>
<thead>
<tr>
<th>Object Privilege</th>
<th>Table</th>
<th>View</th>
<th>Sequence</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DELETE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>INDEX</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Object Privileges (continued)

- A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym. In other words, a synonym is simply a new, easier-to-use name. Using this name to grant a privilege is the same as granting the privilege on the table itself.

<table>
<thead>
<tr>
<th>Object Privilege</th>
<th>Table</th>
<th>View</th>
<th>Sequence</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INDEX</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>UPDATE</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

The PUBLIC Keyword
An owner of a table can grant access to all users by using the PUBLIC keyword.
The example shown below allows all users on the system to query data from Alice’s DEPARTMENTS table.

GRANT select
ON alice.departments
TO PUBLIC;
The PUBLIC Keyword (continued)
If a statement does not use the full name of an object, the Oracle server implicitly prefixes the object name with the current user’s name (or schema). If user Scott queries the DEPARTMENTS table, for example, the system selects from the SCOTT.DEPARTMENTS table.

If a statement does not use the full name of an object, and the current user does not own an object of that name, the system prefixes the object name with PUBLIC.

For example, if user Scott queries the USER_OBJECTS table, and Scott does not own such a table, the system selects from the data dictionary view by way of the PUBLIC.USER_OBJECTS public synonym.
Confirming Granted Privileges

If you attempt to perform an unauthorized operation, such as deleting a row from a table for which you do not have the DELETE privilege, the Oracle server does not permit the operation to take place.

If you receive the Oracle server error message “table or view does not exist,” you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege.
Tell Me / Show Me

View Privileges

You can access the data dictionary to view the privileges that you have. The chart shown describes various data dictionary views.

Using Oracle Application Express Developer, enter USER_ROLE_PRIVS, select the magnifying glass, then select any item to Query By Example. Their privileges will be returned.

<table>
<thead>
<tr>
<th>Data Dictionary View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLE_SYS_PRIVS</td>
<td>System privileges granted to roles</td>
</tr>
<tr>
<td>ROLE_TAB_PRIVS</td>
<td>Table privileges granted to roles</td>
</tr>
<tr>
<td>USER_ROLE_PRIVS</td>
<td>Roles accessible by the user</td>
</tr>
<tr>
<td>USER_TAB_PRIVS_MADE</td>
<td>Object privileges granted on the user’s objects</td>
</tr>
<tr>
<td>USER_TAB_PRIVS_RECD</td>
<td>Object privileges granted to the user</td>
</tr>
<tr>
<td>USER_COL_PRIVS_MADE</td>
<td>Object privileges granted on the columns of the user’s objects</td>
</tr>
<tr>
<td>USER_COL_PRIVS_RECD</td>
<td>Object privileges granted to the user on specific columns</td>
</tr>
<tr>
<td>USER_SYS_PRIVS</td>
<td>Lists system privileges granted to the user</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Terminology
Key terms used in this lesson include:
System privileges
Object privileges
System security
Schema
Role
Object security
Privilege
WITH GRANT OPTION
GRANT privilege
PUBLIC privilege
Database link
CREATE SESSION privilege
Summary

Objectives Summarized
In this lesson you have learned to:

• Compare the difference between object privileges and system privileges
• Construct the two commands required to enable a user to have access to a database
• Construct and execute a GRANT… ON …TO statement to assign privileges to objects in their schema to other users and/or PUBLIC
• Query the data dictionary to confirm privileges granted
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Creating and Revoking Object Privileges
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Explain what a ROLE is and what its advantages are
• Construct a statement to create a ROLE and GRANT privileges to it
• Construct a GRANT .. ON .. TO.. WITH GRANT OPTION statement to assign privileges to objects in their schema to other users and/or PUBLIC
• Construct and execute a statement to REVOKE object privileges from other users and/or from PUBLIC
• Distinguish between privileges and roles
• Explain the purpose of a database link
Why Learn It?

Purpose
If you share a computer with others, whether at school or at home, you've probably had something you're working on or something you've saved either viewed, changed, or deleted by someone else. Wouldn't it be nice to be able to control the privileges others have to your personal files?

For databases, just as at school or home, data security is very important.

In this lesson, you will learn how to grant or take away access to database objects as a means to control who can alter, delete, update, insert, index, or reference the database objects.
Tell Me / Show Me

Roles
A role is a named group of related privileges that can be granted to a user. This method makes it easier to revoke and maintain privileges. A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

To create and assign a role, first the DBA must create the role. Then the DBA can assign privileges to the role and the role to users.

SQL> CREATE ROLE manager;
Role created.

SQL> GRANT create table, create view TO manager;
Grant succeeded.

SQL> GRANT manager TO jennifer_cho;
Grant succeeded.
Roles (continued)

Use the following syntax to create a role:

```
CREATE ROLE role_name;
```

After the role is created, the DBA can use the GRANT statement to assign the role to users as well as assign privileges to the role. The example shown creates a manager role and then allows managers to create tables and views. It then grants the role to a user. Now the user can create tables and views. If users have multiple roles granted to them, they receive all of the privileges associated with all of the roles.

```
SQL> CREATE ROLE manager;
Role created.

SQL> GRANT create table, create view TO manager;
Grant succeeded.

SQL> GRANT manager TO jennifer_cho;
Grant succeeded.
```

Note: The CREATE ROLE is a system privilege that has not been issued to Academy classrooms.
Tell Me / Show Me

Characteristics Of Roles

- They are named groups of related privileges.
- They can be granted to users.
- They simplify the process of granting and revoking privileges.
- They are created by a DBA.
**Granting Object Privileges**

Use the following syntax to grant object privileges:

```
GRANT object_priv [(column_list)]
ON object_name
TO {user|role|PUBLIC}
[WITH GRANT OPTION];
```

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_priv</td>
<td>is an object privilege to be granted</td>
</tr>
<tr>
<td>column_list</td>
<td>specifies a column from a table or view on which privileges are granted</td>
</tr>
<tr>
<td>ON object_name</td>
<td>is the object on which the privileges are granted</td>
</tr>
<tr>
<td>TO user</td>
<td>role</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>grants object privileges to all users</td>
</tr>
<tr>
<td>WITH GRANT OPTION</td>
<td>Allows the grantee to grant the object privileges to other users and roles</td>
</tr>
</tbody>
</table>
Object Privileges Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges WITH GRANT OPTION.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

### Syntax Defined

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_priv</td>
<td>is an object privilege to be granted</td>
</tr>
<tr>
<td>column_list</td>
<td>specifies a column from a table or view on which privileges are granted</td>
</tr>
<tr>
<td>ON object_name</td>
<td>is the object on which the privileges are granted</td>
</tr>
<tr>
<td>TO user</td>
<td>role</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>grants object privileges to all users</td>
</tr>
<tr>
<td>WITH GRANT OPTION</td>
<td>Allows the grantee to grant the object privileges to other users and roles</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

GRANT Examples

Scott King (username scott_king) has created a d_songs table. In Example 1 on the right, all users are granted permission to SELECT from Scott's d_songs table.

Example 2 grants UPDATE privileges to Jennifer and to the manager role on specific columns in Scott's d_songs table.

1. GRANT select
   ON  d_songs
   TO  PUBLIC;

2. GRANT update (title, artist)
   ON  d_songs
   TO  jennifer_cho, manager;

3. SELECT *
   FROM  scott_king.d_songs;

4. CREATE SYNONYM songs
   FOR scott_king.d_songs;

5. SELECT *
   FROM  songs;
Tell Me / Show Me

GRANT Examples (continued)

If Jennifer now wants to SELECT data from Scott's table, the syntax she must use is listed in Example 3.

