3

Manipulating Large Data Sets

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- Manipulate data using subqueries
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s
  - Unconditional `INSERT`
  - Pivoting `INSERT`
  - Conditional `ALL INSERT`
  - Conditional `FIRST INSERT`
- Merge rows in a table
- Track the changes to data over a period of time

ORACLE

Objectives

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

Using Subqueries to Manipulate Data

You can use subqueries in data manipulation language (DML) statements to:

- Copy data from one table to another
- Retrieve data from an inline view
- Update data in one table based on the values of another table
- Delete rows from one table based on rows in another table

ORACLE

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.

Copying Rows from Another Table

- Write your INSERT statement with a subquery.

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

33 rows inserted.
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause with that in the subquery.

ORACLE

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;
```

For more information, see *Oracle Database 10g SQL Reference*.

Inserting Using a Subquery as a Target

```
INSERT INTO
        (SELECT employee_id, last_name,
                email, hire_date, job_id, salary,
                department_id
         FROM   empl3
         WHERE  department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row inserted.
```


Inserting 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 employee ID or leave out a value for a mandatory `NOT NULL` column.

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

Inserting Using a Subquery as a Target

Verify the results.

```
SELECT employee_id, last_name, email, hire_date,
       job_id, salary, department_id
FROM   empl3
WHERE  department_id = 50;
```

	EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
1	120	Weiss	MWEISS	18-JUL-96	ST_MAN	8000	50
2	121	Fripp	AFRIPP	10-APR-97	ST_MAN	8200	50
3	122	Kaufling	PKAUFLIN	01-MAY-95	ST_MAN	7900	50
...							
45	199	Grant	DGRANT	13-JAN-00	SH_CLERK	2600	50
46	99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

Inserting Using a Subquery as a Target (continued)

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

Retrieving Data with a Subquery as Source

```
SELECT  a.last_name, a.salary,
        a.department_id, b.salavg
FROM    employees a JOIN (SELECT   department_id,
                          AVG(salary) salavg
                          FROM     employees
                          GROUP BY department_id) b
ON a.department_id = b.department_id
AND     a.salary > b.salavg;
```

	LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
1	King	24000	90	19333.33333333333333333333333333333333333333
2	Hunold	9000	60	5760
3	Ernst	6000	60	5760
4	Greenberg	12000	100	8600
5	Faviet	9000	100	8600
6	Raphaely	11000	30	4150

...

ORACLE

Retrieving Data 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. The example in the slide displays employee last names, salaries, department numbers, and average salaries for all the employees who earn more than the average salary in their department. The subquery in the FROM clause is named b, and the outer query references the SALAVG column using this alias.

Updating Two Columns with a Subquery

Update the job and salary of employee 114 to match the job of employee 205 and the salary of employee 168:

```
UPDATE   empl3
SET      job_id  = (SELECT  job_id
                    FROM    employees
                    WHERE   employee_id = 205),
         salary  = (SELECT  salary
                    FROM    employees
                    WHERE   employee_id = 168)
WHERE    employee_id    =  114;
1 rows updated.
```


ORACLE

Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries.

Syntax

```
UPDATE table
SET     column  =
                  (SELECT    column
                   FROM table
                   WHERE condition)
   [ ,
        column  =
                  (SELECT    column
                  FROM table
                   WHERE condition)]
[WHERE  condition ]    ;
```

Note: If no rows are updated, a message "`0 rows updated`." is returned.

Updating Rows Based on Another Table

Use subqueries in UPDATE statements to update rows in a table based on values from another table:

```
UPDATE  empl3
SET     department_id  = (SELECT department_id
                          FROM employees
                          WHERE employee_id = 100)
WHERE   job_id         = (SELECT job_id
                          FROM employees
                          WHERE employee_id = 200);
1 rows updated.
```


Updating Rows Based on Another Table

You can use subqueries in UPDATE statements to update rows in a table. The example in the slide updates the EMPL3 table based on the values from the EMPLOYEES table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

Deleting Rows Based on Another Table

Use subqueries in DELETE statements to remove rows from a table based on values from another table:

```
DELETE FROM empl3
WHERE  department_id =
                      (SELECT department_id
                       FROM   departments
                       WHERE  department_name
                              LIKE '%Public%');

1 rows deleted.
```


ORACLE

Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees who are in a department where the department name contains the string “Public.” The subquery searches the DEPARTMENTS table to find the department number based on the department name containing the string “Public.” The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEES table based on this department number.

Using the WITH CHECK OPTION Keyword on DML Statements

- A subquery is used to identify the table and columns of the DML statement.
- The WITH CHECK OPTION keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO  (SELECT employee_id, last_name, email,
                  hire_date, job_id, salary
              FROM   empl3
              WHERE  department_id = 50
                WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
```

