Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #9

Today's Topics

- Packages
- Triggers
- Indexing

Packages

Packages Specifications

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms. All objects placed in the specification are called public objects. Any subprogram not in the package specification but coded in the package body is called a private object.
The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.
The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the cust_sal package created above.

```
PROCEDURE find_sal(c_id customers.id%TYPE) IS
c_sal customers.salary%TYPE;
BEGIN
SELECT salary INTO c_sal
FROM customers
WHERE id = c_id;
dbms_output.put_line('Salary: '|| c_sal);
END find_sal;
END cust_sal;
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the find_sal method of the cust_sal package:

```
DECLARE
  code customers.id%type := &cc_id;
BEGIN
```

```
     cust_sal.find_sal(code);
END;
```

Example:

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records:

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |
| 2 | Khilan | 25 | Delhi | 3000.00 |
| 3 | kaushik | 23 | Kota | 3000.00 |
| 4 | Chaitali | 25 | Mumbai | 7500.00 |
| 5 | Hardik | 27 | Bhopal | 9500.00 |
| 6 | Komal | 22 | MP | 5500.00 |
+----+----------+-----+-----------+----------+
```

THE PACKAGE SPECIFICATION:

```
CREATE OR REPLACE PACKAGE c_package AS
```

-- Adds a customer

```
PROCEDURE addCustomer(c_id customers.id%type,
c_name customers.name%type,
c_age customers.age%type,
c_addr customers.address%type,
c_sal customers.salary%type);
```

-- Removes a customer

```
PROCEDURE delCustomer(c_id customers.id%TYPE);
```

-- Lists all customers

```
PROCEDURE listCustomer;
END c_package;
```

CREATING THE PACKAGE BODY:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
PROCEDURE addCustomer(c_id customers.id%type,
c_name customers.name%type,
c_age customers.age%type,
c_addr customers.address%type,
c_sal customers.salary%type) IS
BEGIN
INSERT INTO customers (id,name,age,address,salary)
VALUES(c_id, c_name, c_age, c_addr, c_sal);
END addCustomer;
PROCEDURE delCustomer(c_id customers.id%type) IS
BEGIN
DELETE FROM customers
WHERE id = c_id;
END delCustomer; PROCEDURE listCustomer IS
CURSOR c_customers is
SELECT name FROM customers;
TYPE c_list is TABLE OF customers.name%type;
name_list c_list := c_list();
counter integer :=0;
BEGIN
FOR n IN c_customers LOOP
counter := counter +1;
name_list.extend;
```

```
name_list(counter) := n.name;
dbms_output.put_line('Customer(' ||counter||
        ')'||name_list(counter));
END LOOP;
END listCustomer;
END c_package;
```

USING THE PACKAGE:
The following program uses the methods declared and defined in the package c_package.

