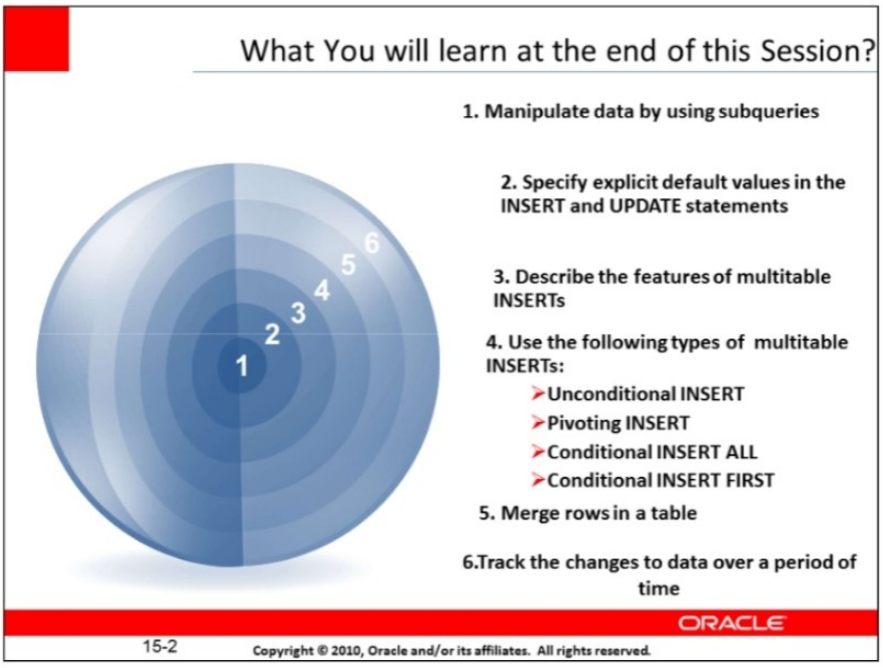
Lesson 15

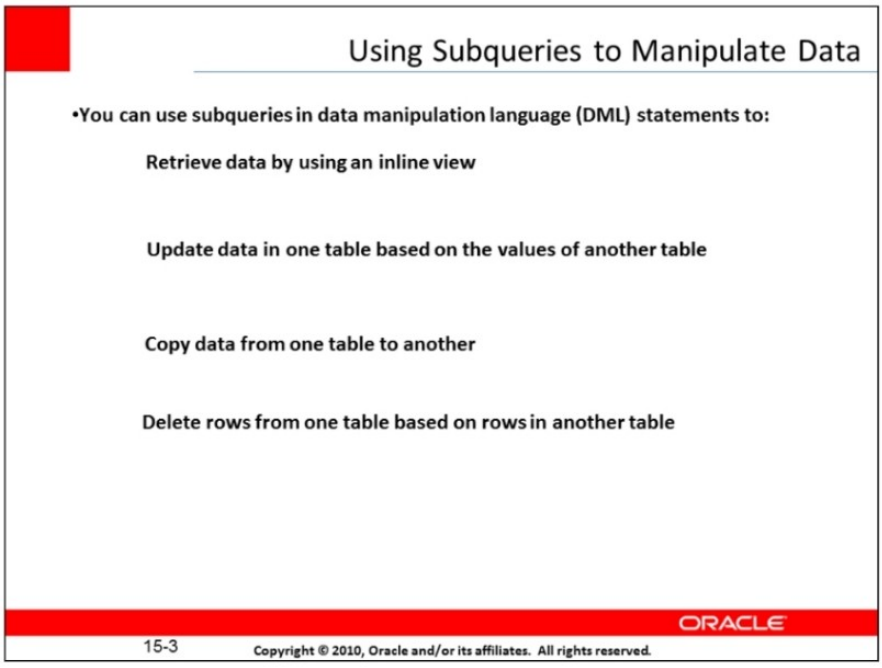
Manipulating Large Data Sets

ORACLE



What You will learn at the end of this Session?

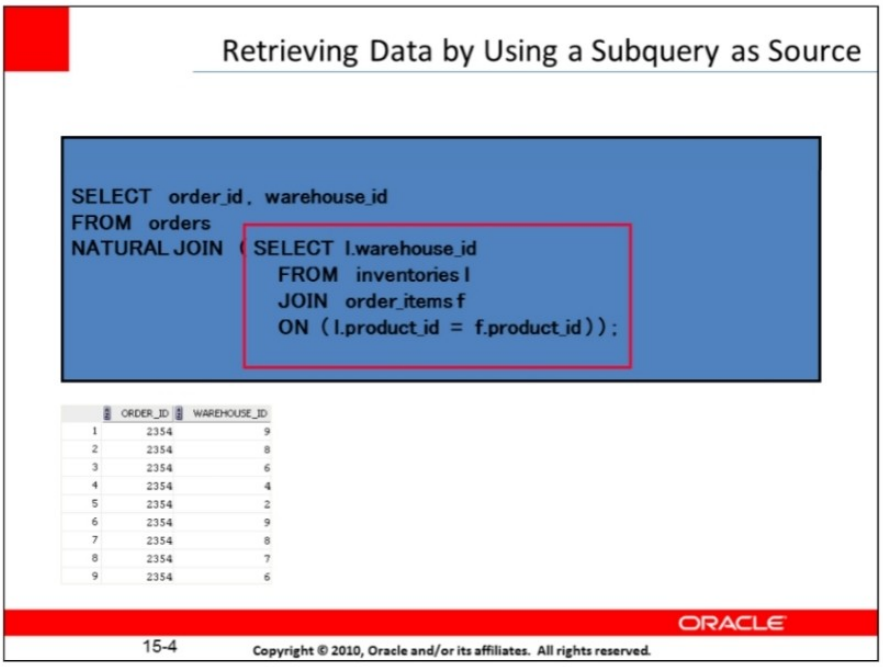
In this lesson, you learn how to manipulate data in the Oracle database by using subqueries. You learn how to use the `DEFAULT` keyword in `INSERT` and `UPDATE` statements to identify a default column value. You also learn about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.



Using Subqueries to Manipulate Data

Subqueries can be used to retrieve data from a table that you can use as input to an `INSERT` into a different table. In this way, you can easily copy large volumes of data from one table to another with one single `SELECT` statement. Similarly, you can use subqueries to do mass updates and deletes by using them in the `WHERE` clause of the `UPDATE` and `DELETE` statements. You can also use subqueries in the `FROM` clause of a `SELECT` statement. This is called an inline view.

Note: You learned how to update and delete rows based on another table in the course titled *Oracle Database: SQL Fundamentals I*.



Retrieving Data by Using a Subquery as Source

You can use a subquery in the `FROM` clause of a `SELECT` statement, which is very similar to how views are used. A subquery in the `FROM` clause of a `SELECT` statement is also called an *inline* view. A subquery in the `FROM` clause of a `SELECT` statement defines a data source for that particular `SELECT` statement, and only that `SELECT` statement. As with a database view, the `SELECT` statement in the subquery can be as simple or as complex as you like.

When a database view is created, the associated `SELECT` statement is stored in the data dictionary. In situations where you do not have the necessary privileges to create database views, or when you would like to test the suitability of a `SELECT` statement to become a view, you can use an inline view.

Retrieving Data by Using a Subquery as Source (continued)

You can display the same output as in the example in the slide by performing the following two steps:

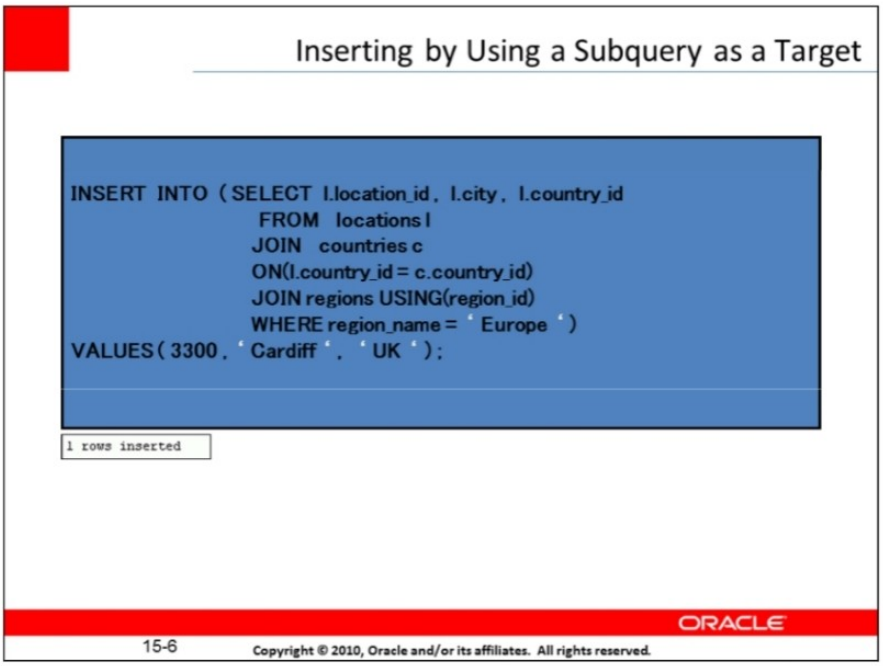
1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe';
```

2. Join the `EUROPEAN_CITIES` view with the `DEPARTMENTS` table:

```
SELECT department_name, city
FROM   departments
NATURAL JOIN european_cities;
```

Note: You learned how to create database views in the course titled *Oracle Database: SQL Fundamentals I*.



Inserting by Using a Subquery as a Target

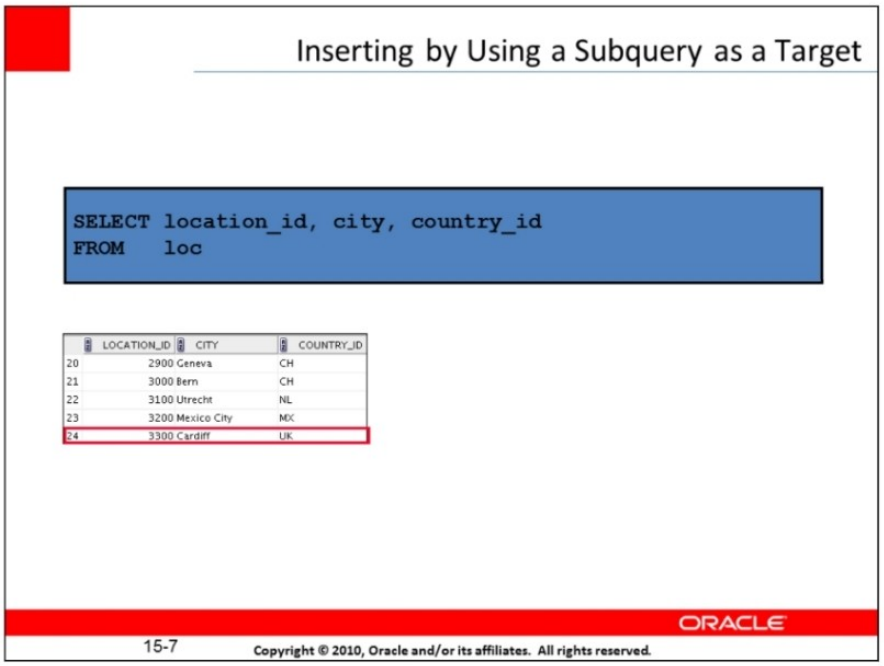
You can use a subquery in place of the table name in the `INTO` clause of the `INSERT` statement. The `SELECT` list of this subquery must have the same number of columns as the column list of the `VALUES` clause. Any rules on the columns of the base table must be followed in order for the `INSERT` statement to work successfully. For example, you cannot put in a duplicate location ID or leave out a value for a mandatory `NOT NULL` column.

This use of subqueries helps you avoid having to create a view just for performing an `INSERT`.

The example in the slide uses a subquery in the place of `LOC` to create a record for a new European city.

Note: You can also perform the `INSERT` operation on the `EUROPEAN_CITIES` view by using the following code:

```
INSERT INTO european_cities
VALUES (3300,'Cardiff','UK');
```

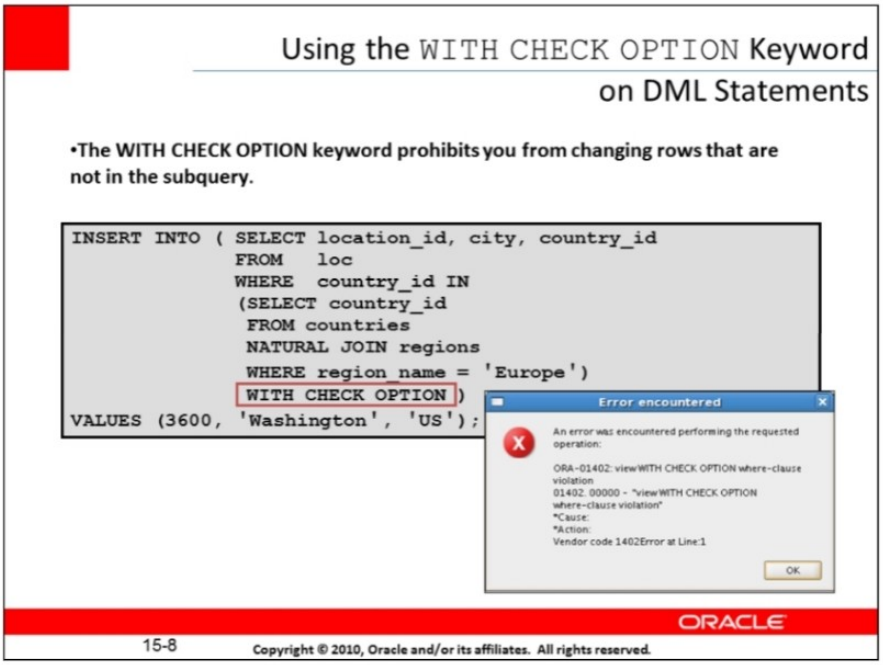
Inserting by Using a Subquery as a Target (continued)

The example in the slide shows that the insert via the inline view created a new record in the base table `LOC`.

The following example shows the results of the subquery that was used to identify the table for the `INSERT` statement.

```
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe'
```

	LOCATION_ID	CITY	COUNTRY_ID
6	2700	Munich	DE
7	2900	Geneva	CH
8	3000	Bern	CH
9	3100	Utrecht	NL
10	3300	Cardiff	UK



Using the WITH CHECK OPTION Keyword on DML Statements

Specify the WITH CHECK OPTION keyword to indicate that if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, no changes that produce rows that are not included in the subquery are permitted to that table.

The example in the slide shows how to use an inline view with WITH CHECK OPTION. The INSERT statement prevents the creation of records in the LOC table for a city that is not in Europe.

The following example executes successfully because of the changes in the VALUES list.

```
INSERT INTO (SELECT location_id, city, country_id
             FROM   loc
             WHERE  country_id IN
             (SELECT country_id
              FROM countries
              NATURAL JOIN regions
              WHERE region_name = 'Europe')
              WITH CHECK OPTION)
VALUES (3500, 'Berlin', 'DE');
```


Using the WITH CHECK OPTION Keyword on DML Statements (continued)

The use of an inline view with the WITH CHECK OPTION provides an easy method to prevent changes to the table.

To prevent the creation of a non-European city, you can also use a database view by performing the following steps:

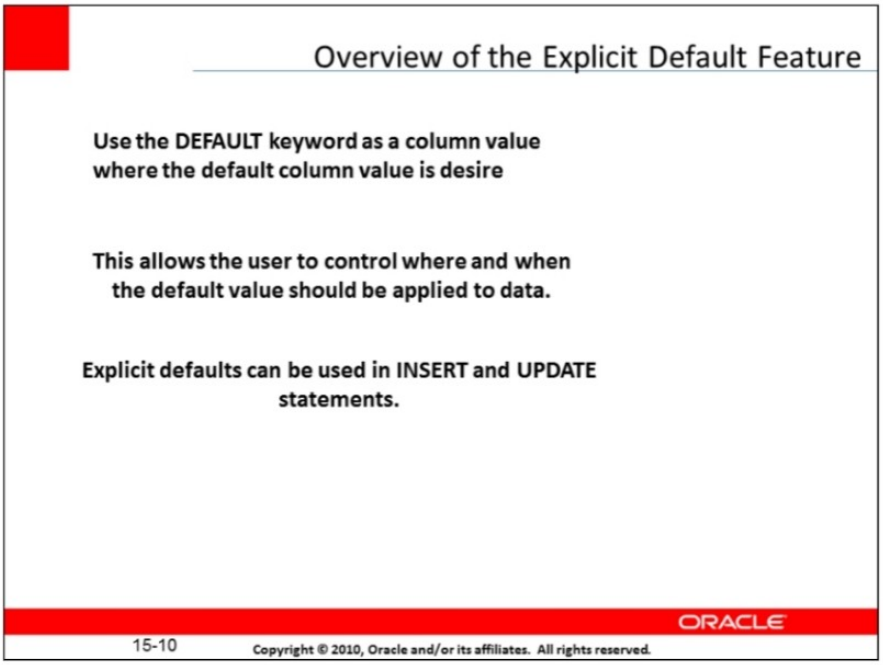
1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT location_id, city, country_id
FROM   locations
WHERE  country_id in
  (SELECT country_id
   FROM countries
   NATURAL JOIN regions
   WHERE region_name = 'Europe')
WITH CHECK OPTION;
```

2. Verify the results by inserting data:

```
INSERT INTO european_cities
VALUES (3400,'New York','US');
```

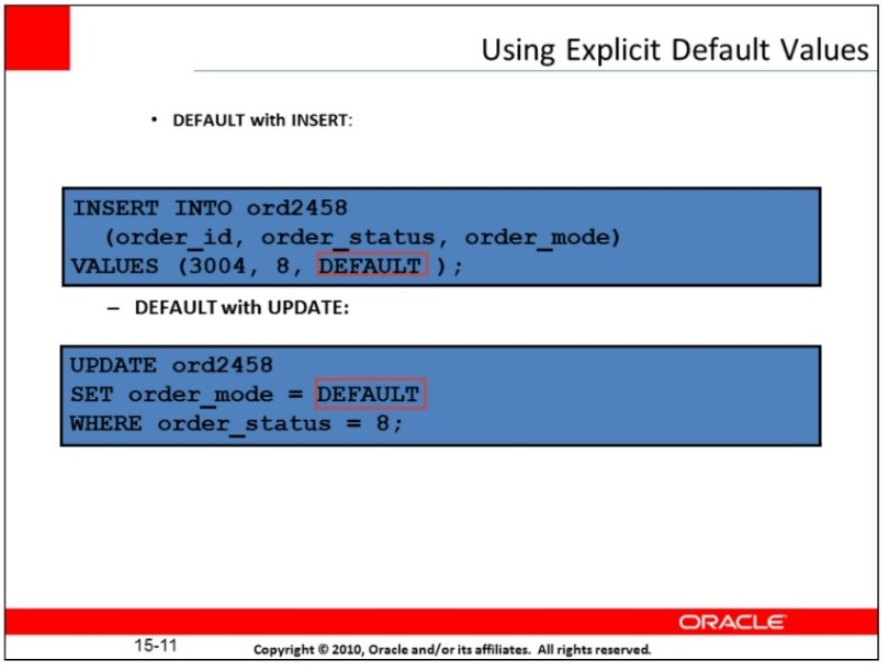
The second step produces the same error as shown in the slide.



Explicit Defaults

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

The `DEFAULT` option saves you from having to hard code the default value in your programs or querying the dictionary to find it, as was done before this feature was introduced. Hard coding the default is a problem if the default changes, because the code consequently needs changing. Accessing the dictionary is not usually done in an application; therefore, this is a very important feature.



