20

Oracle9*i* Extensions to DML and DDL Statements

ORACLE

Schedule:	Timing	Topic
	40 minutes	Lecture
	30 minutes	Practice
	70 minutes	Total

Objectives

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

- **Describe the features of multitable inserts**
- **Use the following types of multitable inserts**
  - **Unconditional `INSERT`**
  - **Pivoting `INSERT`**
  - **Conditional `ALL INSERT`**
  - **Conditional `FIRST INSERT`**
- **Create and use external tables**
- **Name the index at the time of creating a primary key constraint**

ORACLE

Lesson Aim

This lesson addresses the Oracle9*i* extensions to DDL and DML statements. It focuses on multitable `INSERT` statements, types of multitable `INSERT` statements, external tables, and the provision to name the index at the time of creating a primary key constraint.

Review of the INSERT Statement

- Add new rows to a table by using the INSERT statement.

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- Only one row is inserted at a time with this syntax.

```
INSERT INTO departments(department_id, department_name,
                        manager_id, location_id)
VALUES      (70, 'Public Relations', 100, 1700);
1 row created.
```

ORACLE

Review of the INSERT Statement

You can add new rows to a table by issuing the INSERT statement.

In the syntax:

table	is the name of the table
column	is the name of the column in the table to populate
value	is the corresponding value for the column

Note: This statement with the VALUES clause adds only one row at a time to a table.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Review of the UPDATE Statement

- Modify existing rows with the UPDATE statement.

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- Update more than one row at a time, if required.
- Specific row or rows are modified if you specify the WHERE clause.

```
UPDATE employees
SET    department_id = 70
WHERE  employee_id = 142;
1 row updated.
```

ORACLE

Review of the UPDATE Statement

You can modify existing rows by using the UPDATE statement.

In the syntax:

table	is the name of the table
column	is the name of the column in the table to populate
value	is the corresponding value or subquery for the column
condition	identifies the rows to be updated and is composed of column names expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

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 do multiple inserts using IF...THEN syntax**

ORACLE

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 needs to 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 has to be identified and extracted from many different sources, such as database systems and applications. After extraction, the data has to 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.

Once data is loaded into an Oracle9*i*, database, data transformations can be executed using SQL operations. With Oracle9*i* multitable INSERT statements is one of the techniques for implementing SQL data transformations.

Multitable `INSERTS` statement offer the benefits of the `INSERT ... SELECT` statement when multiple tables are involved as targets. Using functionality prior to Oracle9*i*, 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 more relational database table environment. To implement this functionality before Oracle9*i*, you had to write multiple `INSERT` statements.

Types of Multitable INSERT Statements

Oracle9*i* introduces the following types of multitable insert statements:

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

ORACLE

Types of Multitable INSERT Statements

Oracle 9*i* introduces the following types of multitable INSERT statements:

- 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. There are four types of multitable insert statements.

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

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 FIRST: INSERT

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 inserts only on tables, not on views or materialized views.
- You cannot perform a multitable insert into a remote table.
- You cannot specify a table collection expression when performing a multitable insert.
- In a multitable insert, all of the 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 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;
8 rows created.
```

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 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, as 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 have to 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 8 rows created can be interpreted to mean that a total of eight insertions were performed on the base tables SAL_HISTORY and MGR_HISTORY.

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script files lab\cre_sal_history.sql and lab\cre_mgr_history.sql, which create the SAL_HISTORY and MGR_HISTORY tables.

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 the preceding problem is shown in 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 created.
```

ORACLE

Conditional INSERT ALL (continued)

The example in the slide is similar to the example on the previous slide as it 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 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 a conditional ALL INSERT, as 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 10000 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 FIRST INSERT

- Select the DEPARTMENT_ID, SUM(SALARY) and MAX(HIRE_DATE) from the EMPLOYEES table.
- If the SUM(SALARY) is greater than $25,000 then insert these values into the SPECIAL_SAL, 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, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIRE_DATE column using a conditional multitable INSERT.

ORACLE

Conditional FIRST INSERT

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

Conditional FIRST INSERT

```
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;
8 rows created.
```

ORACLE

Conditional FIRST INSERT (continued)

The example in the slide inserts rows into more than one table, using one single INSERT statement. The SELECT statement retrieves the details of 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, as 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 just as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

The feedback 8 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.