```
DECLARE
code customers.id%type:= 8;
BEGIN
c_package.addcustomer(7, 'Rajnish', 25, 'Chennai',
        3500);
c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
c_package.listcustomer;
c_package.delcustomer(code);
c_package.listcustomer;
END;
```

Triggers

PL/SQL block or PL/SQL procedure associated with table, view, schema, or the database
It implicitly gets executed whenever a particular event takes place and the event might be in the form of:

- System event
- DML statement being issued against the table
- Oracle also allows creating triggers for views.

Features of Triggers

- Trigger includes execution of SQL and PL/SQL statements as a unit.
- Triggers are stored separately from their associated tables in Db.
- Triggers are similar to subprograms in the following ways:
  - Implementing PL/SQL blocks with declarative, executable, and exception-handling sections
  - Can be stored in Database and can invoke other stored procedures

Benefits of Triggers

- Audit data modifications
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicated tables
- Enable building complex updatable views
- Useful in tracking system events

Types of Triggers

- Application trigger
  - Executes whenever an event occurs within a particular application
- Database trigger
  - Executes whenever a data event or system event occurs on a schema or database
    - Data Event implies DML or DDL.

        - System Event implies SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.

Difference between Triggers and Subprograms

Triggers	Subprograms
Defined with CREATE TRIGGER	Defined with CREATE PROCEDURE / CREATE FUNCTION
Are executed implicitly whenever the triggering event occurs	Are executed explicitly by a user, application, or a trigger
Do not accept arguments	Can accept arguments
Data dictionary contains source code in USER_TRIGGERS	Data dictionary contains source code in USER_SOURCE
Implicitly invoked	Explicitly invoked
COMMIT, SAVEPOINT, and ROLLBACK are not allowed	COMMIT, SAVEPOINT, and ROLLBACK are allowed

Application Triggers

- Are executed implicitly when a DML event occurs with respect to an application (usually a front end application)
- Operates in the Client layer in client/server architecture

Database Triggers

- Are PL/SQL program units that are associated with a specific table or view
- Operate in Client/Server architecture in the server layer
- Are compiled and stored permanently in the database
    – Hence the name “database” triggers!
- Are integrated with middle tier applications

Parts of Database Triggers

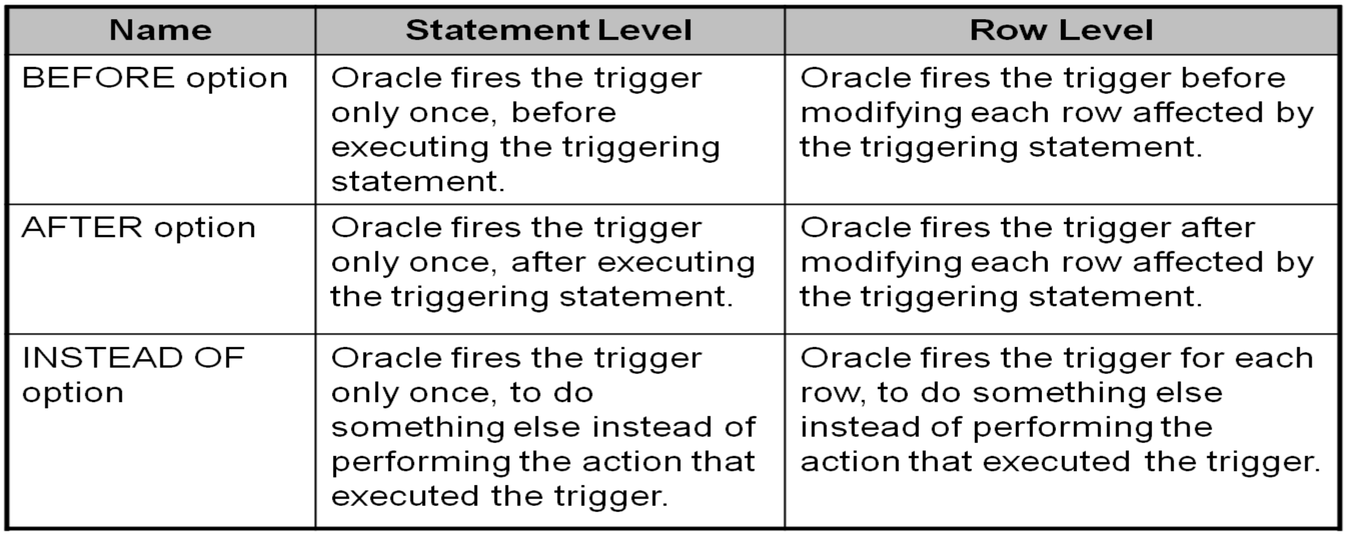
- Whenever the trigger event occurs the database trigger is implicitly fired and the PL/SQL block performs the action.
- The four parts of database trigger are:
    – Trigger Timing
        - Determines when the trigger executes in relation to the triggering event
        - Can be BEFORE, AFTER, or INSTEAD OF
    – Triggering Event
        - (or statement) is the SQL statement that causes a trigger to be executed
        - Can be an INSERT, UPDATE, or DELETE statement for a specific table or view
    – Trigger Restriction (optional)
        - Is also known as Trigger Constraint
        - Specifies a Boolean restriction that must be TRUE for the trigger action to be executed to fire
        - Is specified using a WHEN clause
        - Is an option available for triggers which is applicable for each row
        - Conditionally controls the execution of a trigger action

  - Trigger Action
    - Is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed
    - Is executed when a triggering statement is issued and the trigger restriction (if present) evaluates to TRUE

Types of Database Triggers

Based on Level	• STATEMENT level Trigger • ROW level Trigger
Based on Timing	• BEFORE Trigger • AFTER Trigger • INSTEAD OF Trigger

Types of Triggers based on timing



Name	Statement Level	Row Level
BEFORE option	Oracle fires the trigger only once, before executing the triggering statement.	Oracle fires the trigger before modifying each row affected by the triggering statement.
AFTER option	Oracle fires the trigger only once, after executing the triggering statement.	Oracle fires the trigger after modifying each row affected by the triggering statement.
INSTEAD OF option	Oracle fires the trigger only once, to do something else instead of performing the action that executed the trigger.	Oracle fires the trigger for each row, to do something else instead of performing the action that executed the trigger.

Defining Database Triggers

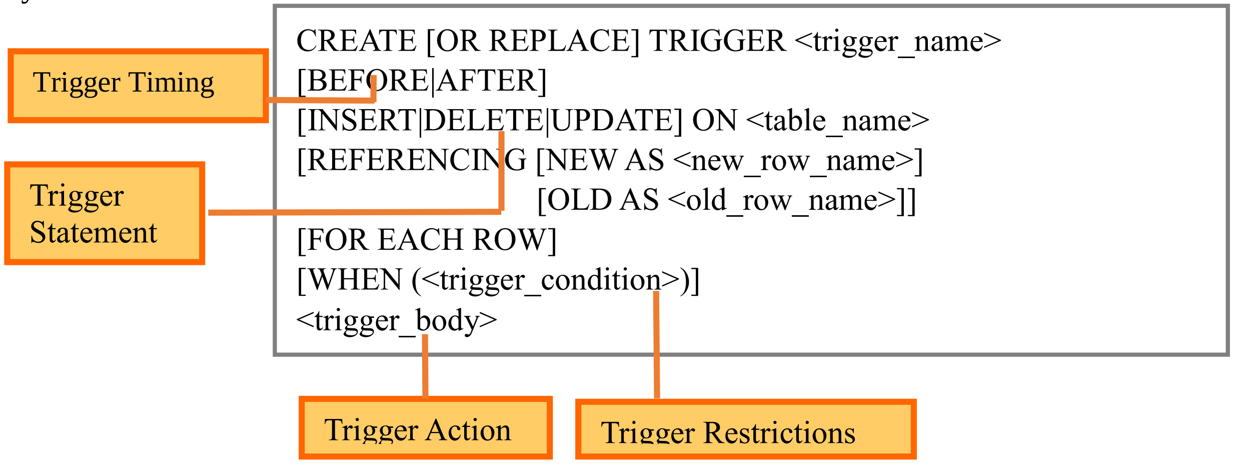
- Database triggers can be defined on:
  - DML statements
    - INSERT, UPDATE, and DELETE
  - DDL events
    - CREATE, DROP, and ALTER
  - Database operations
    - SERVERERROR, LOGON, LOGOFF, STARTUP, and SHUTDOWN

Creating Database Trigger

- CREATE TRIGGER is used to create a new trigger.
- REPLACE TRIGGER is used to replace an existing trigger.
- To create a trigger the user must have:

- CREATE TRIGGER privileges to own the table
- ALTER TABLE privileges for that table
- ALTER ANY TABLE privileges

Syntax:-