ERROR:

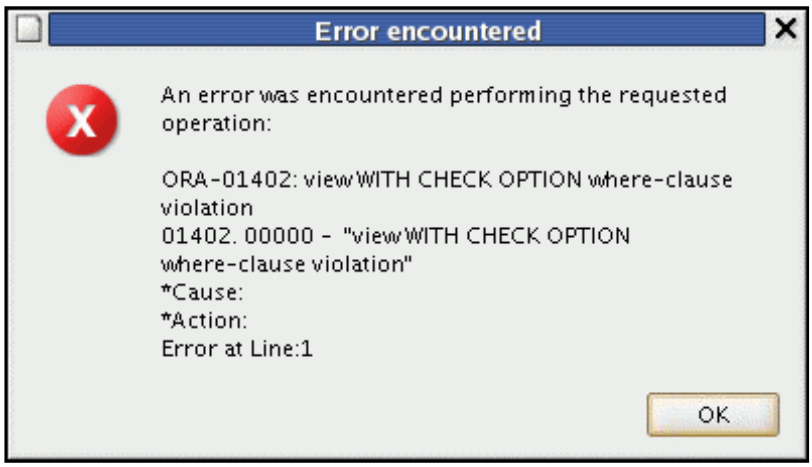
```
ORA-01402: view WITH CHECK OPTION where-clause
   violation
```

ORACLE

WITH CHECK OPTION Keyword

Specify WITH CHECK OPTION 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.

In the example shown, the WITH CHECK OPTION keyword is used. The subquery identifies rows that are in department 50, but the department ID is not in the SELECT list, and a value is not provided for it in the VALUES list. Inserting this row results in a department ID of null, which is not in the subquery.



Overview of the Explicit Default Feature

- With the explicit default feature, you can use the DEFAULT keyword as a column value where the column default is desired.
- The addition of this feature is for compliance with the SQL:1999 standard.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in INSERT and UPDATE statements.

ORACLE

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 hard coding 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 program, so this is a very important feature.

Using Explicit Default Values

- DEFAULT with INSERT:

```
INSERT INTO deptm3
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- DEFAULT with UPDATE:

```
UPDATE deptm3
SET manager_id = DEFAULT
WHERE department_id = 10;
```


ORACLE

Using Explicit Default Values

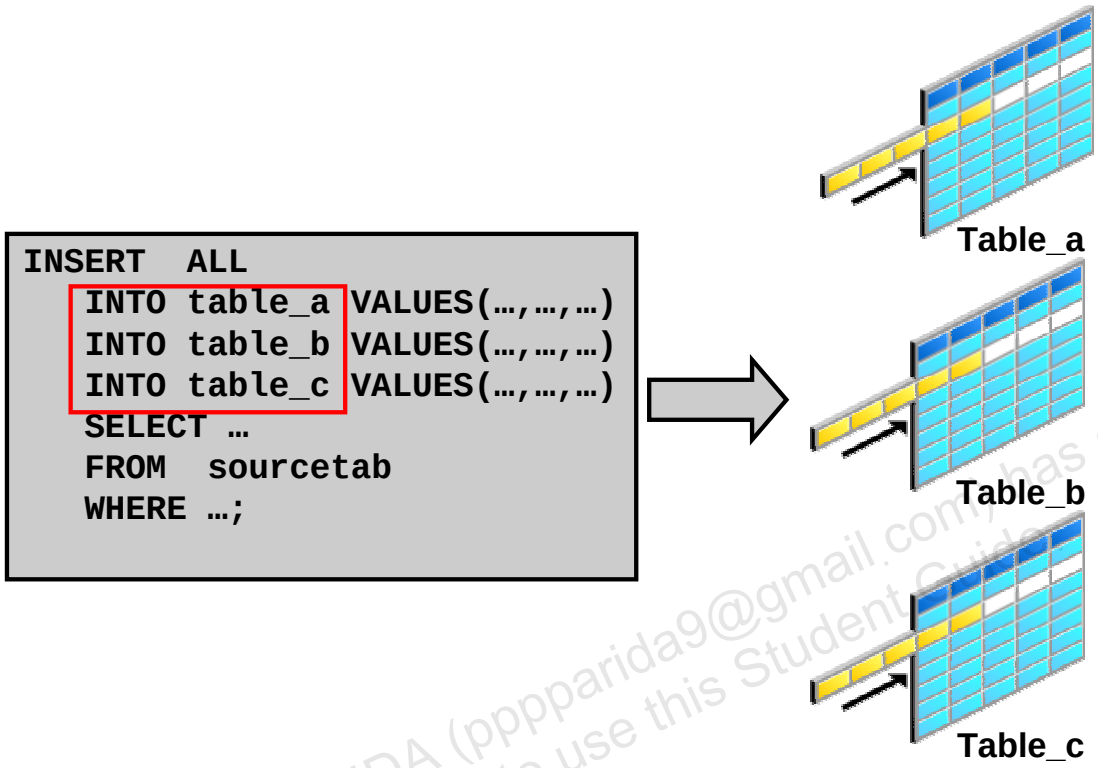
Specify DEFAULT to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, then the Oracle server sets the column to null.

In the first example in the slide, the INSERT statement uses a default value for the MANAGER_ID 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 MANAGER_ID column to a default value for department 10. If no default value is defined for the column, then 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*.

Overview of Multitable INSERT Statements



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 can play a very useful role 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

- The INSERT…SELECT statement can be used to insert rows into multiple tables as part of a single DML statement.
- Multitable INSERT statements can be used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.
- They provide significant performance improvement over:
  - Single DML versus multiple INSERT…SELECT statements
  - Single DML versus a procedure to perform multiple inserts by using the IF...THEN syntax

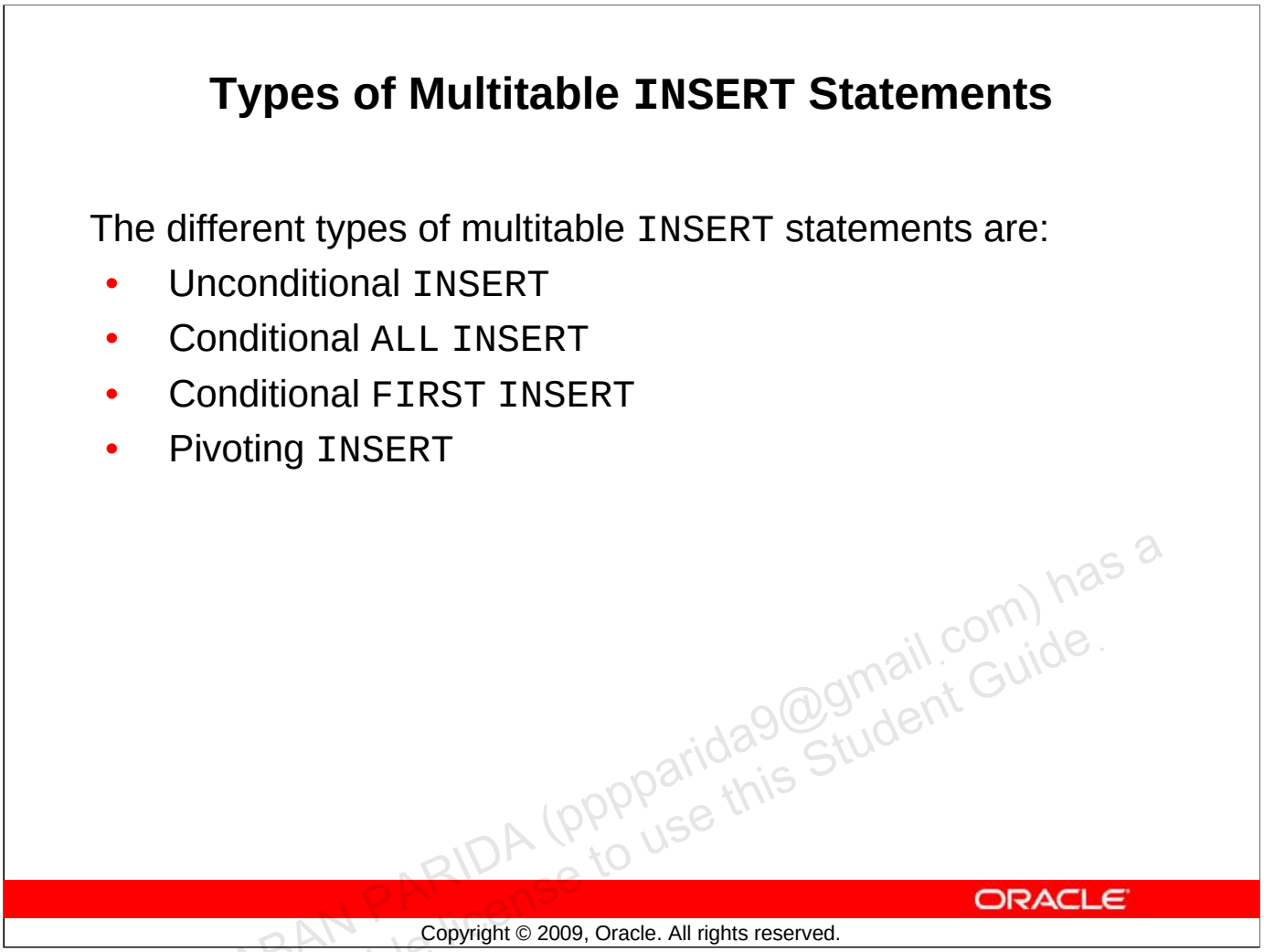
ORACLE

Overview of Multitable INSERT Statements (continued)

Multitable INSERT statements offer the benefits of the INSERT . . . SELECT statement when multiple tables are involved as targets. Using functionality before Oracle9*i* Database, 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

The types of multitable INSERT statements are:

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

You use different clauses to indicate the type of INSERT to be executed.

Multitable INSERT Statements

- Syntax:

```
INSERT [ALL] [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)
```

- conditional_insert_clause:

```
[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]
```


ORACLE

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

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- Insert these values into the SAL_HISTORY and MGR_HISTORY tables by using a multitable INSERT.

```
INSERT ALL
   INTO sal_history VALUES(EMPID,HIREDATE,SAL)
   INTO mgr_history VALUES(EMPID,MGR,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, manager_id MGR
   FROM  employees
   WHERE employee_id > 200;
12 rows inserted.
```


ORACLE

Unconditional INSERT ALL

The example in the slide inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables.

The SELECT statement retrieves the employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The employee ID, manager ID, and salary are inserted into the MGR_HISTORY 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: SAL_HISTORY and MGR_HISTORY. 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 SAL_HISTORY table and one for the MGR_HISTORY table.

The feedback 12 rows created can be interpreted to mean that a total of eight insertions were performed on the base tables, SAL_HISTORY and MGR_HISTORY.

Conditional INSERT ALL

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- If the SALARY is greater than $10,000, insert these values into the SAL_HISTORY table using a conditional multitable INSERT statement.
- If the MANAGER_ID is greater than 200, insert these values into the MGR_HISTORY table using a conditional multitable INSERT statement.

ORACLE

Conditional INSERT ALL

The problem statement for a conditional INSERT ALL statement is specified in the slide. The solution to this problem is shown on the next page.

Conditional INSERT ALL

```
INSERT ALL
  WHEN SAL > 10000 THEN
    INTO sal_history VALUES(EMPID,HIREDATE,SAL)
  WHEN MGR > 200   THEN
    INTO mgr_history VALUES(EMPID,MGR,SAL)
    SELECT employee_id EMPID,hire_date HIREDATE,
           salary SAL, manager_id MGR
    FROM   employees
    WHERE  employee_id > 200;
4 rows inserted.
```


ORACLE

Conditional INSERT ALL (continued)

The example in the slide is similar to the example in the previous slide because it inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as a conditional ALL INSERT 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 value of the SAL column is more than 10,000 are inserted in the SAL_HISTORY table, and similarly, only those rows where the value of the MGR column is more than 200 are inserted in the MGR_HISTORY table.

Observe that unlike the previous example (where eight rows were inserted into the tables), in this example, only four rows are inserted.

The feedback `4 rows created` can be interpreted to mean that a total of four inserts were performed on the base tables, SAL_HISTORY and MGR_HISTORY.

Conditional INSERT FIRST

- Select the DEPARTMENT_ID, SUM(SALARY), and MAX(HIRE_DATE)from the EMPLOYEES table.
- If the SUM(SALARY) is greater than $25,000, insert these values into the SPECIAL_SAL by using a conditional FIRST multitable INSERT.
- If the first WHEN clause evaluates to true, the subsequent WHEN clauses for this row should be skipped.
- For the rows that do not satisfy the first WHEN condition, insert into the HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIRE_DATE column using a conditional multitable INSERT.

Conditional INSERT FIRST

The problem statement for a conditional FIRST INSERT statement is specified in the slide. The solution to this problem is shown on the next page.

Conditional INSERT FIRST