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script files lab\cre_special_sal.sql, lab\cre_hiredate_history_99.sql lab\cre_hiredate_history_00.sql and lab\cre_hiredate_history.sql

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 would 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 need to 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.

In order to solve the problem mentioned in the slide, you need to 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 the preceding problem is shown in 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 created.
```

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)

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script files `lab\cre_sales_source_data.sql`, `lab\cre_sales_info.sql` and `lab\popul_sales_source_data.sql`.

```
SELECT * FROM SALES_SOURCE_DATA;
```

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

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

External Tables

- **External tables are read-only tables in which the data is stored outside the database in flat files.**
- **The metadata for an external table is created using a `CREATE TABLE` statement.**
- **With the help of external tables, Oracle data can be stored or unloaded as flat files.**
- **The data can be queried using SQL, but you cannot use DML and no indexes can be created.**

External Tables

An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. Using the Oracle9*i* external table feature, you can use external data as a virtual table. This data can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (`UPDATE`, `INSERT`, or `DELETE`) are possible, and no indexes can be created on them.

The means of defining the metadata for external tables is through the `CREATE TABLE ... ORGANIZATION EXTERNAL` statement. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database.

The Oracle Server provides two major access drivers for external tables. One, the loader access driver, or `ORACLE_LOADER`, is used for reading of data from external files using the Oracle loader technology. This access driver allows the Oracle Server to access data from any data source whose format can be interpreted by the SQL*Loader utility. The other Oracle provided access driver, the import/export access driver, or `ORACLE_INTERNAL`, can be used for both the importing and exporting of data using a platform independent format.

Creating an External Table

- **Use the `external_table_clause` along with the `CREATE TABLE` syntax to create an external table.**
- **Specify `ORGANIZATION` as `EXTERNAL` to indicate that the table is located outside the database.**
- **The `external_table_clause` consists of the access driver `TYPE`, `external_data_properties`, and the `REJECT LIMIT`.**
- **The `external_data_properties` consist of the following:**
  - **`DEFAULT DIRECTORY`**
  - **`ACCESS PARAMETERS`**
  - **`LOCATION`**

ORACLE

Creating an External Table

You create external tables using the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement. You are not in fact creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. The `ORGANIZATION` clause lets you specify the order in which the data rows of the table are stored. By specifying `EXTERNAL` in the `ORGANIZATION` clause, you indicate that the table is a read-only table located outside the database.

`TYPE access_driver_type` indicates the access driver of the external table. The access driver is the Application Programming Interface (API) that interprets the external data for the database. If you do not specify `TYPE`, Oracle uses the default access driver, `ORACLE_LOADER`.

The `REJECT LIMIT` clause lets you specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is 0.

`DEFAULT DIRECTORY` lets you specify one or more default directory objects corresponding to directories on the file system where the external data sources may reside. Default directories can also be used by the access driver to store auxiliary files such as error logs. Multiple default directories are permitted to facilitate load balancing on multiple disk drives.

The optional `ACCESS PARAMETERS` clause lets you assign values to the parameters of the specific access driver for this external table. Oracle does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

The `LOCATION` clause lets you specify one external locator for each external data source. Usually the `location_specifier` is a file, but it need not be. Oracle does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.

Example of Creating an External Table

Create a DIRECTORY object that corresponds to the directory on the file system where the external data source resides.

```
CREATE DIRECTORY emp_dir AS '/flat_files' ;
```

ORACLE

Example of Creating an External Table

Use the CREATE DIRECTORY statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than hard-code the operating system pathname, for greater file management flexibility.

You must have CREATE ANY DIRECTORY system privileges to create directories. When you create a directory, you are automatically granted the READ object privilege and can grant READ privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

Syntax

```
CREATE [OR REPLACE] DIRECTORY AS 'path_name';
```

In the syntax:

OR REPLACE Specify OR REPLACE to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges.

directory Specify the name of the directory object to be created. The maximum length of directory is 30 bytes. You cannot qualify a directory object with a schema name.

'path_name' Specify the full pathname of the operating system directory on the result that the path name is case sensitive.