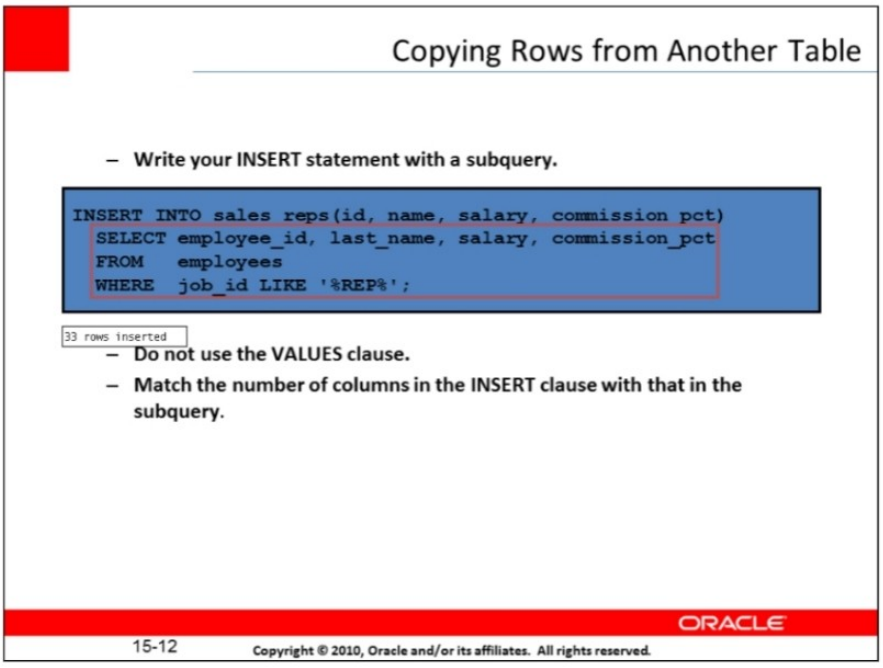
Using Explicit Default Values

Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, the Oracle server sets the column to null.

In the first example in the slide, the `INSERT` statement uses a default value for the `order_mode` column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the `UPDATE` statement to set the order_mode column to a default value for status 8. If no default value is defined for the column, it changes the value to null.

Note: When creating a table, you can specify a default value for a column. This is discussed in *SQL Fundamentals I*.



Copying Rows from Another Table

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. In place of the `VALUES` clause, you use a subquery.

Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

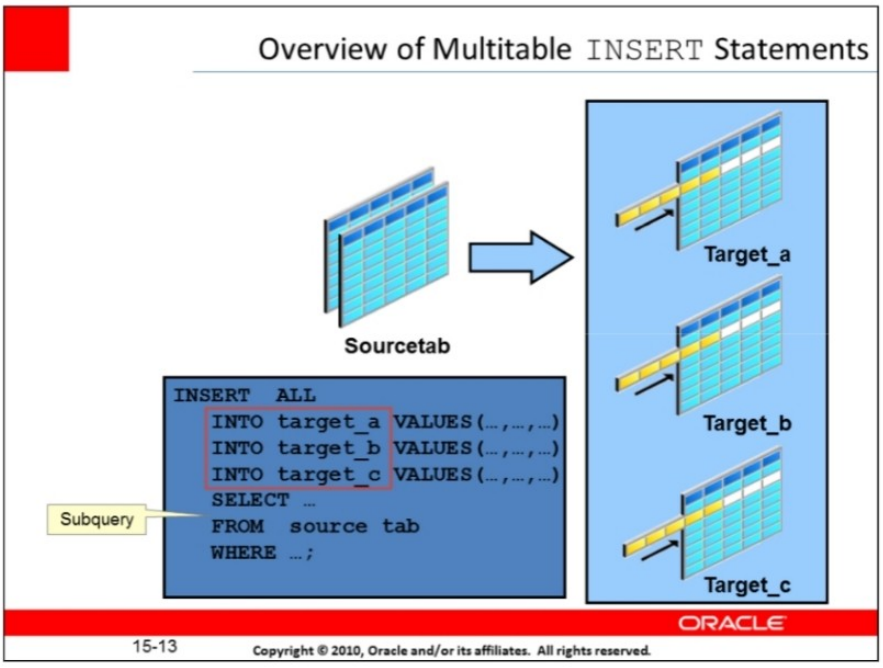
table	Is the table name
column	Is the name of the column in the table to populate
subquery	Is the subquery that returns rows into the table

The number of columns and their data types in the column list of the `INSERT` clause must match the number of values and their data types in the subquery. To create a copy of the rows of a table, use `SELECT *` in the subquery.

```
INSERT INTO EMPL3
   SELECT *
   FROM   employees;
```

Note: You use the `LOG ERRORS` clause in your DML statement to enable the DML operation to complete regardless of errors. Oracle writes the details of the error message to an error-logging table that you have created. For more information, see the *Oracle Database SQL Reference for*

10g or 11g database.



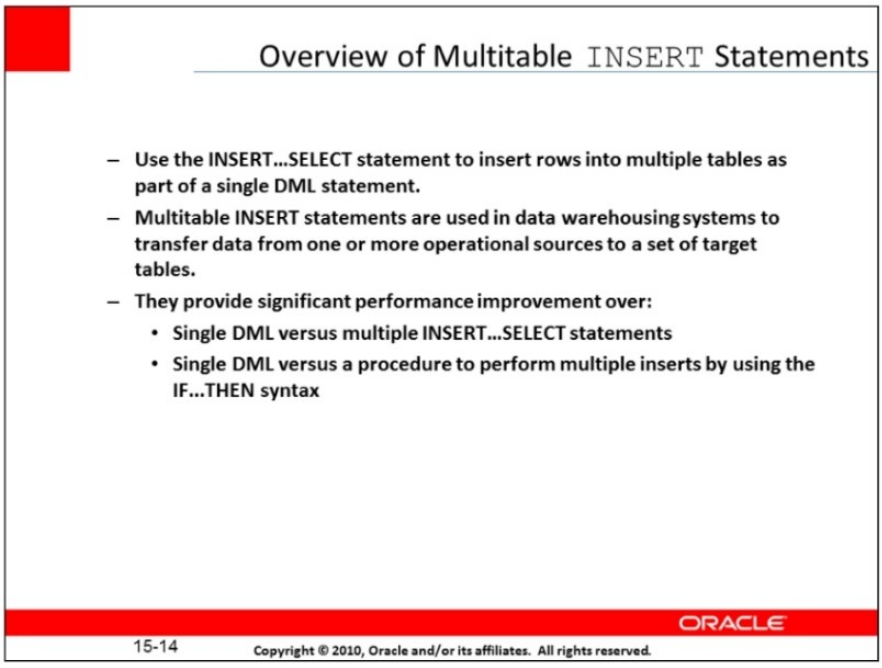
Overview of Multitable INSERT Statements

In a multitable INSERT statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Multitable INSERT statements are useful in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.

During extraction, the desired data must be identified and extracted from many different sources, such as database systems and applications. After extraction, the data must be physically transported to the target system or an intermediate system for further processing. Depending on the chosen means of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the SELECT statement.

After data is loaded into the Oracle database, data transformations can be executed using SQL operations. A multitable INSERT statement is one of the techniques for implementing SQL data transformations.

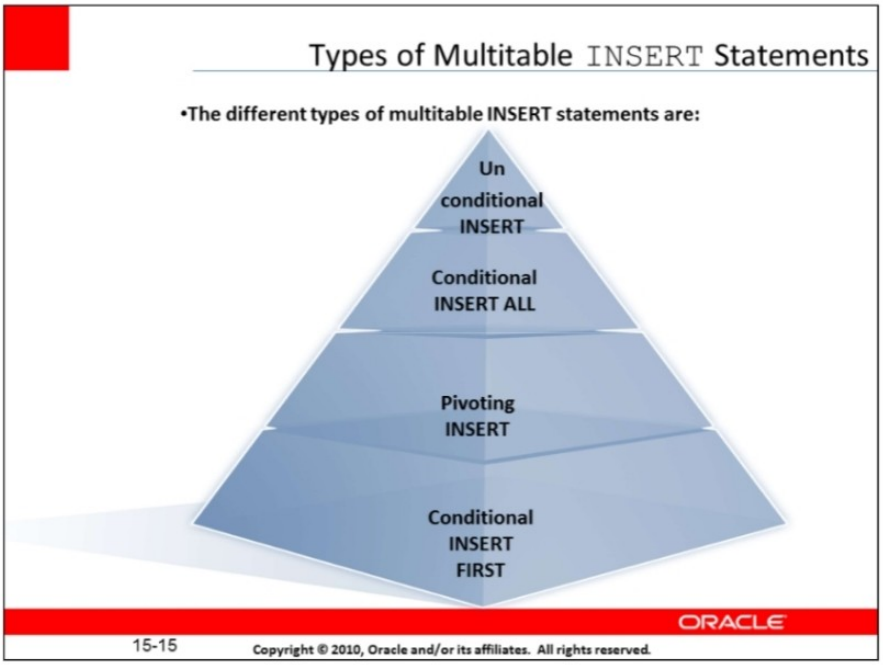


Overview of Multitable `INSERT` Statements (continued)

Multitable `INSERT` statements offer the benefits of the `INSERT . . . SELECT` statement when multiple tables are involved as targets. Without multitable `INSERT`, you had to deal with *n* independent `INSERT . . . SELECT` statements, thus processing the same source data *n* times and increasing the transformation workload *n* times.