```
INSERT FIRST
  WHEN SAL  > 25000          THEN
    INTO special_sal VALUES(DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES(DEPTID,HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
  ELSE
  INTO hiredate_history VALUES(DEPTID, HIREDATE)
  SELECT department_id DEPTID, SUM(salary) SAL,
         MAX(hire_date) HIREDATE
  FROM   employees
  GROUP BY department_id;
12 rows inserted.
```

ORACLE

Conditional INSERT FIRST (continued)

The example in the slide inserts rows into more than one table using a single INSERT statement. The SELECT statement retrieves the department ID, total salary, and maximum hire date for every department in the EMPLOYEES table.

This INSERT statement is referred to as a conditional FIRST INSERT because an exception is made for the departments whose total salary is more than $25,000. The condition WHEN ALL > 25000 is evaluated first. If the total salary for a department is more than $25,000, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.

For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated in the same way as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

The feedback 12 rows created can be interpreted to mean that a total of eight INSERT statements were performed on the base tables, SPECIAL_SAL, HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, and HIREDATE_HISTORY.

Pivoting INSERT

- 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
- Using a pivoting INSERT, convert the set of sales records from the nonrelational database table to relational format.

ORACLE

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.

To solve the problem mentioned in the slide, 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 problem statement for a pivoting INSERT statement is specified in the slide. The solution to this problem is shown on the next page.

Pivoting INSERT

```
INSERT ALL
  INTO sales_info VALUES (employee_id,week_id,sales_MON)
  INTO sales_info VALUES (employee_id,week_id,sales_TUE)
  INTO sales_info VALUES (employee_id,week_id,sales_WED)
  INTO sales_info VALUES (employee_id,week_id,sales_THUR)
  INTO sales_info VALUES (employee_id,week_id, sales_FRI)
  SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
         sales_WED, sales_THUR,sales_FRI
  FROM sales_source_data;
5 rows inserted.
```


ORACLE

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)

7 rows selected

Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	176	6	2000	3000	4000	5000	6000

```
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	176	6	2000
2	176	6	3000
3	176	6	4000
4	176	6	5000
5	176	6	6000

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

- Provides the ability to conditionally update or insert data into a database table
- Performs an UPDATE if the row exists, and an INSERT if it is a new row:
  - Avoids separate updates
  - Increases performance and ease of use
  - Is useful in data warehousing applications

ORACLE

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 the 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.

MERGE Statement Syntax

You can conditionally insert or update rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col_val1,
    col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```


ORACLE

Merging Rows

You can update existing rows and insert new rows conditionally by using 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

For more information, see *Oracle Database 10g SQL Reference*.

Merging Rows

Insert or update rows in the EMPL3 table to match the EMPLOYEES table.

```
MERGE INTO empl3  c
  USING 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.department_id  = e.department_id
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);
```


Example of Merging Rows

```
MERGE INTO empl3 c
  USING 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,
    c.commission_pct = e.commission_pct,
    c.manager_id     = e.manager_id,
    c.department_id  = e.department_id
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

```
TRUNCATE TABLE empl3;

