9 Manipulating Data

Objectives

After completing this lesson, you should be able to:

- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Perform flashback operations
- Track the changes made to data over a period of time

In this lesson, 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, performing flashback operations, and tracking changes in the database.

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

Explicit Default Feature: Overview

- Use the DEFAULT keyword as a column value where the default column value is desired.
- 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

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 query 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

- 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;
```


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 MANAGER_ID column. If there is no default value defined for the column, a null value is inserted instead.

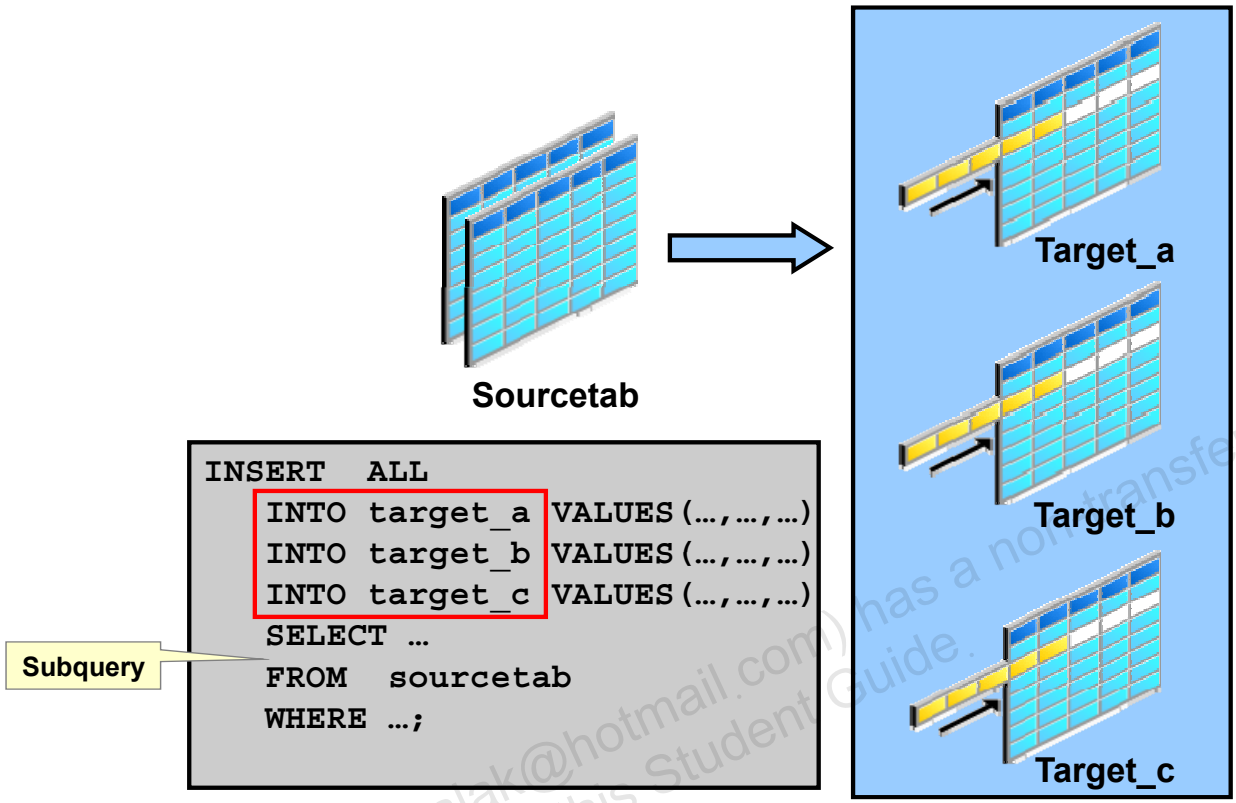
The second example uses the UPDATE statement to set the MANAGER_ID column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

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

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

Multitable INSERT Statements: Overview



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.

Multitable INSERT Statements: Overview

- Use the INSERT…SELECT statement to insert rows into multiple tables as part of a single DML statement.
- Multitable INSERT statements are 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

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

The different types of multitable INSERT statements are:

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

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.
- **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.
- **Pivoting INSERT:** This is a special case of the unconditional INSERT ALL.

Multitable INSERT Statements

- Syntax for multitable INSERT:

```
{ ALL
{ insert_into_clause [ values_clause ]}...
| conditional_insert_clause
 } subquery
```

- conditional_insert_clause:

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

ORACLE

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.

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

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

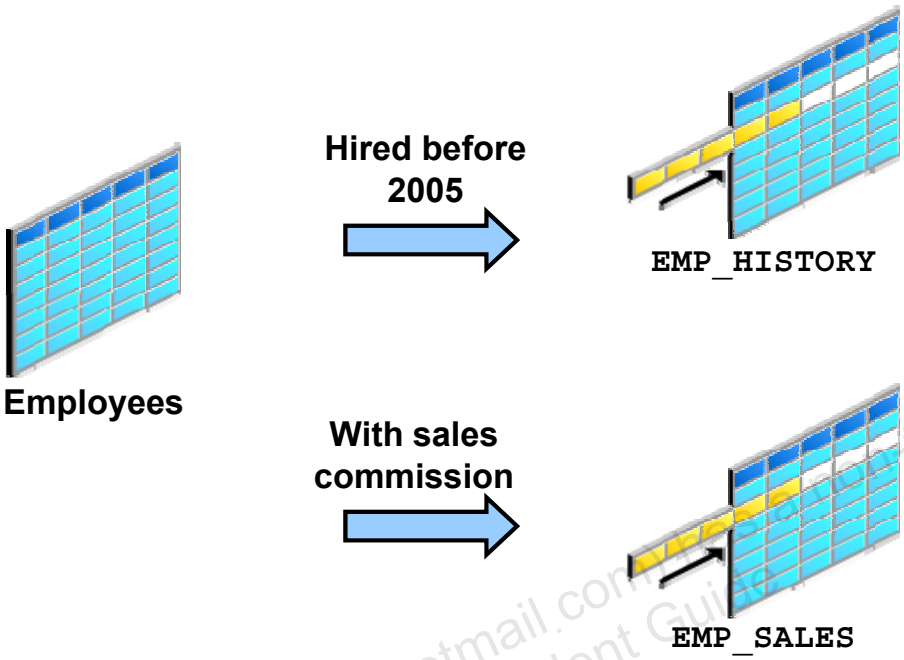
The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager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

A total of 12 rows were inserted:

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

Conditional INSERT ALL: Example



For all employees in the employees tables, if the employee was hired before 2005, 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

```
INSERT  ALL
 WHEN HIREDATE < '01-JAN-05' THEN
   INTO emp_history VALUES(EMPID,HIREDATE,SAL)
 WHEN COMM IS NOT NULL THEN
   INTO emp_sales VALUES(EMPID,COMM,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, commission_pct COMM
   FROM  employees;
```

```
59 rows inserted.
```


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 2005 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;
```

Result: 24 rows fetched.

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

Result: 35 rows fetched.