As with the existing `INSERT . . . SELECT` statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

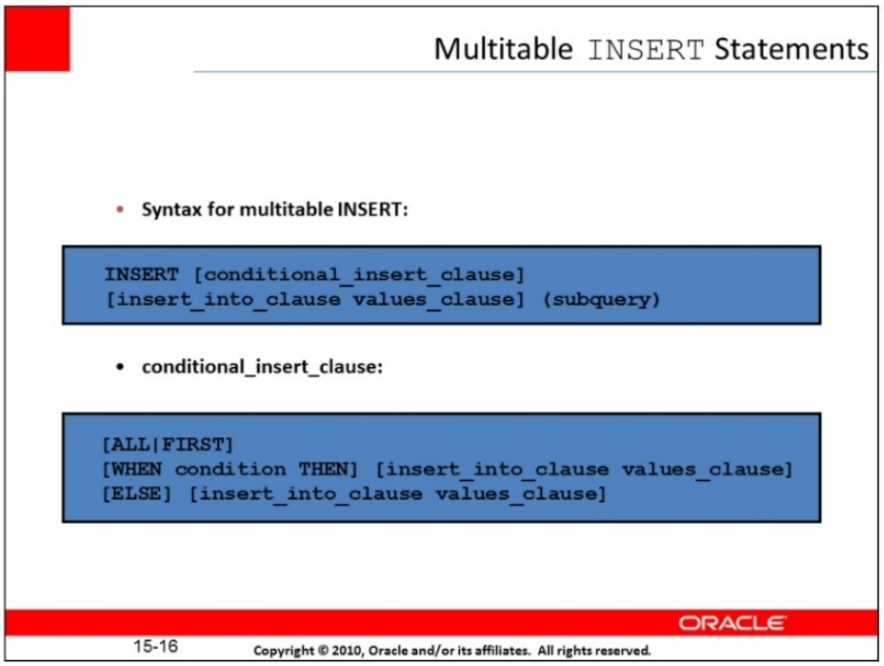
Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for a more relational database table environment. To alternatively implement this functionality, you were required to write multiple `INSERT` statements.



Types of Multitable `INSERT` Statements

You use different clauses to indicate the type of `INSERT` to be executed. The types of multitable `INSERT` statements are:

- **Unconditional `INSERT`:** For each row returned by the subquery, a row is inserted into each of the target tables.
- **Conditional `INSERT ALL`:** For each row returned by the subquery, a row is inserted into each target table if the specified condition is met.
- **Pivoting `INSERT`:** This is a special case of the unconditional `INSERT ALL`.
- **Conditional `INSERT FIRST`:** For each row returned by the subquery, a row is inserted into the very first target table in which the condition is met.



Multitable INSERT Statements

The slide displays the generic format for multitable INSERT statements.

Unconditional INSERT: ALL into_clause

Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable INSERT. The Oracle server executes each insert_into_clause once for each row returned by the subquery.

Conditional INSERT: conditional_insert_clause

Specify the conditional_insert_clause to perform a conditional multitable INSERT. The Oracle server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable INSERT statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL

If you specify ALL, the Oracle server 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 Oracle server executes the corresponding INTO clause list.

Multitable INSERT Statements (continued)

Conditional INSERT: FIRST

If you specify FIRST, the Oracle server evaluates each WHEN clause in the order in which it appears in the statement. If the first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

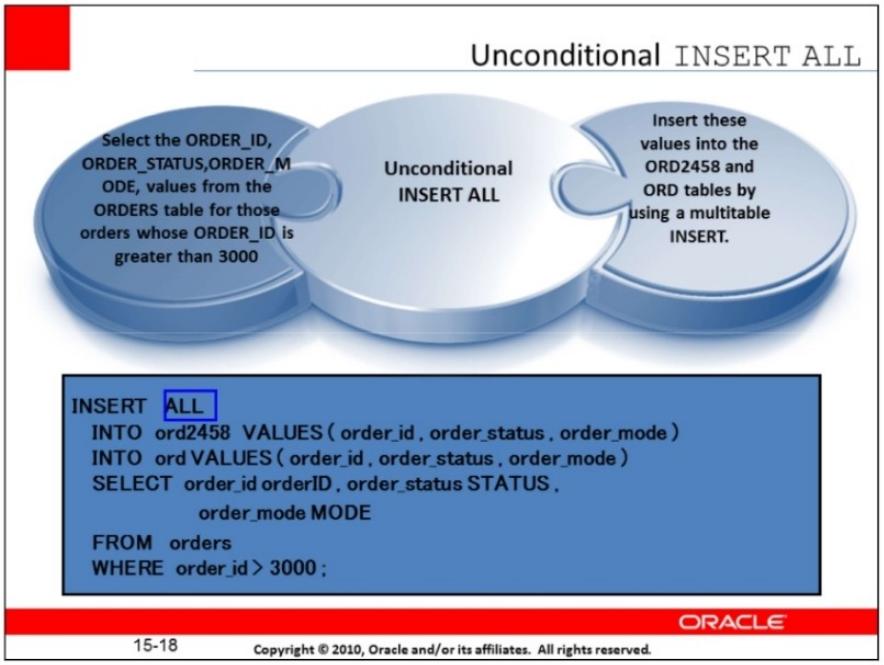
Conditional INSERT: ELSE Clause

For a given row, if no WHEN clause evaluates to true:

- If you have specified an ELSE clause, the Oracle server executes the INTO clause list associated with the ELSE clause
- If you did not specify an ELSE clause, the Oracle server takes no action for that row

Restrictions on Multitable INSERT Statements

- You can perform multitable INSERT statements only on tables, and not on views or materialized views.
- You cannot perform a multitable INSERT on a remote table.
- You cannot specify a table collection expression when performing a multitable INSERT.
- In a multitable INSERT, all insert_into_clauses cannot combine to specify more than 999 target columns.



Unconditional `INSERT ALL`

The example in the slide inserts rows into both the `ORD2458` and the ORD tables.

The `SELECT` statement retrieves the details of order ID, order status, order mode of those orders whose order ID is greater than 300 from the `ORDERS` table. The details of the order ID, order status, order mode are inserted into the `ORD2458` table. The details of order ID, order status, order mode are inserted into the `ORD` table.

This `INSERT` statement is referred to as an unconditional `INSERT` because no further restriction is applied to the rows that are retrieved by the `SELECT` statement. All the rows retrieved by the `SELECT` statement are inserted into the two tables: `ORD2458` and `ORD`. The `VALUES` clause in the `INSERT` statements specifies the columns from the `SELECT` statement that must be inserted into each of the tables. Each row returned by the `SELECT` statement results in two insertions: one for the `ORD2458` table and one for the `ORD` table.

Unconditional INSERT ALL (continued)

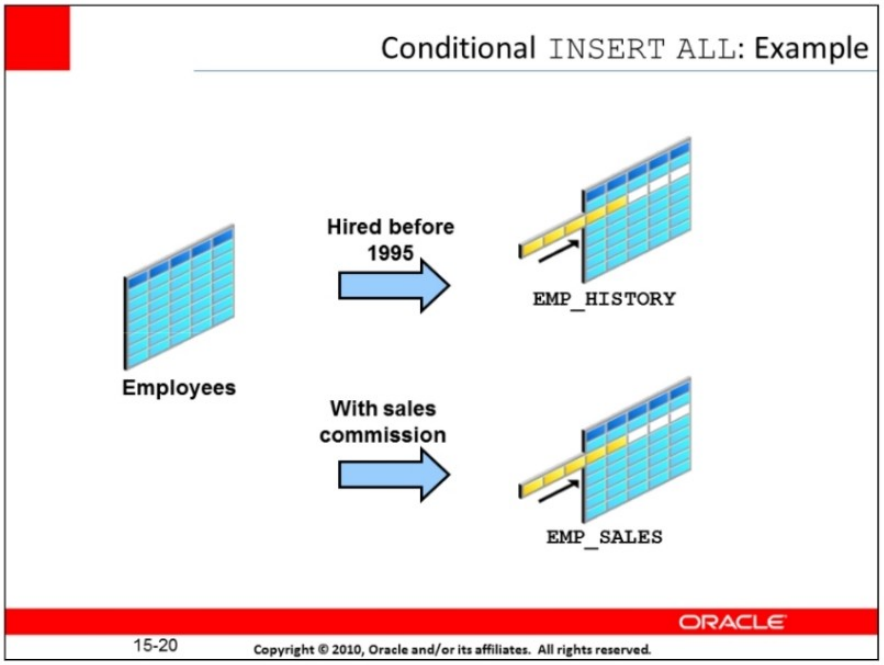
A total of 12 rows were selected:

```
SELECT COUNT(*) total_in_sal FROM sal_history;
```

	TOTAL_IN_SAL
1	6

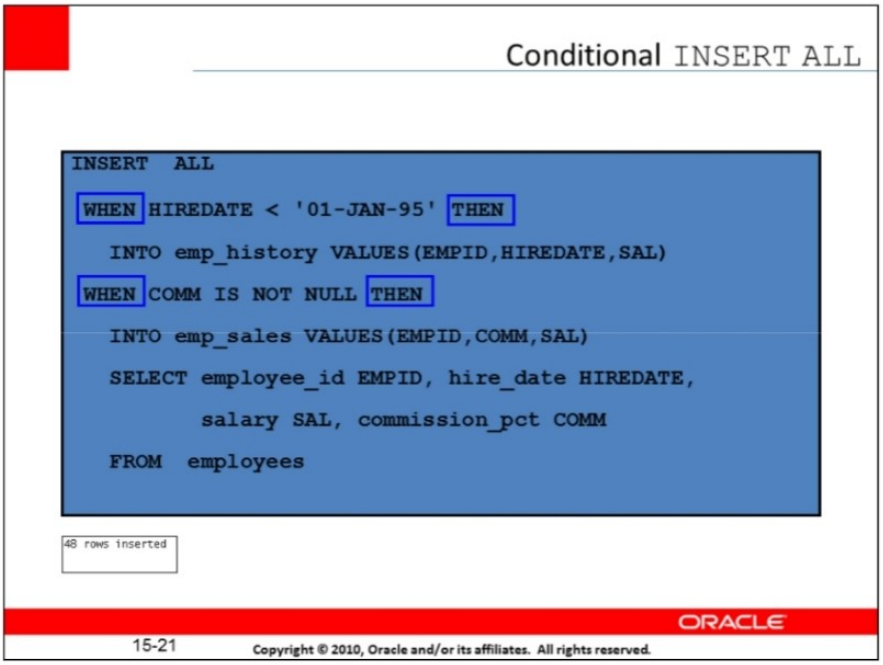
```
SELECT COUNT(*) total_in_mgr FROM mgr_history;
```

	TOTAL_IN_MGR
1	6



Conditional INSERT ALL: Example

For all employees in the employees tables, if the employee was hired before 1995, insert that employee record into the employee history. If the employee earns a sales commission, insert the record information into the EMP_SALES table. The SQL statement is shown on the next page.