Alternatively, Jennifer could create a synonym for Scott's table and SELECT from the synonym. See the syntax in Examples 4 and 5.

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in that user’s schema. A user can grant any object privilege on any schema object that the user owns to any other user or role.

1. GRANT select
   ON d_songs
   TO PUBLIC;

2. GRANT update (title, artist)
   ON d_songs
   TO jennifer_cho, manager;

3. SELECT *
   FROM scott_king.d_songs;

4. CREATE SYNONYM songs
   FOR scott_king.d_songs;

5. SELECT *
   FROM songs;
WITH GRANT OPTION

A privilege that is granted using the WITH GRANT OPTION clause can be passed on to other users and roles by the grantee. Object privileges granted using the WITH GRANT OPTION clause are revoked when the grantor’s privilege is revoked.

The example below gives user Scott access to your d_songs table with the privileges to query the table and add rows to the table. The example also allows Scott to give others these privileges:

```sql
GRANT select, insert
ON d_songs
TO scott_king
WITH GRANT OPTION;
```
Tell Me / Show Me

The PUBLIC Keyword
An owner of a table can grant access to all users by using the PUBLIC keyword. The example shown below allows all users on the system to query data from Jason’s d_songs table:

```
GRANT select
ON jason_tsang.d_songs
TO PUBLIC;
```
Tell Me / Show Me

DELETE Object

If you attempt to perform an unauthorized operation, such as deleting a row from a table on which you do not have the DELETE privilege, the Oracle Server does not permit the operation to take place.

If you receive the Oracle Server error message “table or view does not exist,” you have done one of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privileges
View Privileges
You can access the data dictionary to view the privileges that you have. The chart at right describes various data dictionary views.

<table>
<thead>
<tr>
<th>Data Dictionary View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLE_SYS_PRIVS</td>
<td>System privileges granted to roles</td>
</tr>
<tr>
<td>ROLE_TAB_PRIVS</td>
<td>Table privileges granted to roles</td>
</tr>
<tr>
<td>USER_ROLE_PRIVS</td>
<td>Roles accessible by the user</td>
</tr>
<tr>
<td>USER_TAB_PRIVS_MADE</td>
<td>Object privileges granted on the user’s objects</td>
</tr>
<tr>
<td>USER_TAB_PRIVS_RECD</td>
<td>Object privileges granted to the user</td>
</tr>
<tr>
<td>USER_COL_PRIVS_MADE</td>
<td>Object privileges granted on the columns of the user’s objects</td>
</tr>
<tr>
<td>USER_COL_PRIVS_RECD</td>
<td>Object privileges granted to the user on specific columns</td>
</tr>
<tr>
<td>USER_SYS_PRIVS</td>
<td>Lists system privileges granted to the user</td>
</tr>
</tbody>
</table>
Revoking Object Privileges

You can remove privileges granted to other users by using the REVOKE statement. When you use the REVOKE statement, the privileges that you specify are revoked from the users that you name and from any other users to whom those privileges were granted through the WITH GRANT OPTION clause.
Revoking Object Privileges (continued)

Use the following syntax to revoke object privileges:

```
REVOKE {privilege [, privilege...]|ALL}
ON object
FROM {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

CASCADE CONSTRAINTS is required to remove any referential integrity constraints made to the object by means of the REFERENCES privilege.
Tell Me / Show Me

With Grant Option

The example below revokes SELECT and INSERT privileges given to user Scott on the d_songs table.

```
REVOKE select, insert
ON d_songs
FROM scott_king;
```

If a user is granted a privilege with the WITH GRANT OPTION clause, that user can also grant the privilege using the WITH GRANT OPTION clause. This means that a long chain of grantees is possible, but no circular grants are permitted. If the owner revokes a privilege from a user who granted privileges to other users, the revoke statement cascades to all privileges granted.
Tell Me / Show Me

With Grant Option (continued)

For example, if user A grants SELECT privileges on a table to user B, including the WITH GRANT OPTION clause, user B can grant to user C the SELECT privilege including the WITH GRANT OPTION clause as well. Now, user C can grant to user D the SELECT privilege.
With Grant Option (continued)

However, if user A revokes privileges from user B, then those privileges granted to users C and D are also revoked.
Tell Me / Show Me

Private and Public Synonyms
As mentioned earlier in this lesson, you can create a synonym to eliminate the need to qualify the object name with the schema and provide you with an alternative name for a table, view, sequence, procedure or other objects.

Synonyms can be either private (the default) or public. A public synonym can be created by Database Administrators, or database users who have been given the privileges to do so, but not everyone can automatically create public synonyms.

Note: The CREATE PUBLIC SYNONYM privilege has not been granted to Academy students.
Roles and Privileges differ in a number of ways:

A user **privilege** is a right to execute a particular type of SQL statement, or a right to access another user's object. All privileges are defined by Oracle.

**Roles**, on the other hand, are created by users (usually administrators) and are used to group together privileges or other roles. They are created to make it easier to manage the granting of multiple privileges or roles to users.

So Privileges come with the database and Roles are made by Database Administrators or users of a particular database.
Tell Me / Show Me

Database Links
A database link is a pointer that defines a one-way communication path from one Oracle database to another Oracle database. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.
Database Links (continued)

A database link connection is “one-way” in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A.

CREATE DATABASE LINK – In Oracle Application Express, there is no constant connection to the database, and as a result, this feature is not available.

```
scott_king

SELECT * FROM emp@HQ_ACME.COM

HQ_ACME.COM
database
```

Local

Remote
Tell Me / Show Me

Database Links (continued)
If local users on database B want to access data on database A, they must define a link that is stored in the data dictionary of database B. A database link connection gives local users access to data on a remote database. For this connection to occur, each database in the distributed system must have a unique global database name. The global database name uniquely identifies a database server in a distributed system.
The great advantage of database links is that they allow users to access another user’s objects in a remote database so that they are bounded by the privilege set of the object’s owner. In other words, a local user can access a remote database without having to be a user on the remote database.

The example shows a user scott_king accessing the EMP table on the remote database with the global name HQ.ACME.COM.
Database Links (continued)

Typically, the Database Administrator is responsible for creating the database link. The dictionary view USER_DB_LINKS contains information on links to which a user has access. Once the database link is created, you can write SQL statements against the data in the remote site. If a synonym is set up, you can write SQL statements using the synonym. For example:

```
CREATE PUBLIC SYNONYM HQ_EMP
  FOR emp@HQ.ACME.COM;
```

Then write a SQL statement that uses the synonym:
```
SELECT * FROM HQ_EMP;
```

You cannot grant privileges on remote objects.
Summary

Objectives Summarized
In this lesson you have learned to:

• Explain what a ROLE is and what its advantages are
• Construct a statement to create a ROLE and GRANT privileges to it
• Construct a GRANT .. ON .. TO.. WITH GRANT OPTION statement to assign privileges to objects in their schema to other users and/or PUBLIC
• Construct and execute a statement to REVOKE object privileges from other users and/or from PUBLIC
• Distinguish between privileges and roles
• Explain the purpose of a database link
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Regular Expressions
What Will I Learn?

Objectives
In this lesson, you will learn to:

• Describe regular expressions
• Use regular expressions to search, match, and replace strings in SQL statements
• Construct and execute regular expressions and check constraints
Why Learn It?

Purpose
Sometimes you have to find or replace a particular piece of text in column, text string, or document.

You already know how to perform simple pattern matching using LIKE and wildcards. Sometimes you might need to look for very complex text strings such as finding the word “Winchester” in a specified text or extracting all URLs from a piece of text. Other times you might be asked to do a more complex search such as finding all words whose every second character is a vowel.

Regular expressions are a method of describing both simple and complex patterns for searching and manipulating. They are used widely in the computing industry, and are not limited to Oracle. Oracle’s implementation of regular expressions is an extension of the POSIX (Portable Operating System for UNIX) and are as such, fully compatible with the POSIX standard, as controlled by the Institute of Electrical and Electronics Engineers (IEEE).
Regular Expressions

The use of regular expressions is based on the use of meta characters.

Meta characters are special characters that have a special meaning, such as a wildcard character, a repeating character, a non-matching character, or a range of characters. You can use several predefined meta character symbols in the pattern matching.

The next slides list the meta characters and provide a brief explanation of each.
Tell Me / Show Me

META Characters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches zero or more occurrences</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^/$</td>
<td>Matches the start-of-line/end-of-line</td>
</tr>
<tr>
<td>[ ]</td>
<td>Bracket expression for a matching list matching any one of the expressions represented in the list</td>
</tr>
<tr>
<td>{m}</td>
<td>Matches exactly <em>m</em> times</td>
</tr>
<tr>
<td>{m,n}</td>
<td>Matches at least <em>m</em> times but no more than <em>n</em> times</td>
</tr>
<tr>
<td>[: :]</td>
<td>Specifies a character class and matches any character in that class</td>
</tr>
</tbody>
</table>
## META Characters (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Can have 4 different meanings: 1. Stand for itself. 2. Quote the next character. 3. Introduce an operator. 4. Do nothing.</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more occurrence</td>
</tr>
<tr>
<td>?</td>
<td>Matches zero or one occurrence</td>
</tr>
<tr>
<td>.</td>
<td>Matches any character in the supported character set, except NULL</td>
</tr>
<tr>
<td>()</td>
<td>Grouping expression, treated as a single subexpression</td>
</tr>
<tr>
<td>[==]</td>
<td>Specifies equivalence classes</td>
</tr>
<tr>
<td>\n</td>
<td>Back-reference expression</td>
</tr>
<tr>
<td>[..]</td>
<td>Specifies one collation element, such as a multi-character element</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Regular Expression Examples
A simple regular expression is very similar to the wildcard searches you are already familiar with. Let’s take a look at an example: let’s look for the letter ‘a’ followed by the letter ‘c’.

As a regular expression, this would be done as: ‘a.c’. The same expression as a standard SQL wildcard search would be: WHERE column LIKE ‘a_c’.

Which of the following strings would match ‘a.c’?

Regular Expression Examples (continued)
The strings in red would match the search string ‘a.c’


The other examples fail either due to them being the wrong character in the wrong place or in the wrong case (uppercase not lowercase as specified in the search string).
Regular Expression Examples (continued)
Assume you were asked to list all employees with a first name of Stephen or Steven. If you used standard Oracle wildcard searching, this would be hard to achieve, but with regular expressions, you could simply specify '^[Ste(v|ph)en]$'.

“^” specifies the start of the string that is being searched
Uppercase “S”
lowercase “t”
lowercase “e”
“(“ starts a group
lowercase “v”
“|” specifies an OR
lowercase “p”
Lowercase “h”
“)” finishes the group of choices,
lowercase “e”
lowercase “n”
“$” specifies the end of the string that is being searched
Tell Me / Show Me

Regular Expression Functions

Oracle provides a set of SQL functions that you can use to search and manipulate strings using regular expressions. You can use these functions on any data type that holds character data such as `CHAR`, `CLOB` and `VARCHAR2`. A regular expression must be enclosed in single quotation marks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGEXP_LIKE</td>
<td>Similar to the LIKE operator, but performs regular expression matching instead of simple pattern matching</td>
</tr>
<tr>
<td>REGEXP_REPLACE</td>
<td>Searches for a regular expression pattern and replaces it with a replacement string</td>
</tr>
<tr>
<td>REGEXP_INSTR</td>
<td>Searches for a given string for a regular expression pattern and returns the position where the match is found</td>
</tr>
<tr>
<td>REGEXP_SUBSTR</td>
<td>Searches for a regular expression pattern within a given string and returns the matched substring</td>
</tr>
<tr>
<td>REGEXP_COUNT</td>
<td>Returns the number of times a pattern appears in a string. You specify the string and the pattern. You can also specify the start position and matching options (for example, c for case sensitivity).</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Regular Expression Function Examples
Use of the regular expression REGEXP_LIKE could be used to solve the problem of listing either Steven or Stephen:

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steven</td>
<td>King</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Regular Expression Function Examples (continued)

Searching for addresses that don’t start with a number and listing the position of the first non-alpha character in that address could be done like this:

```
SELECT street_address,
       REGEXP_INSTR(street_address,'[^[:alpha:]]')
FROM   locations
WHERE  REGEXP_INSTR(street_address,'[^[:alpha:]]') > 1;
```

Explanation and result can be found on the next slide.
Tell Me / Show Me

Regular Expression Function Examples (continued)
REGEXP_INSTR(street_address,'[^[:alpha:]]')

“[“ specifies the start of the expression
“^” indicates NOT
“[:alpha:]“ specifies alpha character class, i.e. not numbers
“]” ends the expression

<table>
<thead>
<tr>
<th>STREET_ADDRESS</th>
<th>REGEXP_INSTR(STREET_ADDRESS,'[^[:ALPHA:]]')</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magdalen Centre, The Oxford Science Park</td>
<td>9</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Regular Expression Function Examples (continued)
To return only the first word in a column containing a sentence you could issue following statement:

```sql
SELECT REGEXP_SUBSTR(street_address , ' \[^ \]+ ') "Road"
FROM locations;
```

<table>
<thead>
<tr>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jabberwocky</td>
</tr>
<tr>
<td>Interiors</td>
</tr>
<tr>
<td>Charade</td>
</tr>
<tr>
<td>Bloor</td>
</tr>
<tr>
<td>Centre,</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Regular Expression Function Examples (continued)

REGEXP_SUBSTR(street_address, ' [^ ]+ ')

"[" specifies the start of the expression
"^" indicates NOT
" " indicates a space
"]" ends the expression
"+" indicates one or more
" " indicates a space

<table>
<thead>
<tr>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>Via</td>
</tr>
<tr>
<td>Calle</td>
</tr>
<tr>
<td>Jabberwocky</td>
</tr>
<tr>
<td>Interiors</td>
</tr>
<tr>
<td>Zagora</td>
</tr>
<tr>
<td>Charade</td>
</tr>
</tbody>
</table>
Regular Expressions could also be used as part of the application code to ensure that only valid data is stored in the database. It is possible to include a call to a regular expression function in, for instance, a CHECK constraint. So if you want to ensure that no email addresses without ‘@’ were captured in a table in your database, you could simply add the following check constraint:

```
ALTER TABLE employees
ADD CONSTRAINT email_addr_chk
CHECK(REGEXP_LIKE(email,'@'));
```

This would ensure that all email addresses include an “@” sign.
Another example could be to check the format of telephone numbers:

```sql
CREATE TABLE contacts

    (contact_name  VARCHAR2(30),
     phone_number  VARCHAR2(30)
    CONSTRAINT c_contacts_pnf
        CHECK (REGEXP_LIKE (phone_number, '^\(\d{3}\) \d{3}-\d{4}$'))
    );
```

This constraint will ensure that all phone numbers are in the format of (XXX) XXX-XXXX.
Tell Me / Show Me

Regular Expressions in Check Constraints (continued)

(REGEXP_LIKE (phone_number, '^\([^\d\{3\}\)\d\{3\}-\d\{4\}$')

“^” The beginning of the string

“\(“ A left parenthesis where at backward slash (\) is used as an escape character indicating that the left parenthesis following it is a literal rather than a grouping expression

“\d\{3\}” Exactly three digits

“\)” A right parenthesis. The backward slash (\) is an escape character as before

“(space character)” A space character

“\d\{3\}” Exactly three digits

“-” A hyphen

“\d\{4\}” Exactly four digits

“$” The end of the string
Tell Me / Show Me

Regular Expressions in Check Constraints (continued)

So the following row would work:

```
INSERT INTO contacts VALUES ('Natacha Hansen', '(191) 167-7611')
```

1 row(s) inserted.

And this one would not:

```
INSERT INTO contacts VALUES ('Kasper Hansen', '(191) 167 7611')
```

ORA-02290: check constraint (US_CURR_SQL01_T01.C_CONTACTS_PNF) violated
Another use of Regular expressions in a check constraint could be to make sure a VARCHAR2 or CHAR column does not allow numbers:

```
ALTER TABLE contacts
ADD CONSTRAINT no_number_chk
CHECK (regexp_instr(contact_name,'[:digit:]')=0)
```

`[:digit:]` is the POSIX expression that identified digits or numeric values. REGEXP_INSTR returns the position of any digits, and if that returned value is not = 0, then the constraint will fail.
Regular Expressions in Check Constraints (continued)
So, the contact name ‘Natacha Hansen’ would be accepted by the database, because the number returned by regexp_insr would be 0.

regexp_insr(‘Natacha Hansen’,'[:digit:]') returns a 0, so this insert will work.

If we try to add a number to the name, for instance: ‘Natacha Hansen 1’, the insert fails. regexp_insr(‘Natacha Hansen 1’,'[:digit:]') returns 16.
16 = 0, is not true, so the insert fails.
Tell Me / Show Me

Subexpressions

From Oracle 11g we can also use subexpressions when we are using regular expressions.

Parenthesis are used to identify the subexpressions within the expression, and they are supported in the REGEXP_INSTR and REGEXP_SUBSTR functions.
Subexpressions (continued)

Look at this expression:

\[(1 \ 2 \ 3) \ (4 \ (5 \ 6) \ (7 \ 8) )\]

The subexpressions here are:

A. 1 2 3
B. 4 5 6 7 8
C. 5 6
D. 7 8
Subexpressions (continued)

Subexpressions are especially useful in the real world for instance when working with for example DNA sequencing.

Look at a partial example of a mouse DNA sequence:

“ccacctttccctccactcagtttcacctgtaaagcgtcctccctccctcatc cccatgccccccctacctgcagggtagagtaggtagctagaaaccagag agctccaagctccatctgtggagaggtggtgccttcctgctggctcagagagaggagaatttgcccaaagctgcctttgaacgatggagacatgattg cccgtaaagggtcctgaatgcatgagatgtcttttgagagtagccgggta cgggtaaaaaggctcatgagaactttgctccatcattacgatcgtggtaacaca catatgagtatagagacacattgaggccaaagagttgagattgagag”
Subexpressions (continued)

Imagine you are working with DNA sequences, and you have to find the offset (position from the start) of a specific sequence, starting with gtc followed by tcac and then aaag, in that order. This could easily be done with the REGEXP_INSTR function, which would return the position where a match is found.

```
SELECT REGEXP_INSTR('ccacctttccctccactcagttctcacctgtaaagcgtccctccctcatcccccatgcccccctttaccctgcagggtagagtagggctagaaaccagagaggtcctcaag
ctccatctgtgagaggtgcactcttggtggctgagagagaggagaatttgcccaaaagcgtgccctgacagtgcagtcaggtacctgcacgttccgtgatgtcattacgacggttaa
ctcacatatgagtatagacacattggccagagttgagatggtggttacggtacttgcagtcgcagggaggtgctgacttgcttgggctgctgagagaggagacgtgc
ccctttttgagatggagacagttacggttaaaggtcatgagactttcgatcattacgatcgtgtaatacacacatagtagagacacattttgccccagagtagtttagagag',
'gtc(tcac)(aaag))',1,1,0,'i",1) "Position"
FROM DUAL;
```
Subexpressions (continued)

```
SELECT REGEXP_INSTR('ccacctttccccctccactcagtttcacctgttaaagcgctccctccctcatcccccatgcccccttaccctg
cagggtagagtggctagaaaccagagagctccaagctccatctgtggagaggtgacctcttggtggctgcagagagagag
aattgccaaauggctgcctgtttgaacgatggagacatgattgcccgttaaaggtctctgtctcacaagggagagatgtctttcagag
agtaccggtttacgggttaaaggtcatgagacttccgatcattacgatctgtttgaacgatggagacatgattgcccgttaaaggtctctgtctcacaagggagagatgtctttcagag
agtaccggtttacgggttaaaggtcatgagacttccgatcattacgatctgtttgaacgatggagacatgattgcccgttaaaggtctctgtctcacaagggagagatgtctttcagag', -- The string you are searching in
'(gtc(tcac)(aaag))', -- The subexpressions, here we have 3
1, -- Start position of search
1, -- Identifies occurrence of pattern you are searching for. 1 means First occurrence
0, -- Return option. The position of the character following the occurrence is returned. 0 is position of match and 1 means the character position after the occurrence is returned.
'i', -- Case insensitive or not
0) "Position" -- Which subexpression you want returned, 1, 2 or 3 would be valid values in this example, as we have 3 subexpressions
FROM DUAL;
```
Oracle 11g also has a new regular expression function: REGEXP_COUNT. This function greatly simplifies counting the number of times a pattern appears inside a string.

So using the Mouse DNA example, if we wanted to count how many times the pattern ‘gtc’ was represented in the DNA sample, we could simply count them.

```sql
SELECT REGEXP_COUNT('ccacctttccctccactcagtctcactgtaaagcgtccctccctcatccccatgccccctta
cctgcagggtagtaggcttagaaccagagagctcaagctccatctgtgagaggtgcccctttgggtgcgtcagagagagag
aatggcccaagcgtgctgtttgaacgatggagacatgattgcgccgtaaagggtctctgtctcacaagggagatgtctttcgaga
gtaccgggtttacgggtaaaaggctcatgagacttcgatcattacgatcgtgtgtaaacacagcatatagatagagacacattgccc
aagagtggagattgagag',
'gtc') AS "Count"
FROM DUAL;
```

<table>
<thead>
<tr>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

1 rows returned in 0.00 seconds
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

REGULAR EXPRESSIONS
Subexpressions
Summary

Objectives Summarized
In this lesson you have learned to:

- Describe regular expressions
- Use regular expressions to search, match, and replace strings in SQL statements
- Construct and execute regular expressions and check constraints
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Database Transactions
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Define the terms COMMIT, ROLLBACK, and SAVEPOINT as they relate to data transactions

• List three advantages of the COMMIT, ROLLBACK and SAVEPOINT statements

• Explain why it is important, from a business perspective, to be able to control the flow of transaction processing
Why Learn It?

Purpose
What if a bank had no systematic process for recording deposits and withdrawals?

How would you know if a deposit was credited to your account before you needed to withdraw money? You can imagine the confusion it would cause.

Fortunately, banks control transaction processes to ensure data consistency.

In this lesson you will learn how the process of changing data is managed and how changes to a database are committed or cancelled.
Transactions

Transactions are a fundamental concept of all database systems. Transactions allow users to make changes to data then decide whether to save or discard the work.

Database transactions bundle multiple steps into one logical unit of work. A transaction consists of one of the following:

- DML statements which constitute one consistent change to the data. The DML statements include INSERT, UPDATE, DELETE and MERGE
- One DDL statement such as CREATE, ALTER, DROP, RENAME or TRUNCATE
- One DCL statement such as GRANT or REVOKE
Transaction Analogy

A bank database contains balances for various customer accounts, as well as total deposit balances for other branches. Suppose a customer wants to withdraw and transfer money from their account and deposit it into another customer’s account at a different branch.

There are several separate steps involved to accomplish this rather simple operation. Both bank branches want to be assured that either all steps in the transaction happen, or none of them happen and if the system crashes, the transaction is not left partially complete. Grouping the withdrawal and deposit steps into a transaction provides this guarantee.

A transaction either happens completely or not at all.
Tell Me / Show Me

Controlling Transactions

Transactions are controlled using the following statements:

**COMMIT:** Represents the point in time where the user has made all the changes he or she wants to have logically grouped together, and because no mistakes have been made, the user is ready to save the work. When a COMMIT statement is issued, the current transaction ends making all pending changes permanent.

**ROLLBACK:** Enables the user to discard changes made to the database. When a ROLLBACK statement is issued, all pending changes are discarded.

**SAVEPOINT:** Creates a marker in a transaction, which divides the transaction into smaller pieces.

**ROLLBACK TO SAVEPOINT:** Allows the user to roll back the current transaction to a specified a savepoint. If an error was made, the user can issue a ROLLBACK TO SAVEPOINT statement discarding only those changes made after the SAVEPOINT was established.
Transaction Example

In the example shown, the user has issued an UPDATE statement and immediately created SAVEPOINT one. After an INSERT statement and an UPDATE statement, the user realized that a WHERE clause was not included in the last UPDATE. To remedy the mistake, the user issued a ROLLBACK TO SAVEPOINT one. The data is now restored to its state at SAVEPOINT one.