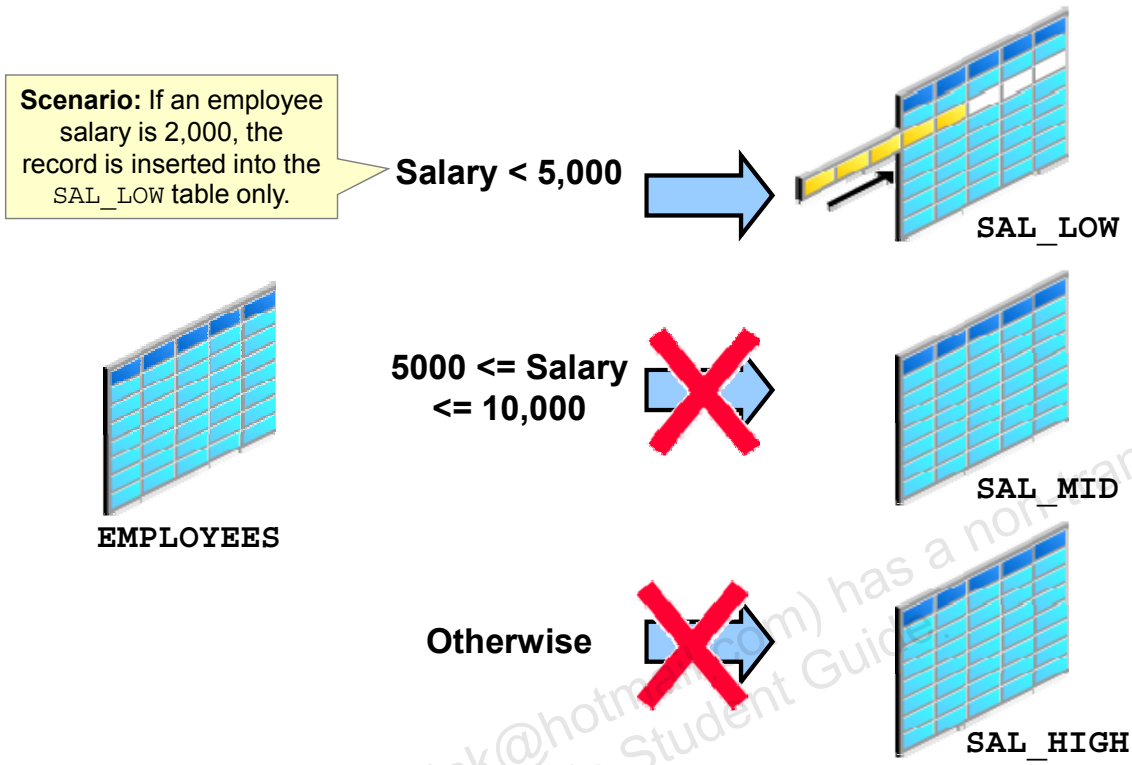
You can also optionally use the ELSE clause with the INSERT ALL statement.
Example:

```
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

```
INSERT FIRST
WHEN salary < 5000 THEN
  INTO sal_low VALUES (employee_id, last_name, salary)
WHEN salary between 5000 and 10000 THEN
  INTO sal_mid VALUES (employee_id, last_name, salary)
ELSE
  INTO sal_high VALUES (employee_id, last_name, salary)
SELECT employee_id, last_name, salary
FROM employees;
```

```
107 rows inserted
```


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.

A total of 107 rows were inserted:

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

49 rows fetched.



SELECT count(*) mid FROM sal_mid;

43 rows fetched.



SELECT count(*) high FROM sal_high;

15 rows fetched.
```

Pivoting INSERT

Convert the set of sales records from the nonrelational database table to relational format.

Emp_ID	Week_ID	MON	TUES	WED	THUR	FRI
176	6	2000	3000	4000	5000	6000

Employee_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

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

```
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
```


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
```

```
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)
```

```
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
```

```
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`.

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

MERGE Statement

- Provides the ability to conditionally update, insert, or delete 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

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.

MERGE Statement Syntax

You can conditionally insert, update, or delete 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 = col1_val,
    col2 = col2_val
 WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```


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 Language Reference* for Oracle Database 12*c*.

Merging Rows: Example

Insert or update rows in the COPY_EMP3 table to match the EMPLOYEES table.

```
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,
...

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);
```

```
107 rows merged.
```



```
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);
```

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;
```

Observe that in the output, 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

```
TRUNCATE TABLE copy_emp3;
SELECT * FROM copy_emp3;
no rows selected
```

```
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,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, ...
```

```
SELECT * FROM copy_emp3;
107 rows selected.
```

ORACLE

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;
```

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

FLASHBACK TABLE Statement

- Enables you to recover tables to a specified point in time with a single statement
- Restores table data along with associated indexes and constraints
- Enables you to revert the table and its contents to a certain point in time or system change number (SCN)

Oracle Flashback Table enables you to recover tables to a specified point in time with a single statement. You can restore table data along with associated indexes and constraints while the database is online, undoing changes to only the specified tables.

The Flashback Table feature is similar to a self-service repair tool. For example, if a user accidentally deletes important rows from a table and then wants to recover the deleted rows, you can use the `FLASHBACK TABLE` statement to restore the table to the time before the deletion and see the missing rows in the table.