Conditional INSERT ALL

The example in the slide is similar to the example in the previous slide because it inserts rows into both the EMP_HISTORY and the EMP_SALES tables. The SELECT statement retrieves details such as employee ID, hire date, salary, and commission percentage for all employees from the EMPLOYEES table. Details such as employee ID, hire date, and salary are inserted into the EMP_HISTORY table. Details such as employee ID, commission percentage, and salary are inserted into the EMP_SALES table.

This INSERT statement is referred to as a conditional INSERT ALL because a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the hire date was prior to 1995 are inserted in the EMP_HISTORY table. Similarly, only those rows where the value of commission percentage is not null are inserted in the EMP_SALES table.

```
SELECT count(*) FROM emp_history;
```

	COUNT(*)
1	13

```
SELECT count(*) FROM emp_sales;
```

	COUNT(*)
1	35

Conditional `INSERT ALL` **(continued)**

You can also optionally use the `ELSE` clause with the `INSERT ALL` statement.

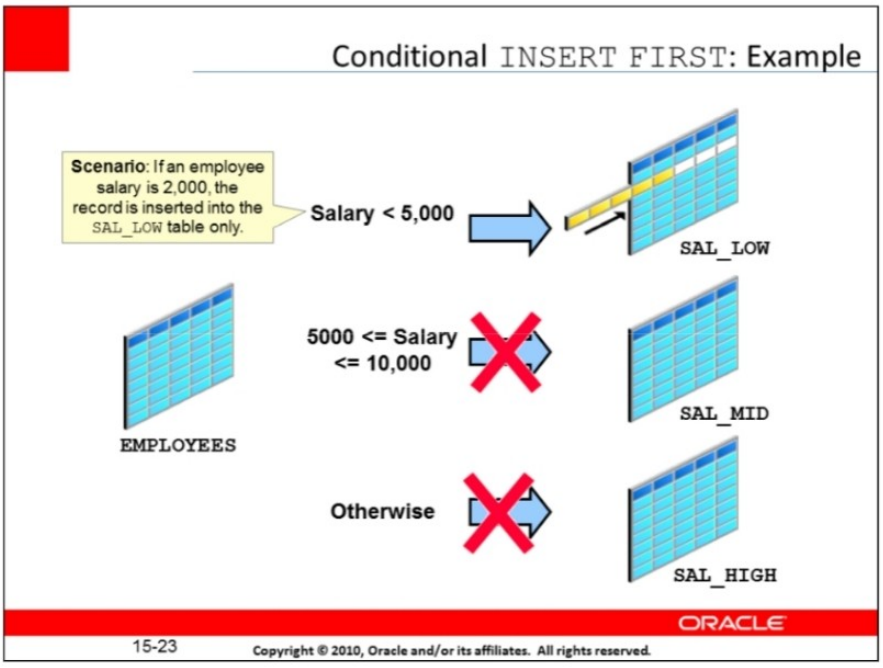
Example:

```
INSERT ALL
WHEN job_id IN
(select job_id FROM jobs WHERE job_title LIKE '%Manager%')
THEN
INTO managers2(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
WHEN SALARY>10000 THEN
INTO richpeople(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
ELSE
INTO poorpeople VALUES (last_name,job_id,SALARY)
SELECT * FROM employees;
```

Result:

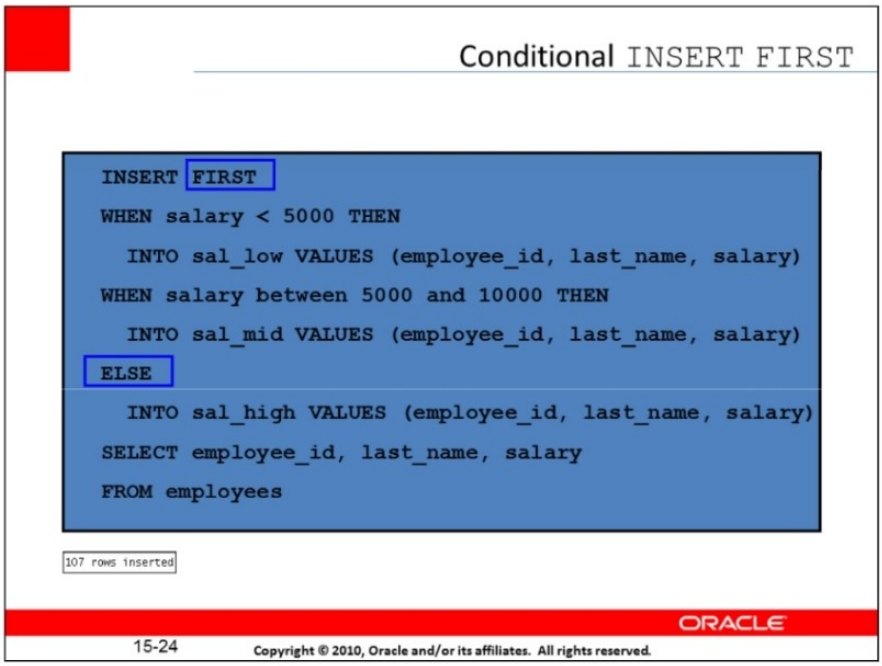
```
116 rows inserted
```





Conditional INSERT FIRST: Example

For all employees in the EMPLOYEES table, insert the employee information into the first target table that meets the condition. In the example, if an employee has a salary of 2,000, the record is inserted into the SAL_LOW table only. The SQL statement is shown on the next page.



Conditional INSERT FIRST

The SELECT statement retrieves details such as employee ID, last name, and salary for every employee in the EMPLOYEES table. For each employee record, it is inserted into the very first target table that meets the condition.

This INSERT statement is referred to as a conditional INSERT FIRST. The WHEN salary < 5000 condition is evaluated first. If this first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and inserts the record into the SAL_LOW table. It skips subsequent WHEN clauses for this row.

If the row does not satisfy the first WHEN condition (WHEN salary < 5000), the next condition (WHEN salary between 5000 and 10000) is evaluated. If this condition evaluates to true, the record is inserted into the SAL_MID table, and the last condition is skipped.

If neither the first condition (WHEN salary < 5000) nor the second condition (WHEN salary between 5000 and 10000) is evaluated to true, the Oracle server executes the corresponding INTO clause for the ELSE clause.

Conditional INSERT FIRST (continued)

A total of 20 rows were inserted:

```
SELECT count(*) low FROM sal_low;
```

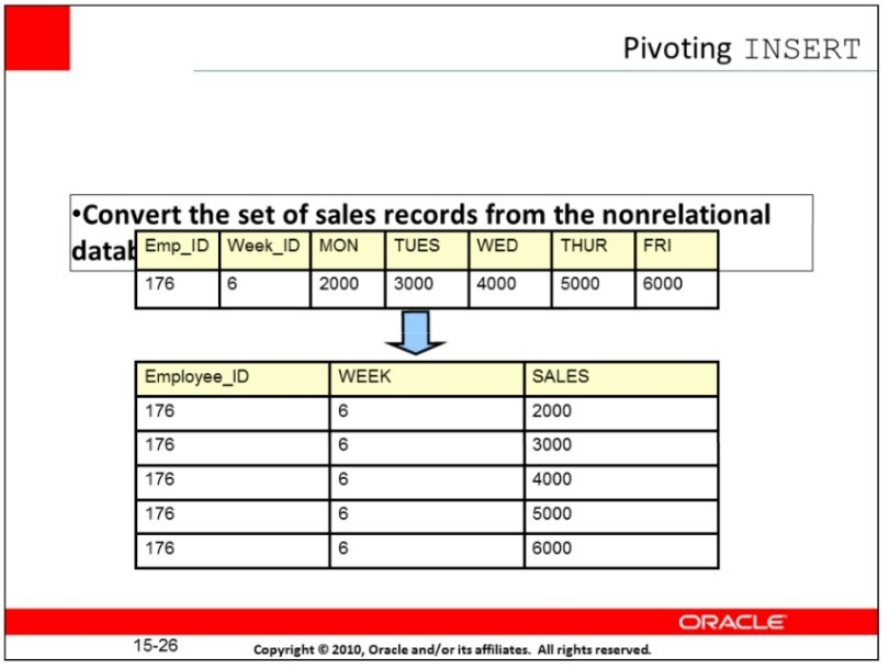
	LOW
1	49

```
SELECT count(*) mid FROM sal_mid;
```

	MID
1	43

```
SELECT count(*) high FROM sal_high;
```

	HIGH
1	15



Pivoting `INSERT`

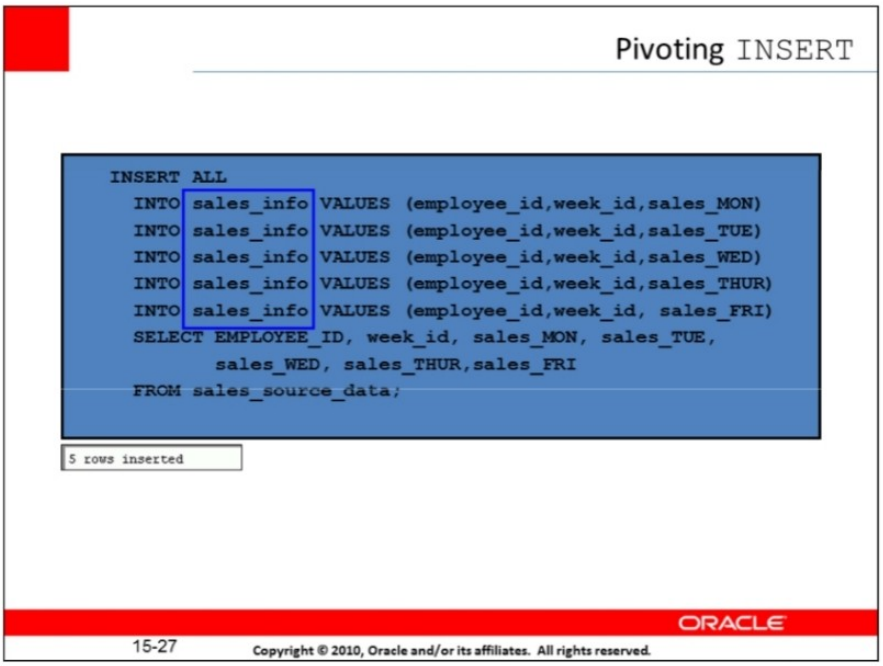
Pivoting is an operation in which you must build a transformation such that each record from any input stream, such as a nonrelational database table, must be converted into multiple records for a more relational database table environment.