```
CREATE TABLE oldemp (
  empno NUMBER, empname CHAR(20), birthdate DATE)
  ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
  DEFAULT DIRECTORY emp_dir
  ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE
  BADFILE 'bad_emp'
  LOGFILE 'log_emp'
  FIELDS TERMINATED BY ','
  (empno CHAR,
  empname CHAR,
  birthdate CHAR date_format date mask "dd-mon-yyyy"))
  LOCATION ('emp1.txt'))
  PARALLEL 5
  REJECT LIMIT 200;
Table created.
```

ORACLE

Example of Creating an External Table (continued)

Assume that there is a flat file that has records in the following format:

```
10,jones,11-Dec-1934
20,smith,12-Jun-1972
```

Records are delimited by new lines, and the fields are all terminated by a comma (,). The name of the file is: `/flat_files/emp1.txt`

To convert this file as the data source for an external table, whose metadata will reside in the database, you need to perform the following steps:

1. Create a directory object `emp_dir` as follows:

   ```
   CREATE DIRECTORY emp_dir AS '/flat_files' ;
   ```

2. Run the `CREATE TABLE` command shown in the slide.

The example in the slide illustrates the table specification to create an external table for the file:

```
/flat_files/emp1.txt
```

In the example, the `TYPE` specification is given only to illustrate its use. `ORACLE_LOADER` is the default access driver if not specified. The `ACCESS PARAMETERS` provide values to parameters of the specific access driver and are interpreted by the access driver, not by the Oracle Server.

The `PARALLEL` clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the `INSERT INTO TABLE` statement. For example, if `PARALLEL=5` were specified, then more that one parallel execution server could be working on a data source. Because external tables can be very large, for performance reasons it is advisable to specify the `PARALLEL` clause, or a parallel hint for the query.

The REJECT LIMIT clause specifies that if more than 200 conversion errors occur during a query of the external data, the query is aborted and an error returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition.

Once the CREATE TABLE command executes successfully, the external table OLDEMP can be described and queried like a relational table.

```
DESC oldemp
```

Name	Null?	Type
EMPNO		NUMBER
EMPNAME		CHAR(20)
BIRTHDATE		DATE

In the following example, the INSERT INTO TABLE statement generates a dataflow from the external data source to the Oracle SQL engine where data is processed. As data is extracted from the external table, it is transparently converted by the ORACLE_ LOADER access driver from its external representation into an equivalent Oracle native representation. The INSERT statement inserts data from the external table OLDEMP into the BIRTHDAYS table:

```
INSERT INTO birthdays(empno, empname, birthdate)
               SELECT empno, empname, birthdate
               FROM   oldemp;
2 rows created.
```

We can now select from the BIRTHDAYS table.

```
SELECT * FROM birthdays;
```

EMPNO	EMPNAME	BIRTHDATE
10	jones	11-DEC-34
20	smith	12-JUN-97

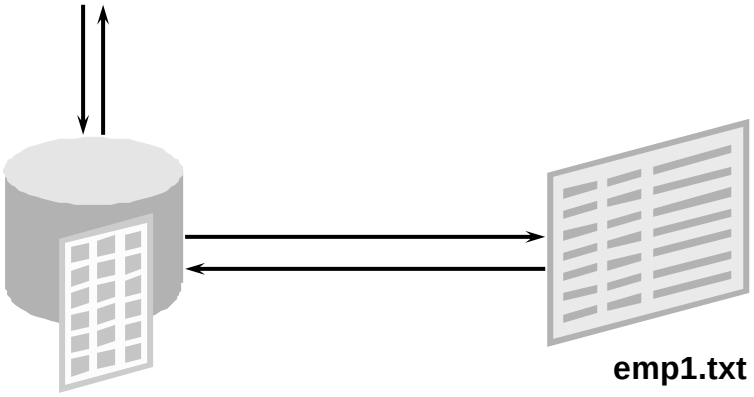
Inst

To run the code example in the slide, do the following:

1. Login to unix teach account and type the following:
   ```
   cd FLAT_FILES
   pwd
   ```
   The output should resemble /home#/teach#/FLAT_FILES
2. Open the file cre_dir.sql from the lab folder and replace the last command in the file with the output from the unix pwd.
   The last command in the file will now look like this:
   ```
   CREATE OR REPLACE emp_dir as '<output from unix pwd>';
   ```
3. Save the file cre_dir.sql and execute this file in *i*SQL*Plus
4. Run the cre_birthdays.sql script to create the BIRTHDAYS table.

Querying External Tables

```
SELECT  *
FROM oldemp
```



Querying External Table

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

When the database server needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring the data from the data source is processed so that it matches the definition of the external table.

CREATE INDEX with CREATE TABLE Statement