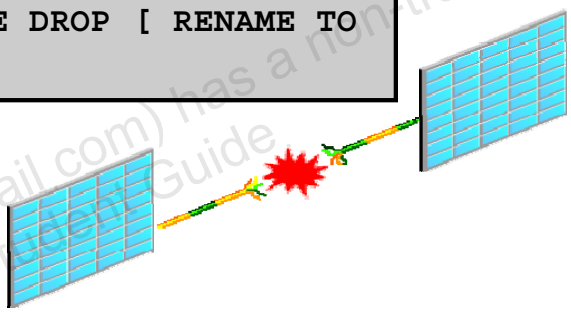
When using the `FLASHBACK TABLE` statement, you can revert the table and its contents to a certain time or to an SCN.

Note: The SCN is an integer value associated with each change to the database. It is a unique incremental number in the database. Every time you commit a transaction, a new SCN is recorded.

FLASHBACK TABLE Statement

- Repair tool for accidental table modifications
  - Restores a table to an earlier point in time
  - Offers ease of use, availability, and fast execution
  - Is performed in place
- Syntax:

```
FLASHBACK TABLE [ schema. ] table [, [ schema.
] table ]... TO { { { SCN | TIMESTAMP } expr |
RESTORE POINT restore_point } [ { ENABLE |
DISABLE } TRIGGERS ] | BEFORE DROP [ RENAME TO
table ] } ;
```



Self-Service Repair Facility

Oracle Database provides a SQL data definition language (DDL) command, `FLASHBACK TABLE`, to restore the state of a table to an earlier point in time in case it is inadvertently deleted or modified. The `FLASHBACK TABLE` command is a self-service repair tool to restore data in a table along with associated attributes such as indexes or views. This is done, while the database is online, by rolling back only the subsequent changes to the given table. Compared to traditional recovery mechanisms, this feature offers significant benefits such as ease of use, availability, and faster restoration. It also takes the burden off the DBA to find and restore application-specific properties. The flashback table feature does not address physical corruption caused because of a bad disk.

Syntax

You can invoke a `FLASHBACK TABLE` operation on one or more tables, even on tables in different schemas. You specify the point in time to which you want to revert by providing a valid time stamp. By default, database triggers are disabled during the flashback operation for all tables involved. You can override this default behavior by specifying the `ENABLE TRIGGERS` clause.

Note: For more information about recycle bin and flashback semantics, refer to *Oracle Database Administrator's Guide* for Oracle Database 12*c*.

Using the FLASHBACK TABLE Statement

```
DROP TABLE emp3;
```

```
table EMP3 dropped.
```

```
SELECT original_name, operation, droptime FROM
recyclebin;
```

	ORIGINAL_NAME	OPERATION	DROPTIME
1	EMP3	DROP	2012-10-16:05:59:34

...

```
FLASHBACK TABLE emp3 TO BEFORE DROP;
```

```
table EMP3 succeeded.
```


Syntax and Examples

The example restores the `EMP3` table to a state before a `DROP` statement.

The recycle bin is actually a data dictionary table containing information about dropped objects. Dropped tables and any associated objects—such as, indexes, constraints, nested tables, and so on—are not removed and still occupy space. They continue to count against user space quotas until specifically purged from the recycle bin, or until they must be purged by the database because of tablespace space constraints.

Each user can be thought of as an owner of a recycle bin because, unless a user has the `SYSDBA` privilege, the only objects that the user has access to in the recycle bin are those that the user owns. A user can view his or her objects in the recycle bin by using the following statement:

```
SELECT * FROM RECYCLEBIN;
```

When you drop a user, any objects belonging to that user are not placed in the recycle bin and any objects in the recycle bin are purged.