Suppose you receive a set of sales records from a nonrelational database table:
`SALES_SOURCE_DATA`, in the following format:
`EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED, SALES_THUR, SALES_FRI`

You want to store these records in the `SALES_INFO` table in a more typical relational format:
`EMPLOYEE_ID, WEEK, SALES`

To solve this problem, you must build a transformation such that each record from the original nonrelational database table, `SALES_SOURCE_DATA`, is converted into five records for the data warehouse's `SALES_INFO` table. This operation is commonly referred to as *pivoting*.

The solution to this problem is shown on the next page.



Pivoting `INSERT` (continued)

In the example in the slide, the sales data is received from the nonrelational database table `SALES_SOURCE_DATA`, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

```
DESC SALES_SOURCE_DATA
```

```
Name                          Null     Type
----------------------------- -------- -------------
EMPLOYEE_ID                            NUMBER(6)
WEEK_ID                                NUMBER(2)
SALES_MON                              NUMBER(8,2)
SALES_TUE                              NUMBER(8,2)
SALES_WED                              NUMBER(8,2)
SALES_THUR                             NUMBER(8,2)
SALES_FRI                              NUMBER(8,2)
```


Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	178	6	1750	2200	1500	1500	3000

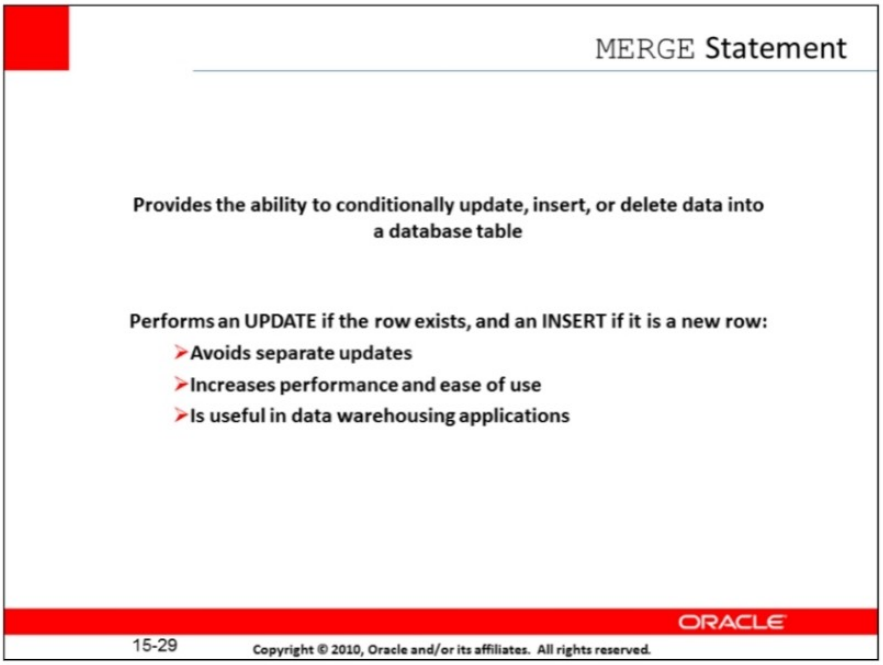
```
DESC SALES_INFO
```

```
Name                          Null     Type
----------------------------- -------- ------------
EMPLOYEE_ID                            NUMBER(6)
WEEK                                   NUMBER(2)
SALES                                  NUMBER(8,2)
```

```
SELECT * FROM sales_info;
```

	EMPLOYEE_ID	WEEK	SALES
1	178	6	1750
2	178	6	2200
3	178	6	1500
4	178	6	1500
5	178	6	3000

Observe in the preceding example that by using a pivoting INSERT, one row from the SALES_SOURCE_DATA table is converted into five records for the relational table, SALES_INFO.



MERGE Statement

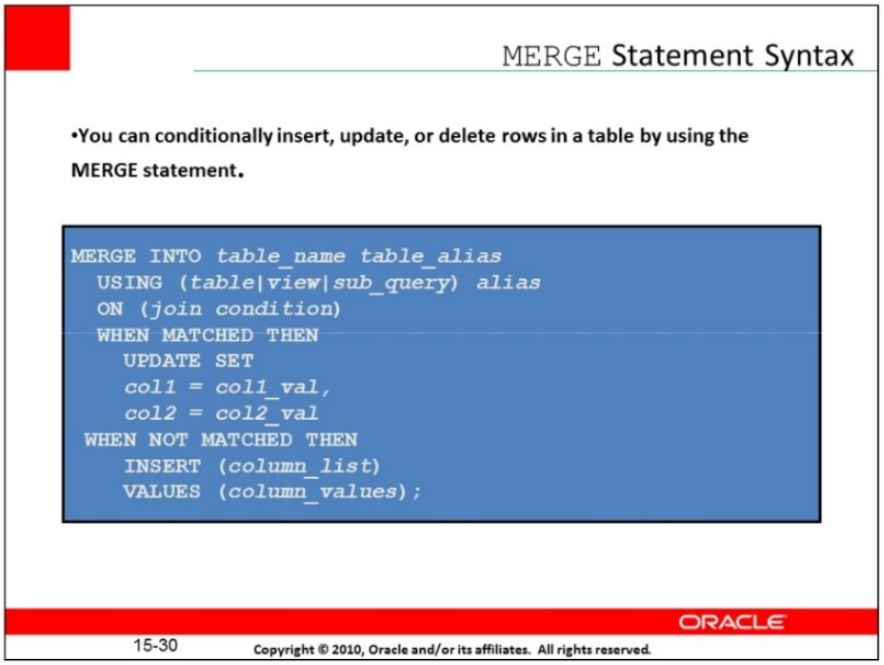
The Oracle server supports the MERGE statement for INSERT, UPDATE, and DELETE operations. Using this statement, you can update, insert, or delete a row conditionally into a table, thus avoiding multiple DML statements. The decision whether to update, insert, or delete into the target table is based on a condition in the ON clause.

You must have the INSERT and UPDATE object privileges on the target table and the SELECT object privilege on the source table. To specify the DELETE clause of merge_update_clause, you must also have the DELETE object privilege on the target table.

The MERGE statement is deterministic. You cannot update the same row of the target table multiple times in the same MERGE statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The MERGE statement, however, is easy to use and more simply expressed as a single SQL statement.

The MERGE statement is suitable in a number of data warehousing applications. For example, in a data warehousing application, you may need to work with data coming from multiple sources, some of which may be duplicates. With the MERGE statement, you can conditionally add or modify rows.



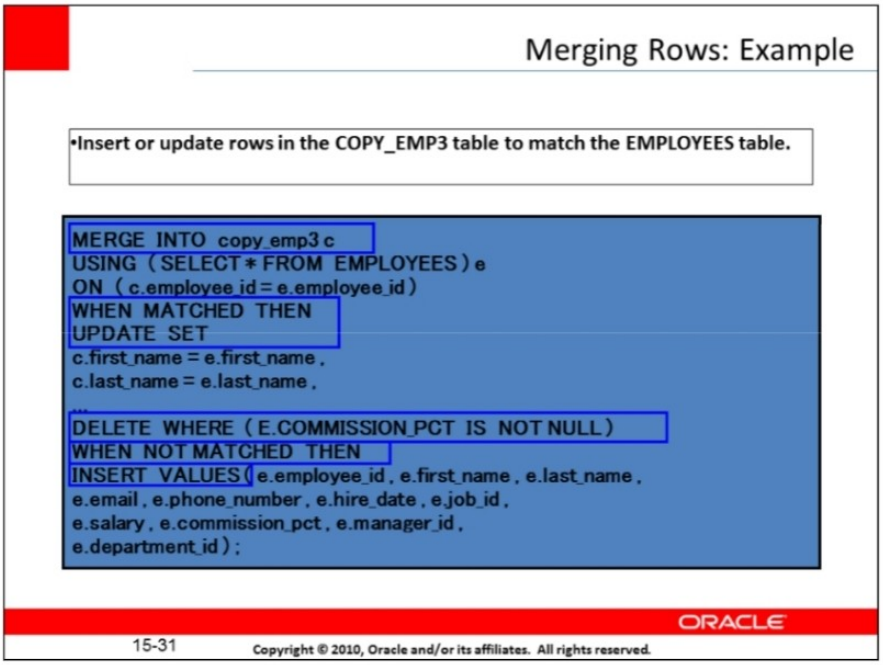
Merging Rows

You can update existing rows, and insert new rows conditionally by using the `MERGE` statement. Using the `MERGE` statement, you can delete obsolete rows at the same time as you update rows in a table. To do this, you include a `DELETE` clause with its own `WHERE` clause in the syntax of the `MERGE` statement.