```
UPDATE d_cds
SET cd_number = 96
WHERE title = 'Graduation Songbook';
SAVEPOINT one;
INSERT INTO d_cds(cd_number, title, producer, year)
VALUES(100, 'Go For It', 'The Music Man', 2004) ;
UPDATE d_cds
SET cd_number = 101;
ROLLBACK TO SAVEPOINT one;
COMMIT;
```
When Does a Transaction Start or End?

A transaction **begins** with the first DML (INSERT, UPDATE, DELETE or MERGE) statement.

A transaction **ends** when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued
- A DDL(CREATE, ALTER, DROP, RENAME or TRUNCATE) statement is issued
- A DCL(GRANT or REVOKE) statement is issued
- The user exits iSQL*Plus or SQL*Plus
- A machine fails or the system crashes

After one transaction ends, the next executable SQL statement automatically starts the next transaction. A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction. Every data change made during a transaction is temporary until the transaction is committed.
Data Consistency

Imagine spending several hours making changes to employee data only to find out that someone else was entering information that conflicted with your changes!

To prevent such disruptions or conflicts and to allow multiple users to access the database at the same time, database systems employ an automatic implementation called "read consistency."

Read consistency guarantees a consistent view of the data by all users at all times. Readers do not view data that is in the process of being changed. Writers are ensured that the changes to the database are done in a consistent way. Changes made by one writer do not destroy or conflict with changes another writer is making.
Read Consistency
Read consistency is an automatic implementation.
A partial copy of the database is kept in undo segments. When User A issues an insert, update or delete operation to the database, the Oracle server takes a snapshot (copy) of the data before it is changed and writes it to an undo (rollback) segment. User B still sees the database as it existed before the changes started; s/he views the undo segment’s snapshot of the data.

Before changes are committed to the database, only the user who is changing the data sees the changes; everyone else sees the snapshot in the undo segment. This guarantees that readers of the data see consistent data that is not currently undergoing change.
Changes Visible

When a DML statement is committed, the change made to the database becomes visible to anyone executing a SELECT statement.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.
- All users see the database as it existed before the transaction began.
COMMIT, ROLLBACK and SAVEPOINT

COMMIT and ROLLBACK ensure data consistency, making it possible to preview data changes before making changes permanent and a way to group logically related operations.

SAVEPOINT allows to create a point in a transaction to which we can rollback without having to undo the entire transaction. However, SAVEPOINT is not supported in Oracle Application Express, due to the way Oracle Application Express manages connections to the database.
**Tell Me / Show Me**

**COMMIT, ROLLBACK and SAVEPOINT (Continued)**

In the transaction shown in the graphic, a DELETE statement was issued and then SAVEPOINT A was established. This SAVEPOINT acts like a marker that will allow the user to rollback any subsequent changes made to the data back to the state of the data as it existed at this point.
COMMIT, ROLLBACK and SAVEPOINT (Continued)

In the example, following SAVEPOINT A, the user issues some INSERT and UPDATE statements, then establishes another rollback marker at SAVEPOINT B. If for some reason the user does not want these INSERT and/or UPDATE statements to occur, the user can issue a ROLLBACK TO SAVEPOINT A statement. This will rollback to the state of the data as it was at the SAVEPOINT A marker.
COMMIT, ROLLBACK and SAVEPOINT (Continued)

Adding other SAVEPOINTS creates additional markers for rollback points. If a user issues a ROLLBACK without a ROLLBACK TO SAVEPOINT statement, the entire transaction is ended and all pending data changes are discarded.
Implicit Transaction Processing

Automatic commit of data changes occurs under the following circumstances:

- a DDL statement is issued
- a DCL statement is issued
- a normal exit from iSQL*Plus or SQL*Plus without explicitly issuing COMMIT or ROLLBACK statements
- When the Autocommit box is checked in Oracle Application Express

Automatic rollback occurs under an abnormal termination of iSQL*Plus, SQL*Plus or when a system failure occurs.

This prevents any errors in the data from causing unwanted changes to the underlying tables. The integrity of the data is therefore protected.
Locking

It is important to prevent data from being changed by more than one user at a time. Oracle uses locks that prevent destructive interaction between transactions accessing the same resource, either a user object (such as tables or rows) or a system object not visible to users (such as shared data structures and data dictionary rows).
How the Oracle Database Locks Data

Oracle locking is performed automatically and requires no user action. Implicit locking occurs for SQL statements as necessary, depending on the action requested. Implicit locking occurs for all SQL statements except SELECT.

The users can also lock data manually, which is called explicit locking.

When a COMMIT or ROLLBACK statement is issued, locks on the affected rows are released.
Terminology
Key terms used in this lesson include:

Transaction
Commit
Savepoint
Rollback
Read consistency
Locks
Summary

Objectives Summarized
In this lesson you have learned to:

• Define the terms COMMIT, ROLLBACK, and SAVEPOINT as they relate to data transactions
• List three advantages of the COMMIT, ROLLBACK, and SAVEPOINT statements
• Explain why it is important, from a business perspective, to be able to control the flow of transaction processing
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Cartesian Product and the Join Operations
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Name the Oracle proprietary joins and their ANSI/ISO SQL: 1999 counterparts
• Describe the purpose of join conditions
• Construct and execute a SELECT statement that results in a Cartesian product
• Construct and execute SELECT statements to access data from more than one table using an equijoin
• Construct and execute SELECT statements that add search conditions using the AND operator
• Apply the rule for using column aliases in a join statement
Why Learn It?

Purpose
Querying and returning information from one database table at a time would not be a problem if all data in the database were stored in only one table. But you know from data modeling that separating data into individual tables and being able to associate the tables with one another is the heart of relational database design. Fortunately, SQL provides join conditions that enable information to be queried from separate tables and combined in one report.
Tell Me / Show Me

Join Commands

The two sets of commands or syntax which can be used to make connections between tables in a database:

- Oracle proprietary joins
- ANSI/ISO SQL 99 compliant standard joins
ORACLE Proprietary Joins
To query data from more than one table using the Oracle proprietary syntax, use a join condition in the WHERE clause.

The basic format of a join statement is:

```
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column1 = table2.column2;
```
ORACLE Proprietary Joins (continued)

SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column1 = table2.column2;

Imagine the problem arising from having two students in the same class with the same last name. When needing to speak to "Jackson," the teacher clarifies which "Jackson" by prefacing the last name with the first name. To make it easier to read a Join statement and to speed up database access, it is good practice to preface the column name with the table name. This is called "qualifying your columns." The combination of table name and column name helps eliminate ambiguous names when two tables contain a column with the same column name. Note: When the same column name appears in both tables, the column name must be prefaced with the name of the table.
Tell Me / Show Me

Join Syntax Example

In the example at right, which two tables are being joined? Which identical columns do these tables share?

If you wanted to join three tables together, how many joins would it take? How many bridges are needed to join three islands?