SELECT *
FROM empl3;
no rows selected
```

```
MERGE INTO empl3 c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
     ...
WHEN NOT MATCHED THEN
 INSERT VALUES...;
```

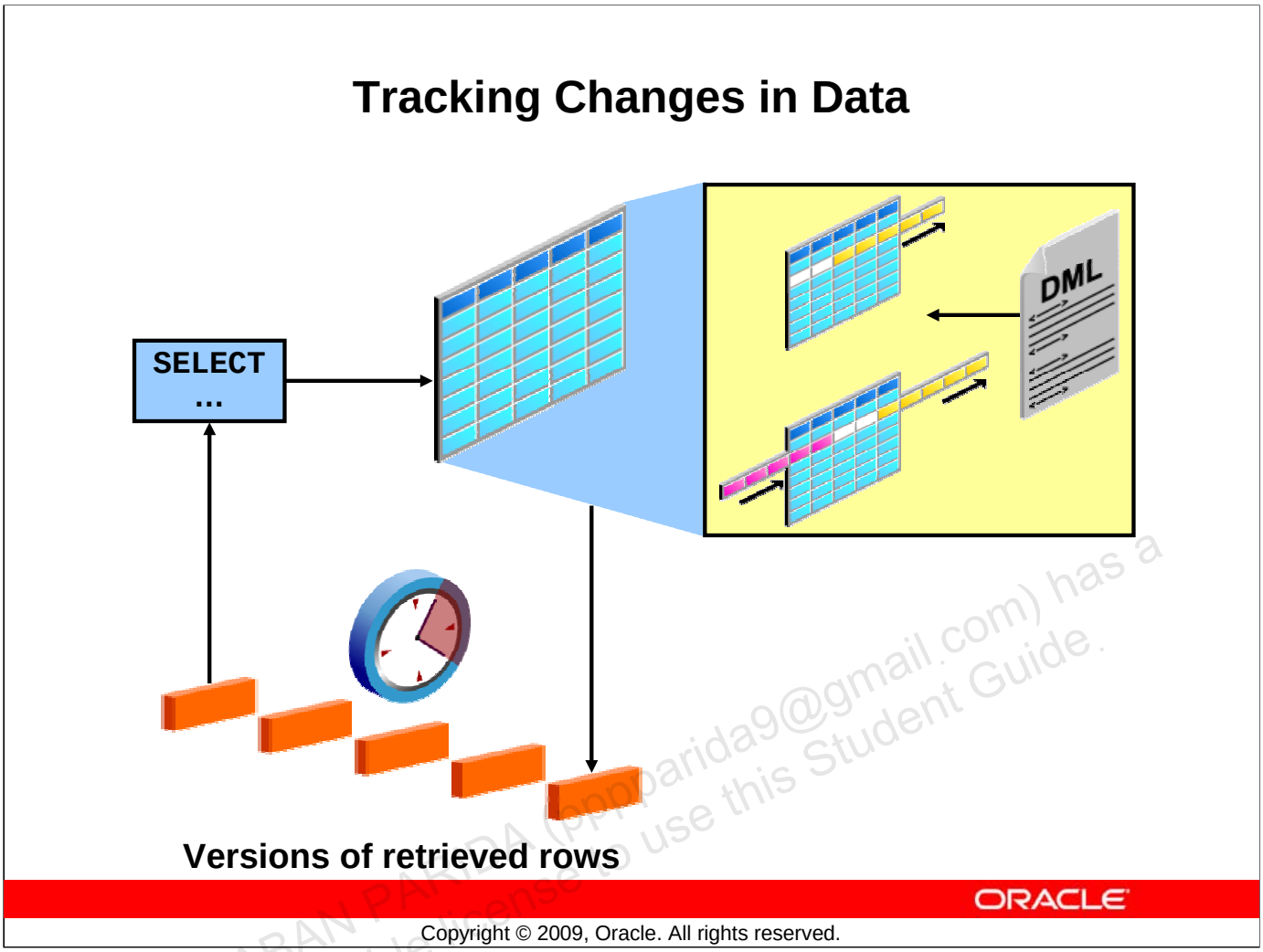
```
SELECT *
FROM empl3;