In the syntax:

`INTO` clause	Specifies the target table you are updating or inserting into
`USING` clause	Identifies the source of the data to be updated or inserted; can be a table, view, or subquery
`ON` clause	The condition on which the `MERGE` operation either updates or inserts
`WHEN MATCHED | WHEN NOT MATCHED`	Instructs the server how to respond to the results of the join condition

Note: For more information, see *Oracle Database SQL Reference for 10g or 11g database*.



Merging Rows: Example

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
c.email = e.email,
c.phone_number = e.phone_number,
c.hire_date = e.hire_date,
c.job_id = e.job_id,
c.salary = e.salary*2,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
```


```
        e.department_id);
```


Merging Rows: Example (continued)

The COPY_EMP3 table is created by using the following code:

```
CREATE TABLE COPY_EMP3 AS SELECT * FROM EMPLOYEES
WHERE SALARY<10000;
```

Then query the COPY_EMP3 table.

```
SELECT employee_id, salary, commission_pct FROM COPY_EMP3;
```

	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	198	5200	(null)
2	199	5200	(null)
3	200	8800	(null)
4	202	12000	(null)
5	203	13000	(null)

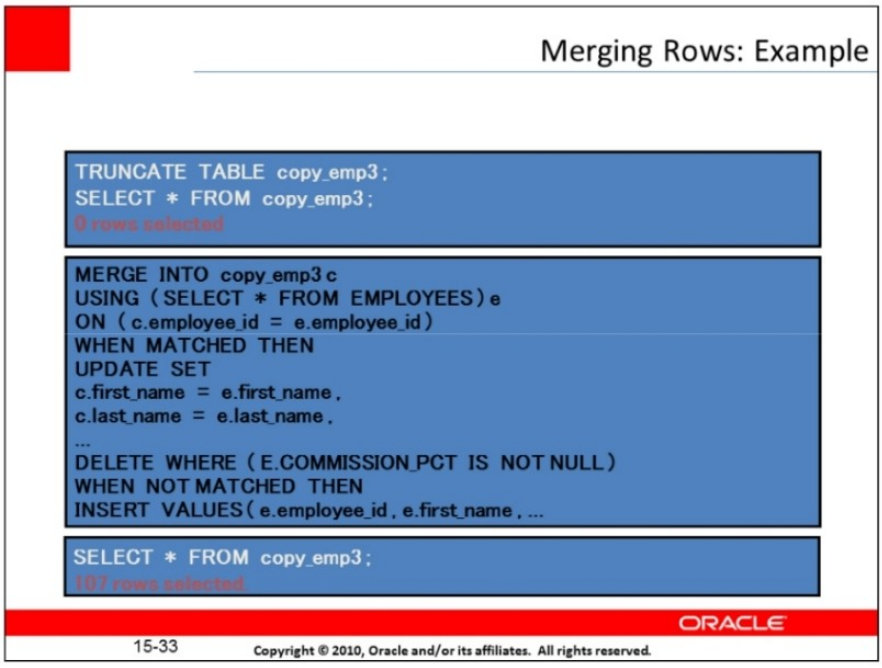
...

64	197	6000	(null)
65	162	10500	0.25
66	146	13500	0.3
67	150	10000	0.3

...

Observe that there are some employees with SALARY < 10000 and there are two employees with COMMISSION_PCT.

The example in the slide matches the EMPLOYEE_ID in the COPY_EMP3 table to the EMPLOYEE_ID in the EMPLOYEES table. If a match is found, the row in the COPY_EMP3 table is updated to match the row in the EMPLOYEES table and the salary of the employee is doubled. The records of the two employees with values in the COMMISSION_PCT column are deleted. If the match is not found, rows are inserted into the COPY_EMP3 table.

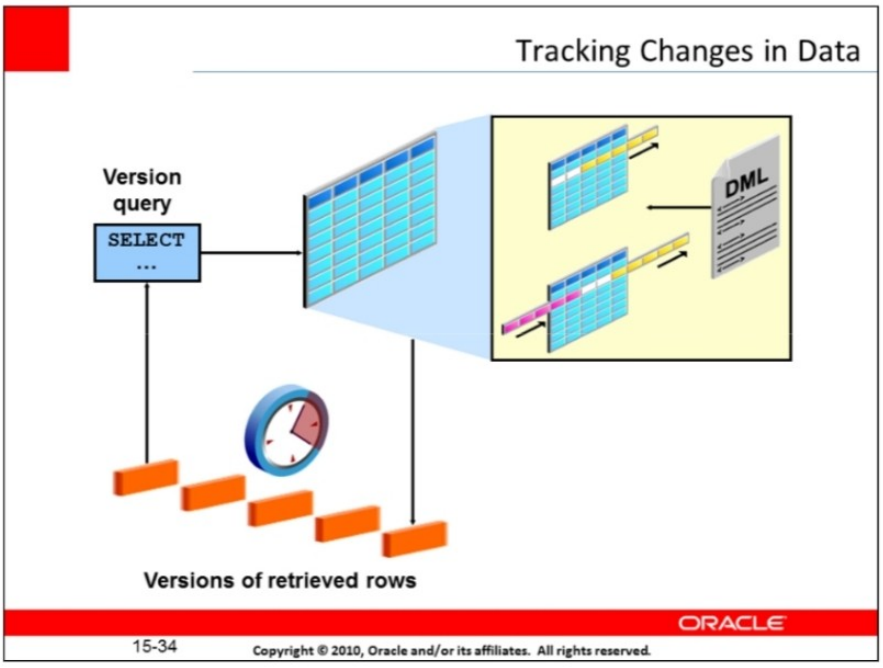


Merging Rows: Example (continued)

The examples in the slide show that the `COPY_EMP3` table is empty. The `c.employee_id = e.employee_id` condition is evaluated. The condition returns false—there are no matches. The logic falls into the `WHEN NOT MATCHED` clause, and the `MERGE` command inserts the rows of the `EMPLOYEES` table into the `COPY_EMP3` table. This means that the `COPY_EMP3` table now has exactly the same data as in the `EMPLOYEES` table.