```
SELECT d_play_list_items.event_id,
       d_play_list_items.song_id,
       d_track_listings.cd_number
FROM    d_play_list_items,
        d_track_listings
WHERE d_play_list_items.song_id =
       d_track_listings.song_id;
```

Table 1  Table 2  Table 3
Tell Me / Show Me

EQUIJOIN

Sometimes called a "simple" or "inner" join, an equijoin is a table join that combines rows that have the same values for the specified columns. In the example shown, the what, where and how are required for the join condition.

What? The SELECT clause specifies the column names to retrieve.

Where? The FROM clause specifies the two tables that the database must access.

How? The WHERE clause specifies how the tables are to be joined.

```
SELECT d_play_list_items.song_id,
       d_play_list_items.event_id,
       d_track_listings.cd_number
FROM    d_play_list_items,
        d_track_listings
WHERE   d_play_list_items.song_id =
        d_track_listings.song_id;
```

<table>
<thead>
<tr>
<th>SONG_ID</th>
<th>EVENT_ID</th>
<th>CD_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>46</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>47</td>
<td>100</td>
<td>91</td>
</tr>
<tr>
<td>48</td>
<td>105</td>
<td>95</td>
</tr>
<tr>
<td>49</td>
<td>105</td>
<td>91</td>
</tr>
<tr>
<td>47</td>
<td>105</td>
<td>91</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Cartesian Product Join

If two tables in a join query have no join condition specified in the WHERE clause or the join condition is invalid, the Oracle Server returns the Cartesian product of the two tables. This is a combination of each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows! This may not be what you were trying to retrieve.

To avoid a Cartesian product, always include a valid join condition in a WHERE clause.
Tell Me / Show Me

Generating a Cartesian Product

EMPLOYEES (20 rows)

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>110</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>110</td>
</tr>
</tbody>
</table>

20 rows selected.

DEPARTMENT_ID | DEPARTMENT_NAME | LOCATION_ID |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td>1700</td>
</tr>
</tbody>
</table>

8 rows selected.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>90</td>
<td>1700</td>
</tr>
<tr>
<td>101</td>
<td>90</td>
<td>1700</td>
</tr>
<tr>
<td>102</td>
<td>90</td>
<td>1700</td>
</tr>
<tr>
<td>103</td>
<td>60</td>
<td>1700</td>
</tr>
<tr>
<td>104</td>
<td>60</td>
<td>1700</td>
</tr>
<tr>
<td>107</td>
<td>60</td>
<td>1700</td>
</tr>
</tbody>
</table>

160 rows selected.
Tell Me / Show Me

Restricting Rows In a Join
As with single-table queries, the WHERE clause can be used to restrict the rows considered in one or more tables of the join. The query shown uses the AND operator to restrict the rows returned. Compare this result with the previous query.

```
SELECT d_play_list_items.song_id,
d_play_list_items.event_id,
d_track_listings.cd_number
FROM   d_play_list_items,
       d_track_listings
WHERE d_play_list_items.song_id =
       d_track_listings.song_id
AND   d_play_list_items.event_id < 105;
```

<table>
<thead>
<tr>
<th>SONG_ID</th>
<th>EVENT_ID</th>
<th>CD_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>46</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>47</td>
<td>100</td>
<td>91</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

**Aliases**

Working with lengthy column and table names can be cumbersome. Fortunately, there is a way to shorten the syntax using aliases. To distinguish columns that have identical names but reside in different tables, use column aliases. Column aliases were used in the query below. When there are no shared column names between two tables, there is no need to add the table name to it.

```
SELECT d_track_listings.song_id AS TRACK, d_play_list_items.song_id AS "PLAY LIST"
FROM   d_play_list_items,
       d_track_listings
WHERE  d_play_list_items.song_id = d_track_listings.song_id;
```

<table>
<thead>
<tr>
<th>TRACK</th>
<th>PLAYLIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>47</td>
<td>47</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Table Alias

Another way to make statements easier to read is to use table aliases. A table alias is just like a column alias, as in, it renames an object within a statement. It is created by entering the new name for the table just after the table name in the from-clause. However, if a table alias is used in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.

```
SELECT p.song_id, t.song_id
FROM d_play_list_items p, d_track_listings t
WHERE p.song_id = t.song_id;
```
Tell Me / Show Me

Terminology
Key terms used in this lesson include:

Alias
Cartesian product
Equijoin
Joint conditions
Proprietary join
Objectives Summarized

In this lesson you have learned to:

- Name the Oracle proprietary joins and their ANSI/ISO SQL: 1999 counterparts
- Describe the purpose of join conditions
- Construct and execute a SELECT statement that results in a Cartesian product
- Construct and execute SELECT statements to access data from more than one table using an equijoin
- Construct and execute SELECT statements that add search conditions using the AND operator
- Apply the rule for using column aliases in a join statement
Summary

Practice Guide
The link for the lesson practice guide can be found in the course outline in Section 0ne.
Nonequijoins
Nonequijoins

What Will I Learn?

Objectives
In this lesson, you will learn to:

• Construct and execute a SELECT statement to access data from more than one table using a nonequijoin
Purpose
What happens if you want to retrieve data from a table that has no corresponding column in another table? For instance, your math percentage grade of 92 is stored in the GRADES column in one table; the letter grade is stored in the LETTER_GRADE column in another table. How can we join the number grade with the letter grade? When data is recorded using a range, retrieving it is the job of a nonequijoin.
Tell Me / Show Me

Nonequijoin

Example:
A company pays its employees who earn an hourly wage in 15-minute increments. One table stores the hours and minutes recorded by a time clock and another table stores the pay range. If the minutes worked is between 0 and 15, the worker is paid for 15 minutes. If the minutes worked is between 16 and 30, the worker is paid for 30 minutes. If the minutes worked is between 31 and 45, the worker is paid for 45 minutes. If the minutes worked is between 46 and 60, the worker is paid for 1 hour.

00-15 Logged = 15 Paid
16-30 Logged = 30 Paid
31-45 Logged = 45 Paid
46-60 Logged = 60 Paid
Nonequijoin (continued)

To join your number grade (or %) in math with its corresponding letter grade, a nonequijoin is needed. Since there is no exact match between the two columns in each table, the equality operator = can't be used. Although comparison conditions such as < = and > = can be used, BETWEEN...AND is a more effective way to execute a nonequijoin.
Nonequi join (Continued)
The query shown joins the D_EVENTS cost column with the D_PACKAGES low_range and high_range columns using BETWEEN...AND