107 rows selected.
```


Example of Merging Rows (continued)

The example in the slide matches the EMPLOYEE_ID in the EMPL3 table to the EMPLOYEE_ID in the EMPLOYEES table. If a match is found, the row in the EMPL3 table is updated to match the row in the EMPLOYEES table. If the row is not found, it is inserted into the EMPL3 table.

The condition `c.employee_id = e.employee_id` is evaluated. Because the EMPL3 table is empty, 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 EMPL3 table.

If rows existed in the EMPL3 table and employee IDs matched in both tables (the EMPL3 and EMPLOYEES tables), then the existing rows in the EMPL3 table would be updated to match the EMPLOYEES table.



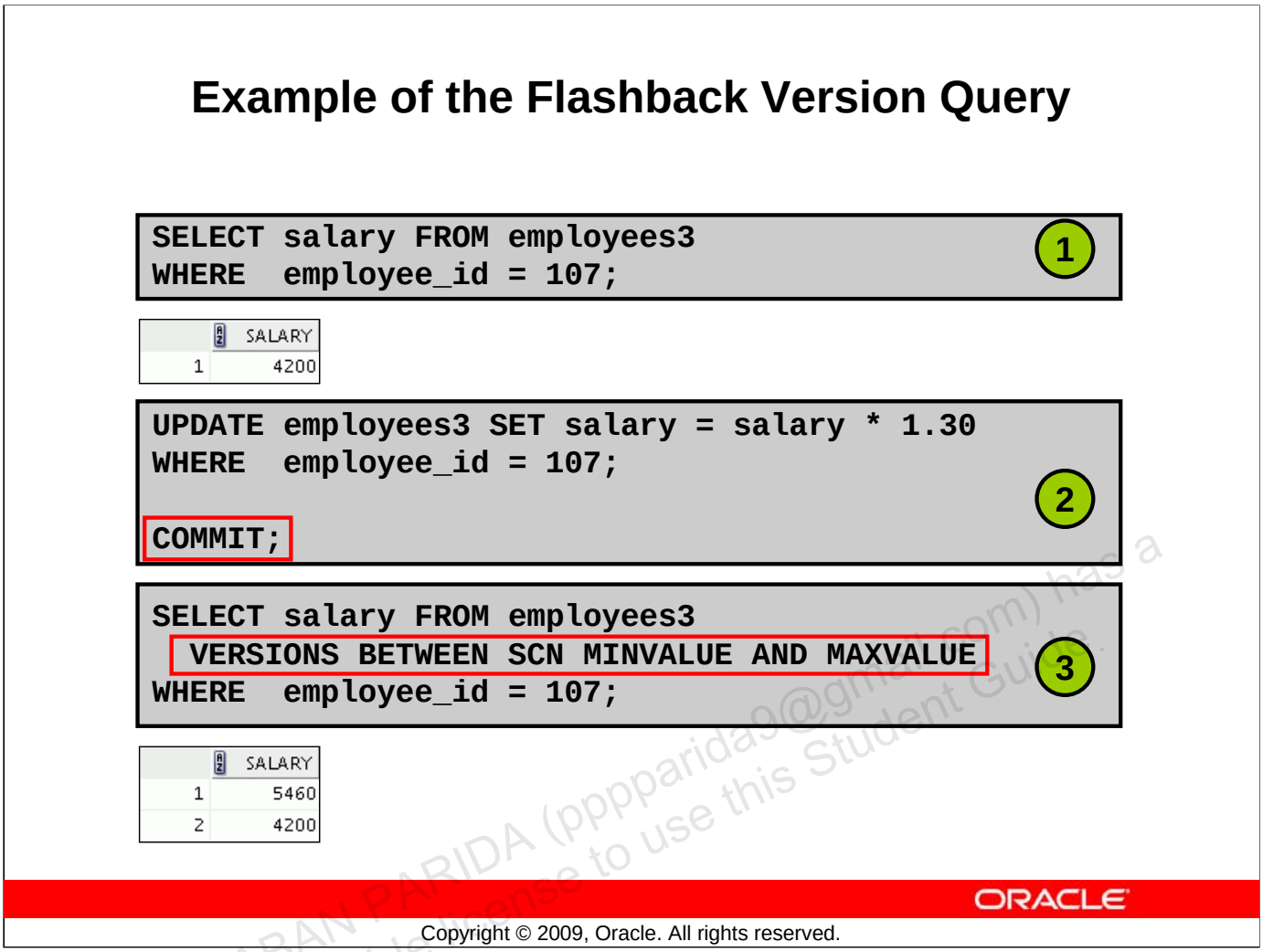
Tracking Changes in Data

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 an SCN or time stamp range between 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 then 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 if 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 a system change number (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, then 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

```
SELECT versions_starttime "START_DATE",
       versions_endtime   "END_DATE",
       salary
FROM   employees
    VERSIONS BETWEEN SCN MINVALUE
    AND MAXVALUE
WHERE  last_name = 'Lorentz';
```

	START_DATE	END_DATE	SALARY
1	14-NOV-08 02.56.36.000000000 AM	(null)	5460
2	(null)	14-NOV-08 02.56.36.000000000 AM	4200

ORACLE

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/SCN of the BETWEEN clause, then 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 bound.

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

Summary

In this lesson, you should have learned how to:

- Use DML statements and control transactions
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Pivoting `INSERT`
  - Conditional `ALL INSERT`
  - Conditional `FIRST INSERT`
- Merge rows in a table
- Manipulate data by using subqueries
- Track the changes to data over a period of time

ORACLE

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 3: Overview

This practice covers the following topics:

- Performing multitable `INSERT`s
- Performing `MERGE` operations
- Tracking row versions

Practice 3: Overview

In this practice, you add rows to the `emp_data` table, update and delete data from the table, and track your transactions.

Practice 3

1. Run the `lab_03_01.sql` script in the lab folder to create the SAL_HISTORY table.
2. Display the structure of the SAL_HISTORY table.

```
Name                          Null     Type
----------------------------- -------- ------------
EMPLOYEE_ID                            NUMBER(6)
HIRE_DATE                              DATE
SALARY                                 NUMBER(8,2)
```

3. Run the `lab_03_03.sql` script in the lab folder to create the MGR_HISTORY table.
4. Display the structure of the MGR_HISTORY table.