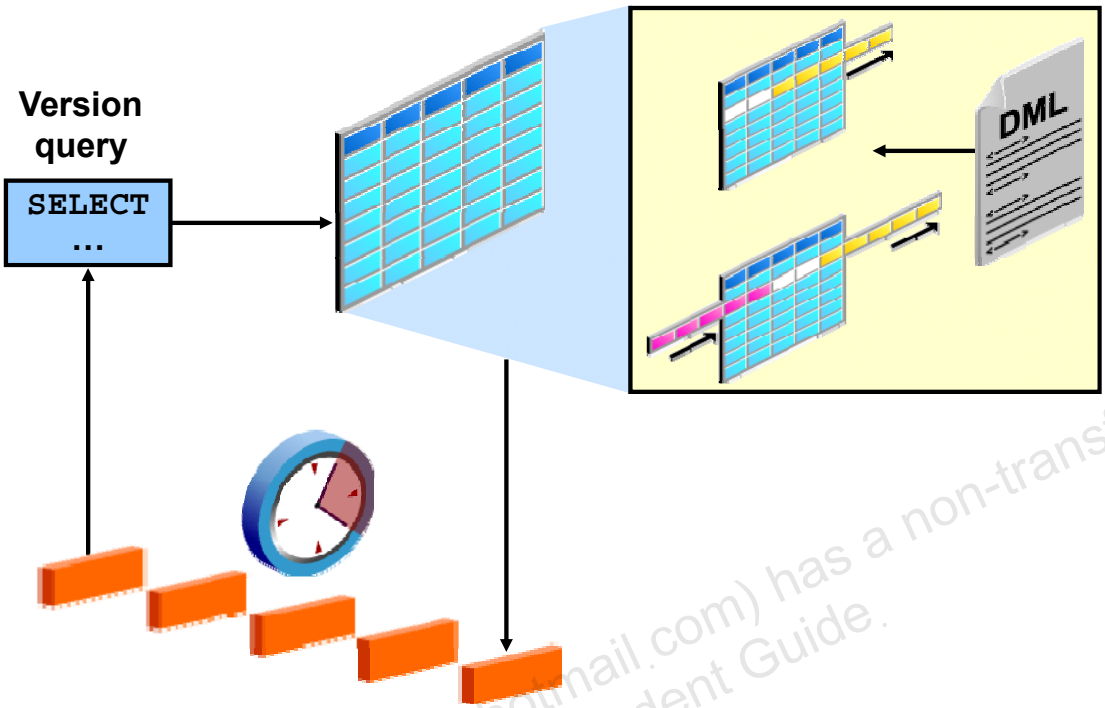
You can purge the recycle bin with the following statement:

```
PURGE RECYCLEBIN;
```

Lesson Agenda

- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking the changes to data over a period of time

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. You can use Oracle Flashback Query to retrieves data as it existed at an earlier 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.

Flashback Query: Example

```
SELECT salary FROM employees3
WHERE last_name = 'Chung';
```

```
UPDATE employees3 SET salary = 4000
WHERE last_name = 'Chung';

SELECT salary FROM employees3
WHERE last_name = 'Chung';
```

```
SELECT salary FROM employees3
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' MINUTE)
WHERE last_name = 'Chung';
```

1

	SALARY
1	3800

2

	SALARY
1	4000

3

	SALARY
1	3800

To use Oracle Flashback Query, use a `SELECT` statement with an `AS OF` clause. Oracle Flashback Query retrieves data as it existed at some time in the past. The query explicitly references a past time through a timestamp or System Change Number (SCN). It returns committed data that was current at that point in time.

In the example in the slide, the salary for employee Chung is retrieved (1). The salary for employee Chung is increased to 4000 (2). To learn what the value was before the update, you can use the Flashback Query(3).

Oracle Flashback Query can be used in the following scenarios:

- Recovering lost data or undoing incorrect, committed changes. For example, if you mistakenly delete or update rows, and then commit them, you can immediately undo the mistake.
- Comparing current data with the corresponding data at some time in the past. For example, you can run a daily report that shows the change in data from yesterday. You can compare individual rows of table data or find intersections or unions of sets of rows.
- Checking the state of transactional data at a particular time

Flashback Version Query: Example

```
SELECT salary FROM employees3
WHERE  employee_id = 107;
```
1

```
UPDATE employees3 SET salary = salary * 1.30
WHERE  employee_id = 107;

COMMIT;
```
2

```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```
3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

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. 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	11-SEP-12 03.38.54.000000000 AM	(null)	5460
2	(null)	11-SEP-12 03.38.54.000000000 AM	4200

```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```

ORACLE

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.

a. True
b. False

Answer: a

Quiz

In all the cases, when you execute a `DROP TABLE` command, the database renames the table and places it in a recycle bin, from where it can later be recovered by using the `FLASHBACK TABLE` statement.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Perform flashback operations
- Track the changes to data over a period of time

In this lesson, you also should have learned about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.

Practice 9: Overview

This practice covers the following topics:

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

ORACLE

In this practice, you learn how to perform multitable `INSERTS`, `MERGE` operations, flashback operations, and tracking row versions.