```
SELECT employee_id, salary, commission_pct from copy_emp3;
```

	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	144	2500	(null)
2	143	2600	(null)
3	202	6000	(null)
4	141	3500	(null)
5	174	11000	0.3
...			
15	149	10500	0.2
16	206	8300	(null)
17	176	8600	0.2
18	124	5800	(null)
19	205	12000	(null)
20	178	7000	0.15



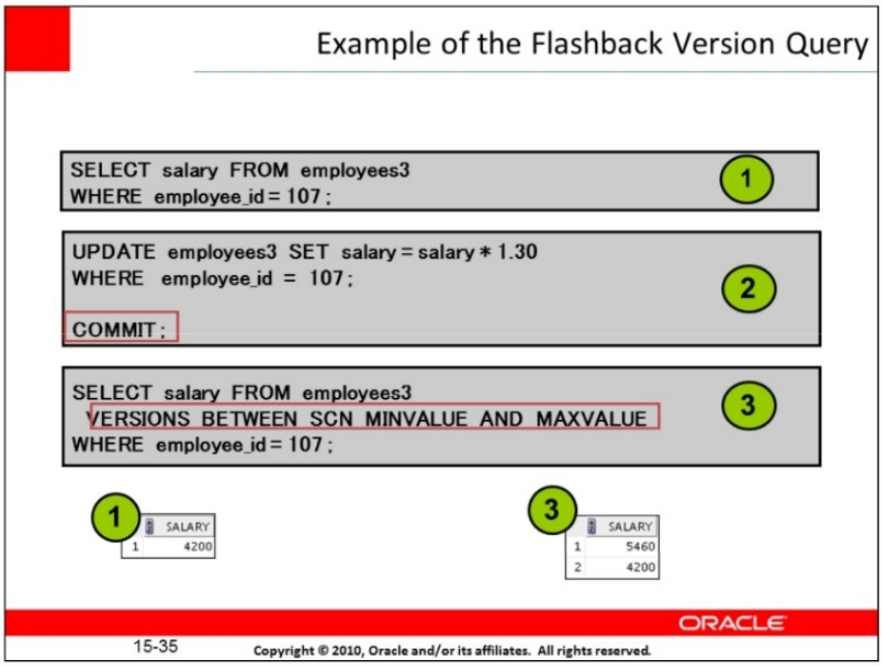
Tracking Changes in Data

You may discover that somehow data in a table has been inappropriately changed. To research this, you can use multiple flashback queries to view row data at specific points in time. More efficiently, you can use the Flashback Version Query feature to view all changes to a row over a period of time. This feature enables you to append a `VERSIONS` clause to a `SELECT` statement that specifies a system change number (SCN) or the time stamp range within which you want to view changes to row values. The query also can return associated metadata, such as the transaction responsible for the change.

Further, after you identify an erroneous transaction, you can use the Flashback Transaction Query feature to identify other changes that were done by the transaction. You then have the option of using the Flashback Table feature to restore the table to a state before the changes were made.

You can use a query on a table with a `VERSIONS` clause to produce all the versions of all the rows that exist or ever existed between the time the query was issued and the `undo_retention` seconds before the current time. `undo_retention` is an initialization parameter, which is an autotuned parameter. A query that includes a `VERSIONS` clause is referred to as a version query. The results of a version query behaves as though the `WHERE` clause were applied to the versions of the rows. The version query returns versions of the rows only across transactions.

System change number (SCN): The Oracle server assigns an SCN to identify the redo records for each committed transaction.



Example of the Flashback Version Query

In the example in the slide, the salary for employee 107 is retrieved (1). The salary for employee 107 is increased by 30 percent and this change is committed (2). The different versions of salary are displayed (3).

The `VERSIONS` clause does not change the plan of the query. For example, if you run a query on a table that uses the index access method, the same query on the same table with a `VERSIONS` clause continues to use the index access method. The versions of the rows returned by the version query are versions of the rows across transactions. The `VERSIONS` clause has no effect on the transactional behavior of a query. This means that a query on a table with a `VERSIONS` clause still inherits the query environment of the ongoing transaction.

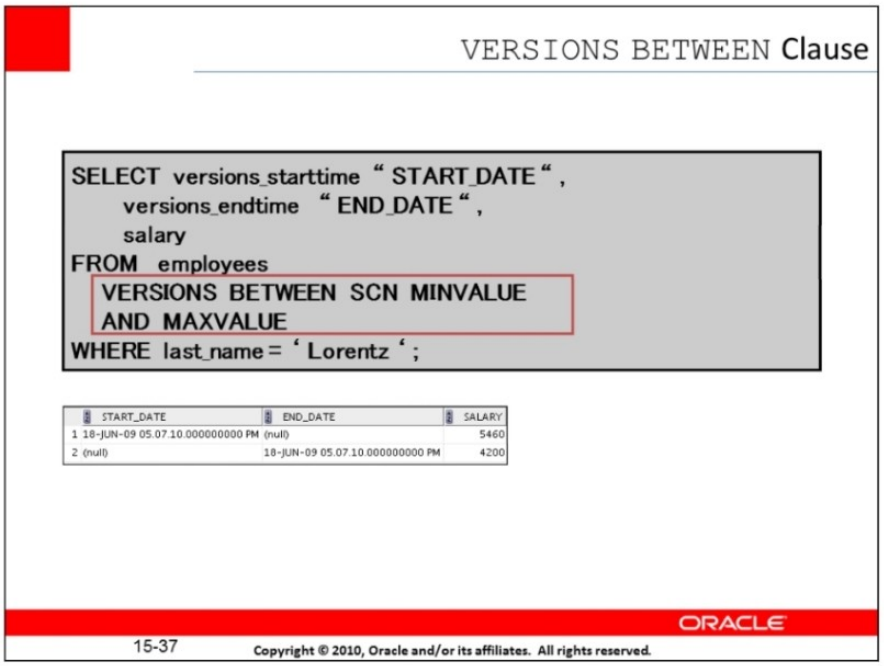
The default `VERSIONS` clause can be specified as `VERSIONS BETWEEN {SCN|TIMESTAMP} MINVALUE AND MAXVALUE`.

The `VERSIONS` clause is a SQL extension only for queries. You can have DML and DDL operations that use a `VERSIONS` clause within subqueries. The row version query retrieves all the committed versions of the selected rows. Changes made by the current active transaction are not returned. The version query retrieves all incarnations of the rows. This essentially means that versions returned include deleted and subsequent reinserted versions of the rows.

Example of the Flashback Version Query (continued)

The row access for a version query can be defined in one of the following two categories:

- **ROWID-based row access:** In case of `ROWID`-based access, all versions of the specified `ROWID` are returned irrespective of the row content. This essentially means that all versions of the slot in the block indicated by the `ROWID` are returned.
- **All other row access:** For all other row access, all versions of the rows are returned.

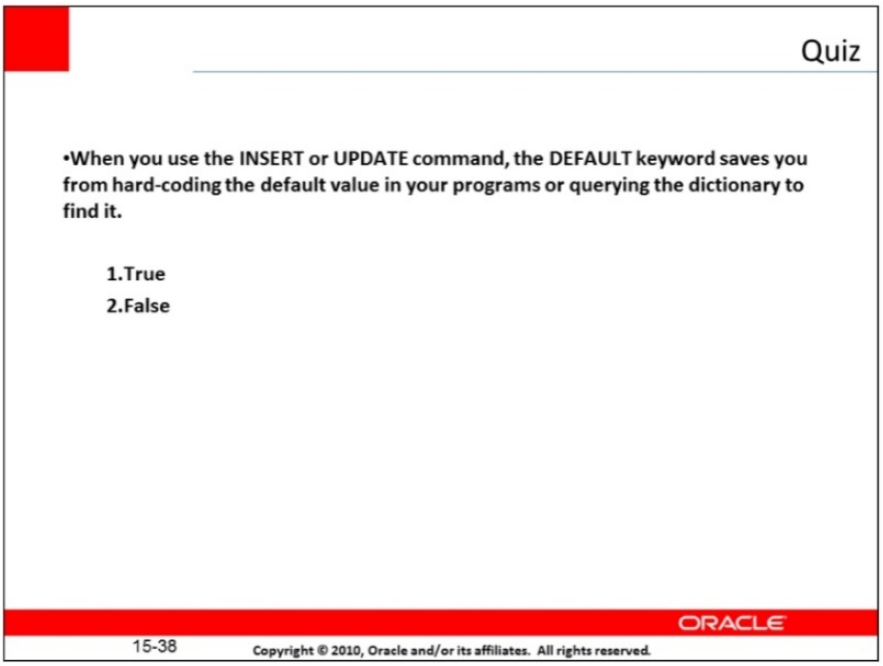


VERSIONS BETWEEN Clause

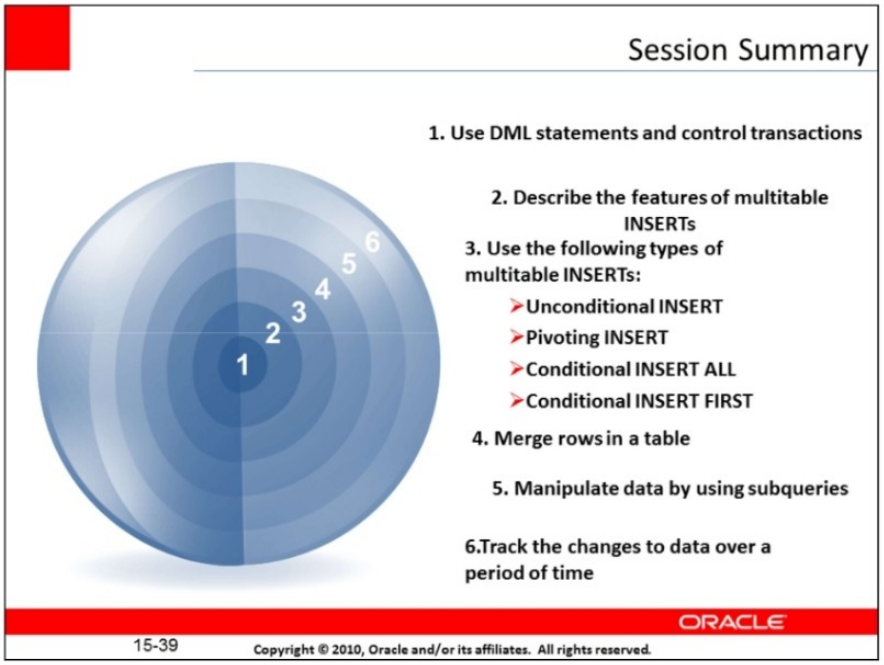
You can use the VERSIONS BETWEEN clause to retrieve all the versions of the rows that exist or have ever existed between the time the query was issued and a point back in time.

If the undo retention time is less than the lower bound time or the SCN of the BETWEEN clause, the query retrieves versions up to the undo retention time only. The time interval of the BETWEEN clause can be specified as an SCN interval or a wall-clock interval. This time interval is closed at both the lower and the upper bounds.

In the example, Lorentz's salary changes are retrieved. The NULL value for END_DATE for the first version indicates that this was the existing version at the time of the query. The NULL value for START_DATE for the last version indicates that this version was created at a time before the undo retention time.

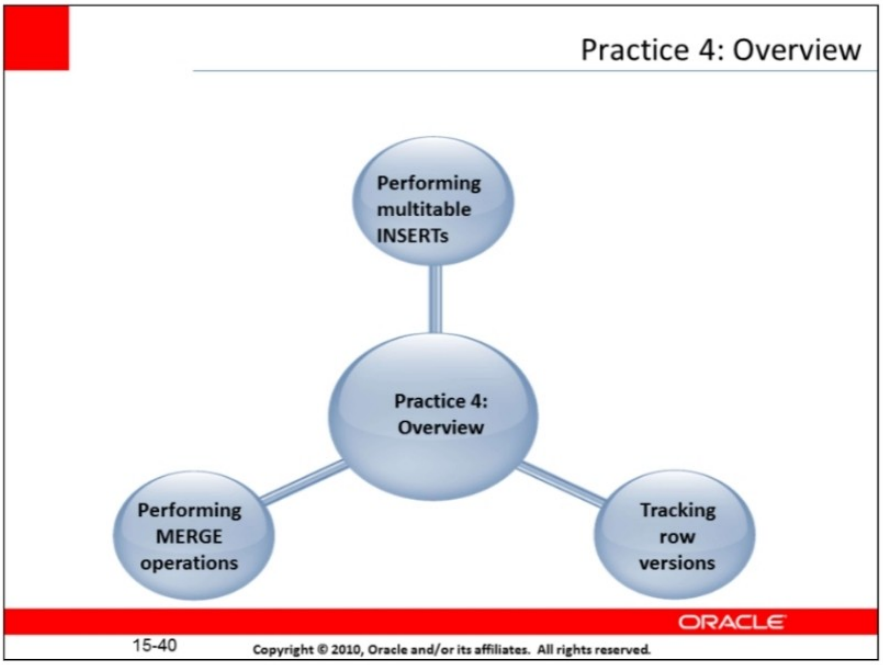

Quiz
•When you use the INSERT or UPDATE command, the DEFAULT keyword saves you from hard-coding the default value in your programs or querying the dictionary to find it.
1.True
2.False
ORACLE
15-38
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Answer: 1



Summary

In this lesson, you should have learned how to manipulate data in the Oracle database by using subqueries. You also should have learned about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.


Practice 4: Overview
Performing multitable INSERTs
Practice 4: Overview
Performing MERGE operations
Tracking row versions
ORACLE
15-40
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.