```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
             PRIMARY KEY USING INDEX
             (CREATE INDEX emp_id_idx ON
             NEW_EMP(employee_id)),
first_name   VARCHAR2(20),
last_name    VARCHAR2(25));
Table created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'NEW_EMP';
```

INDEX_NAME	TABLE_NAME
EMP_ID_IDX	NEW_EMP

ORACLE

CREATE INDEX with CREATE TABLE Statement

In the example in the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a primary key index explicitly. This is an enhancement provided with Oracle9*i*. You can now name your indexes at the time of PRIMARY key creation, unlike before where the Oracle Server would create an index, but you did not have any control over the name of the index. The following example illustrates this:

```
CREATE TABLE EMP_UNNAMED_INDEX
(employee_id NUMBER(6) PRIMARY KEY ,
 first_name VARCHAR2(20),
 last_name VARCHAR2(25));
Table created.
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

INDEX_NAME	TABLE_NAME
SYS_C002835	EMP_UNNAMED_INDEX

(

column. But this name is cryptic and not easily understood. With Oracle9*i*, you can name your PRIMARY KEY column indexes, as you create the table with the CREATE TABLE statement. However, prior to Oracle9*i*, if you named your primary key constraint at the time of constraint creation, the index would also be created with the same name as the constraint name.

Summary

In this lesson, you should have learned how to:

- **Use the `INSERT…SELECT` statement to insert rows into multiple tables as part of a single DML statement**
- **Create external tables**
- **Name indexes using the `CREATE INDEX` statement along with the `CREATE TABLE` statement**

ORACLE

Summary

Oracle 9*i* introduces the following types of multitable `INSERT` statements.

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

Use the `external_table_clause` to create an external table, which is a read-only table whose metadata is stored in the database but whose data is stored outside the database. You can use external tables to query data without first loading it into the database.

With Oracle9*i*, you can name your `PRIMARY KEY` column indexes as you create the table with the `CREATE TABLE` statement.

Practice 20 Overview

This practice covers the following topics:

- **Writing unconditional `INSERT` statements**
- **Writing conditional `ALL INSERT` statements**
- **Pivoting `INSERT` statements**
- **Creating indexes along with the `CREATE TABLE` command**

Practice 20 Overview

In this practice, you write multitable inserts and use the `CREATE INDEX` command at the time of table creation, along with the `CREATE TABLE` command.

1. Run the `cre_sal_history.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 `cre_mgr_history.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 `cre_special_sal.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 details of 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 details of employee ID and salary into the `SPECIAL_SAL` table.
- Insert the details of employee ID, hire date , salary into the `SAL_HISTORY` table.
- Insert the details of the employee ID, manager ID, and salary into the `MGR_HISTORY` table.

Practice 10 (continued)

b. Display the records from the `SPECIAL_SAL` table.

EMPLOYEE_ID	SALARY
100	24000

c. Display the records from the `SAL_HISTORY` table.

EMPLOYEE_ID	HIRE_DATE	SALARY
101	21-SEP-89	17000
102	13-JAN-93	17000
103	03-JAN-90	9000
104	21-MAY-91	6000
107	07-FEB-99	4200
124	16-NOV-99	5800

6 rows selected.

d. Display the records from the `MGR_HISTORY` table.

EMPLOYEE_ID	MANAGER_ID	SALARY
101	100	17000
102	100	17000
103	102	9000
104	103	6000
107	103	4200
124	100	5800

6 rows selected.

Practice 20 (continued)

8.a. Run the `cre_sales_source_data.sql` script in the lab folder to create the SALES_SOURCE_DATA table.

b. Run the `ins_sales_source_data.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
178	6	1750	2200	1500	1500	3000

f. Display the structure of the SALES_INFO table.

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

Practice 20 (continued)

g. Write a query to do the following:

Retrieve the details of 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
178	6	1750
178	6	2200
178	6	1500
178	6	1500
178	6	3000

9. a. Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

COLUMN Name	**Deptno**	**Dname**
Primary Key	Yes	
Datatype	Number	VARCHAR2
Length	4	30

b. Query the USER_INDEXES table to display the INDEX_NAME for the DEPT_NAMED_INDEX table.

INDEX_NAME	TABLE_NAME
DEPT_PK_IDX	DEPT_NAMED_INDEX