```
Name                          Null     Type
----------------------------- -------- ------------
EMPLOYEE_ID                            NUMBER(6)
MANAGER_ID                             NUMBER(6)
SALARY                                 NUMBER(8,2)
```

5. Run the `lab_03_05.sql` script in the lab folder to create the SPECIAL_SAL table.
6. Display the structure of the SPECIAL_SAL table.

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

7. a. Write a query to do the following:
   - Retrieve the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the EMPLOYEES table.
   - If the salary is more than $20,000, insert the employee ID and salary into the SPECIAL_SAL table.
   - Insert the employee ID, hire date, and salary into the SAL_HISTORY table.
   - Insert the employee ID, manager ID, and salary into the MGR_HISTORY table.

   b. Display the records from the SPECIAL_SAL table.

	EMPLOYEE_ID	SALARY
1	100	24000

Practice 3 (continued)

c. Display the records from the SAL_HISTORY table.

	EMPLOYEE_ID	HIRE_DATE	SALARY
1	101	21-SEP-89	17000
2	102	13-JAN-93	17000
3	103	03-JAN-90	9000
4	104	21-MAY-91	6000
...			
24	124	16-NOV-99	5800

d. Display the records from the MGR_HISTORY table.

	EMPLOYEE_ID	MANAGER_ID	SALARY
1	101	100	17000
2	102	100	17000
3	103	102	9000
4	104	103	6000
...			
24	124	100	5800

8. a. Run the lab_03_08a.sql script in the lab folder to create the SALES_SOURCE_DATA table.
b. Run the lab_03_08b.sql script in the lab folder to insert records into the SALES_SOURCE_DATA table.
c. Display the structure of the SALES_SOURCE_DATA table.

```
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)
```

d. Display the records from the SALES_SOURCE_DATA table.

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

Practice 3 (continued)

e. Run the `lab_03_08c.sql` script in the lab folder to create the `SALES_INFO` table.

f. Display the structure of the `SALES_INFO` table.

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

g. Write a query to do the following:

- Retrieve the employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the `SALES_SOURCE_DATA` table.
- Build a transformation such that each record retrieved from the `SALES_SOURCE_DATA` table is converted into multiple records for the `SALES_INFO` table.
  Hint: Use a pivoting `INSERT` statement.

h. Display the records from the `SALES_INFO` table.

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

9. You have the data of past employees stored in a flat file called `emp.data`. You want to store the names and email IDs of all the past and present employees in a table. To do this, first create an external table called `EMP_DATA` using the `emp.dat` source file in the `emp_dir` directory. You can use the script in `lab_03_09.sql` to do this.
10. Next, run the `lab_03_10.sql` script to create the `EMP_HIST` table.
    a. Increase the size of the email column to 45.
    b. Merge the data in the `EMP_DATA` table that was created in step 9 with the data in the `EMP_HIST` table. Assume that the data in the external `EMP_DATA` table is the most up-to-date. If a row in the `EMP_DATA` table matches the `EMP_HIST` table, update the email column of the `EMP_HIST` table to match the `EMP_DATA` table row. If a row in the `EMP_DATA` table does not match, insert it into the `EMP_HIST` table. Rows are considered matching when the employee's first and last names are identical.
    c. Retrieve the rows from `EMP_HIST` after the merge.

Practice 3 (continued)

	FIRST_NAME	LAST_NAME	EMAIL
1	Steven	King	SKING
2	Neena	Kochhar	nkochh@pipit.com
3	Lex	De Haan	LDEHAAN
4	Alexander	Hunold	AHun@MOORHEN.COM
5	Bruce	Ernst	BERNST
6	David	Austin	DAUSTIN
7	Valli	Pataballa	VPATABAL
8	Diana	Lorentz	DLORENTZ
9	Nancy	Greenberg	NGREENBE
10	Daniel	Faviet	DFAVIET

...

145	Diana	lorentz	dlor@limpkin.com
146	Stephen	King	sking@merganser.com
147	Hema	Voight	Hema.Voight@PHALA...
148	Nancy	greenberg	ngreenb@plover.com

11. Create the EMP3 table by using the `lab_03_11.sql` script. In the EMP3 table, change the department for Kochhar to 60 and commit your change. Next, change the department for Kochhar to 50 and commit your change. Track the changes to Kochhar by using the Row Versions feature.

	START_DATE	END_DATE	DEPARTMENT_ID
1	17-NOV-08 03.55.06.000000000 AM	(null)	50
2	17-NOV-08 03.55.06.000000000 AM	17-NOV-08 03.55.06.000000000 AM	60
3	(null)	17-NOV-08 03.55.06.000000000 AM	90