```
SELECT d_packages.code, d_events.cost
FROM d_packages, d_events
WHERE d_events.cost BETWEEN d_packages.low_range AND d_packages.high_range
```

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>EVENT_DATE</th>
<th>DESCRIPTION</th>
<th>COST</th>
<th>VENUE_ID</th>
<th>PACKAGE_CODE</th>
<th>THEME_CODE</th>
<th>CLIENT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Peters Graduation</td>
<td>14-MAY-04</td>
<td>Party for 200</td>
<td>8000</td>
<td>100</td>
<td>112</td>
<td>200</td>
<td>5922</td>
</tr>
<tr>
<td>100</td>
<td>Vigils Wedding</td>
<td>28-APR-04</td>
<td>Black tie, Four Seasons</td>
<td>10000</td>
<td>220</td>
<td>200</td>
<td>200</td>
<td>6133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CODE</th>
<th>LOW_RANGE</th>
<th>HIGH_RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>500</td>
<td>2500</td>
</tr>
<tr>
<td>87</td>
<td>2501</td>
<td>5000</td>
</tr>
<tr>
<td>112</td>
<td>5001</td>
<td>10000</td>
</tr>
<tr>
<td>200</td>
<td>10001</td>
<td>15000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CODE</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>8000</td>
</tr>
<tr>
<td>112</td>
<td>10000</td>
</tr>
</tbody>
</table>
Summary

Objective Summarized
In this lesson you have learned to:

• Construct and execute a SELECT statement to access data from more than one table using a nonequijoin
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Outer Joins
What Will I Learn?

Objective
In this lesson, you will learn to:

- Create and execute a SELECT statement to access data from more than one table using an outer join
Why Learn It?

Purpose
The joins you've studied so far returned rows that either had a matching value in both tables or a value that in one table was between two values in a different table. Those rows that didn't satisfy these conditions were just left out. Sometimes, however, you want all the data from one of the tables even if there is no matching data in the other table. If you wanted to know all the girls or boys who attended a dance, would simply counting the couples dancing work? In Oracle SQL, the missing data can be returned using an outer join.
Tell Me / Show Me

Outer Join
An outer join is used to see rows that have a corresponding value in another table plus those rows in one of the tables that have no matching value in the other table. To indicate which table may have missing data, use a plus sign (+) after the table’s column name in the WHERE clause of the query.
Note that an outer join cannot use the IN operator or be linked to another condition by the OR operator.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>90</td>
</tr>
<tr>
<td>178</td>
<td>Grant</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
</tr>
<tr>
<td>210</td>
<td>Human Resources</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Hartstein</td>
</tr>
<tr>
<td>20</td>
<td>Fay</td>
</tr>
<tr>
<td>110</td>
<td>Higgins</td>
</tr>
<tr>
<td>110</td>
<td>Gietz</td>
</tr>
<tr>
<td></td>
<td>Grant</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Join With Plus Sign
The query below uses the plus sign to indicate the table whose column is missing data. The variations of the outer join are shown.

```
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column(+) = table2.column;
```

```
WHERE table1.column = table2.column(+);
```

```
NEVER table1.column(+) = table2.column(+);
```

```
EMPLOYEE_ID | LAST_NAME | DEPARTMENT_ID
-------------|-----------|---------------
100          | King      | 90            
101          | Kochhar   | 90            
102          | De Haan   | 90            
178          | Grant     |               
```

```
DEPARTMENT_ID | DEPARTMENT_NAME
---------------|------------------
90             | Executive        
110            | Accounting       
190            | Contracting      
210            | Human Resources  
```

```
SELECT d.department_id, e.last_name
FROM employees e, departments d
WHERE e.department_id = d.department_id (+);
```

```
DEPARTMENT_ID | LAST_NAME
---------------|-----------
20             | Hartstein 
20             | Fay       
110            | Higgins   
110            | Gietz     
```

Copyright © 2011, Oracle. All rights reserved.
Summary

Objective Summarized
In this lesson you have learned to:
• Create and execute a SELECT statement to access data from more than one table using an outer join
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.
Testing
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Develop and apply a strategy for testing that a database functions as designed
Why Learn It?

Purpose
Most people, when they buy a car wish to know that it is reliable and will not break down.

So the manufacturers will put the car through a number of tests before it is available to be sold.

The same is true a database, before it is sold to a customer, it is tested that it meets the business requirements.
**Tell Me / Show Me**

**Unit Testing**

If two things are tested at once and the test fails, it is difficult or impossible to work out what has caused the failure. So it is important to test only one thing at a time. This is commonly referred to as unit testing.
Tell Me / Show Me

What Could Be Tested?

When testing a database there is a range of different things that need to be tested.

- Columns should be tested that they contain the correct data type.
- Columns should be tested that they can accommodate the largest amount of data that might be entered.
- Constraints should be checked that they only constrain or limit data that they are supposed and not more or less data.
Tell Me / Show Me

What Should Be Tested?

It is frequently unrealistic to test every column and every constraint in every table in a database if it is a large database. A random spread of tests, that check some columns and some constraints, should be carried out.
Tell Me / Show Me

Designing Tests

Before you carry out a test you should have a good idea of what result you expect to see if the database is working as expected. This should be documented before you carry out the test in a table similar to the one shown:

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Date</th>
<th>Test Description</th>
<th>Input</th>
<th>Expected Output</th>
<th>Result/Discrepancy</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>19/07/06</td>
<td>Confirm NOT NULL constraint on JOB_TITLE in JOBS table</td>
<td>INSERT INTO jobs (job_id, job_title, min_salary, max_salary) VALUES (222,NULL,100,200)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Running Tests
Once you have designed your test, you can run it and record your results:

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Date</th>
<th>Test Description</th>
<th>Input</th>
<th>Expected Output</th>
<th>Result/Discrepancy</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>19/07/06</td>
<td>Confirm NOT NULL constraint on JOB_TITLE in JOBS table</td>
<td>INSERT INTO jobs (job_id, job_title, min_salary, max_salary) VALUES (222,NULL,100,200)</td>
<td>Cannot insert NULL…</td>
<td>Cannot insert NULL…</td>
<td>None</td>
</tr>
</tbody>
</table>
What Will I Learn?

Objectives Summarized

In this lesson, you have learned how to:

- Develop and apply a strategy for testing that a database functions as designed
Summary

Practice Guide

The link for the lesson practice guide can be found in the course resources in Section 0.
Final Project Database Creation
What Will I Learn?

Objective

In this lesson, you will learn to:

• Apply SQL concepts to create a functional database appropriate for a small business
Why Learn It?

Purpose
Most young people know a lot about cars before they ever drive one or know a lot about other parts of the world without even having been there.

It would be a shame to have spent a year studying database modeling and database programming without really creating a database application!

In this lesson, you will have an opportunity to create a functioning database appropriate for a small business.
Tell Me / Show Me

Scope of the Final Project
The Final Project consists of the following steps:

Part 1: Create Table diagram from ERD

Part 2: Write and run scripts to create tables

Part 3: Insert sample data

Part 4: Test database

Part 5: Present final project
Tell Me / Show Me

Evaluation of the Final Project
Each group will present their database to the class. Your group will be evaluated according the Final Project Rubric. Make sure that you read it carefully and understand how you will be assessed.
Summary

Objective Summarized

In this lesson, you have learned to:

• Apply SQL concepts to create a functional database appropriate for a small business
Ensuring Quality Query Results - Advanced Techniques
What Will I Learn?

Objectives

In this lesson, you will learn to:

• Create an advanced query to produce specified data
• Modify an advanced query to produce specified data
Why Learn It?

Purpose
You’ve learned the syntax rules for generating a SQL query but are you sure your are producing the desired data? Looking at the desired output and then figuring out the query to generate that output helps you to gain confidence that your query results are what you expect.
Tell Me / Show Me

Create These Tables

create table emp as select * from employees;
create table dept as select * from departments;
Tell Me / Show Me

Write the Query

**Problem:**

Produce a report that lists the constraint name, type, column name and column position of all the constraints on the JOB_HISTORY table, apart from the not null constraints.

**Tables Used:**

user_constraints, user_cons_columns

<table>
<thead>
<tr>
<th>CONSTRAINT_NAME</th>
<th>CONSTRAINT_TYPE</th>
<th>COLUMN_NAME</th>
<th>POSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHIST_EMP_ID_ST_DATE_PK</td>
<td>P</td>
<td>EMPLOYEE_ID</td>
<td>1</td>
</tr>
<tr>
<td>JHIST_EMP_ID_ST_DATE_PK</td>
<td>P</td>
<td>START_DATE</td>
<td>2</td>
</tr>
<tr>
<td>JHIST_JOB_FK</td>
<td>R</td>
<td>JOB_ID</td>
<td>1</td>
</tr>
<tr>
<td>JHIST_EMP_FK</td>
<td>R</td>
<td>EMPLOYEE_ID</td>
<td>1</td>
</tr>
<tr>
<td>JHIST_DEPT_FK</td>
<td>R</td>
<td>DEPARTMENT_ID</td>
<td>1</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Create the Statement

Create a primary key constraint on the emp table’s employee_id column.

Table altered.
Tell Me / Show Me

Create the Statement

Create a primary key on the dept table’s department_id column

Table altered.
Tell Me / Show Me

Fix the Code

Problem:
Add a foreign constraint between DEPT and EMP so that only valid departments can be entered in the EMP table, but make sure you can delete any row from the DEPT table.

Statement: ALTER TABLE e_mp
CREATE CONSTRAINT FOREIGN KEY (dept_id) REFS dept(deptid) on del cascade

Table altered.
Tell Me / Show Me

Create the Code

Test the foreign key constraint you just created by following the examples on this slide.

Examine the number of rows in the EMP table.

Remove the details of department 10 from the dept table.

Now count emps again and check if there are fewer employees as well.
Tell Me / Show Me

Write the Query

Problem:
Produce a report that returns the last name, salary, department number and average salary of all the departments where salary is greater than the average salary.

Tables Used:
Employees, Departments

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
<th>SALAVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>13000</td>
<td>20</td>
<td>9500</td>
</tr>
<tr>
<td>Mourgos</td>
<td>5800</td>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>60</td>
<td>6400</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>80</td>
<td>10033</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>80</td>
<td>10033</td>
</tr>
<tr>
<td>King</td>
<td>24000</td>
<td>90</td>
<td>19333</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>110</td>
<td>10150</td>
</tr>
</tbody>
</table>
Tell Me / Show Me

Write the Code

Problem:
Create a view named V2 that returns the highest salary, lowest salary, average salary and department name.

Tables Used:
emp, dept
**Tell Me / Show Me**

**Write the Code**

**Problem:**
Create a view named Dept_Managers_view that returns a listing of department names along with the manager initial and surname for that department.

Test the view by returning all the rows from it.

Make sure no rows can be updated through the view.
Try to run an UPDATE statement against the view.

**Tables Used:**
Employees, departments

<table>
<thead>
<tr>
<th>DEPT_NAME</th>
<th>MGR_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive</td>
<td>S.King</td>
</tr>
<tr>
<td>IT</td>
<td>A.Hunold</td>
</tr>
<tr>
<td>Shipping</td>
<td>K.Mourgos</td>
</tr>
<tr>
<td>Sales</td>
<td>E.Zlotkey</td>
</tr>
<tr>
<td>Administration</td>
<td>J.Whalen</td>
</tr>
<tr>
<td>Marketing</td>
<td>M.Hartstein</td>
</tr>
<tr>
<td>Accounting</td>
<td>S.Higgins</td>
</tr>
</tbody>
</table>

ORA-01733: virtual column not allowed here
Tell Me / Show Me

Fix the Code

Problem:
The following statement contains errors.
Fix them and run the code to get the displayed result.

Statement:
DROP V3 views;

View dropped.
Tell Me / Show Me

Create a Sequence and Fix the Code

Problem:
Create a sequence named ct_seq with all the default values. Run the statements and fix the error. Correct the statement to return the subsequent number.

Code:
CREATE SEQUENCE ct_seq;

SELECT ct_seq.currval
FROM dual;

ORA-08002: sequence CT_SEQ.CURRVAL is not yet defined in this session
Tell Me / Show Me

Fix the Code

Problem:
Look at the insert statement and fix the error.

Code:

```
INSERT emp
(employee_id,first_name,last_name,email,phone_number,
hire_date,job_id,salary,commission_pct,manager_id,department_id)
VALUES
(currval.ct_seq,'Kaare','Hansen','KHANSEN','44965 832123',sysdate
,'Manager',6500,null,100,10)
```

ORA-00984: column not allowed here
Tell Me / Show Me

Fix the Code

Problem:
Fix the error in the SQL statement to create the index as shown in the screenshot.

Code:
CREATE INX emp indx FOR TABLE emp(employee_id DESC, UPPR(SUBST(firstname,1.1 ||" "||astname)
Tell Me / Show Me

Write the Code

Problem:
Write the SQL statement to list all the user tables which contains the name PRIV.

Tables Used:
dictionary
Tell Me / Show Me

Fix the Code

Problem:
Give select access to public on the EMP table, and verify the grant by running this query. The query contains errors that you must fix before you can run the select statement.

Code:
GRANT SELECT ON emp TO PUBLIC
SELECT *
FROM   usr_tab_privs
WHERE tablename = “emp”
Tell Me / Show Me

Write the Code

Problem:
Complete the following query using regular expressions to return only the numbers from the following string:
'Oracle Academy9547d6905%&^ db apex'.

Statement:
SELECT REGEXP_REPLACE('Oracle Academy9547d6905%&^ db apex'
YOUR CODE HERE) regexprreplace
FROM DUAL

REGEXPREPLACE
95476905
Write the Code

Problem:
Amend the previous query using regular expressions to return the number of digits from the following string: 'Oracle Academy9547d6905%&^ db'

Statement:
SELECT REGEXP_REPLACE('Oracle Academy9547d6905%&^ db apex' \ YOUR CODE HERE) regexprreplace FROM DUAL
Tell Me / Show Me

Write the Code

Problem:
Amend the query again to return only the non-numeric characters.

Statement:
SELECT REGEXP_REPLACE('Oracle Academy9547d6905%&^ db apex' YOUR CODE HERE) regexprreplace
FROM DUAL
Problem:
Using Oracle proprietary joins, construct a statement that returns all the employee_id's joined to all the department_names.

Tables Used:
Employees, departments
Tell Me / Show Me

Write the Code

Problem:
Still using Oracle Joins, correct the previous statement so that it returns only the name of the department that the employee actually works in.

Tables Used:
Employees, departments
Tell Me / Show Me

Write the Code

Problem:
Still using Oracle Joins, construct a query that lists the employees last name, the department name, the salary and the country name of all employees.

Tables Used:
Employees, departments, locations and countries
Tell Me / Show Me

Write the Code

Problem:
Still using Oracle join syntax, alter the previous query so that it also includes the employee record of the employee with no department_id, ‘Grant’.

Tables Used:
Employees, departments, locations and countries
Summary

Objectives Summarized
In this lesson you have learned to:

• Create an advanced query to produce specified data
• Modify an advanced query to produce specified data
Summary

Practice Guide
The link for the lesson practice guide can be found in the course resources in Section 0.