```
CREATE [OR REPLACE] TRIGGER <trigger_name>
[BEFORE|AFTER]
[INSERT|DELETE|UPDATE] ON <table_name>
[REFERENCING [NEW AS <new_row_name>]
             [OLD AS <old_row_name>]]
[FOR EACH ROW]
[WHEN (<trigger_condition>)]
<trigger_body>
```

- REFERENCING specifies correlation names to avoid name conflicts.
- User could use the correlation names in the PL/SQL block and WHEN clause of a row triggers to refer specifically to old and new values of the current row.
- Default correlation names are OLD and NEW.
- WHEN specifies the trigger restriction. This condition:
  - Has to be satisfied to fire the trigger.
  - Can be specified for the ROW TRIGGER.
- FOR EACH ROW is used only for row level triggers.

Execution of Trigger

- It is executed implicitly when DML, DDL, or system event occurs.
- The act of executing a trigger is also known as **firing the trigger**.
- When multiple triggers are to be executed, they have to be executed in a sequence.
- Database triggers execute with the privileges of the owner, not the current user.
- The following list shows the sample execution sequence:

BEFORE statement trigger	Fires once
BEFORE row trigger	Fires once for each row
AFTER row trigger	Fires once for each row
AFTER statement trigger	Fires once

Example:-

- This example creates a trigger that restricts modification in the EMP table to certain business hours

- Say modification should not be allowed during 14:00 to 18:00 hrs

```
CREATE OR REPLACE TRIGGER emp_update_chk
BEFORE UPDATE ON EMP
BEGIN
IF (TO_CHAR (sysdate, ‘HH24’) BETWEEN ’14’ AND ’18’)
THEN
RAISE_APPLICATION_ERROR (-20555, ‘You cannot modify EMP
table during 14:00 to 18:00 Hrs’);
END IF;
END;
```

Assuming that the above trigger is created and the following DML statement is issued at 17:00 hrs.

```
UPDATE emp
Set salary = salary+2500
Where job =‘ANALYST’
```

It will raise an error “You cannot modify EMP table during 14:00 to 18:00 Hrs”

Conditional Predicates in Trigger

- Conditional predicates, which can be used within the trigger body are:
  - INSERTING
  - DELETING
  - UPDATING
- If multiple DML Operations can fire a trigger (e.g., ON INSERT OR DELETE OR UPDATE OF Emp_tab), then the trigger body can make use of conditional predicates to run specific blocks of code.
  - The use depends on the type of statement that fires the trigger.

Example:

Triggers using conditional predicates to apply restrictions on EMP Table

```
CREATE OR REPLACE TRIGGER trig_restrict_emp
BEFORE INSERT OR UPDATE OR DELETE ON EMP
BEGIN
   IF (to_char (sysdate, ‘HH24’) between ’14’ and ’18’)
   then
      IF INSERTING THEN
        RAISE_APPLICATION_ERROR(-20100, ‘You cannot
        insert into EMP table during 14:00 to 18:00
        Hrs‘);
      END IF;
      IF UPDATING
        RAISE_APPLICATION_ERROR(-20101, ‘You cannot
        update emp table  during 14:00 to 18:00 Hrs ‘);
      IF DELETING THEN
         RAISE_APPLICATION_ERROR(-20102, ‘You cannot
         delete from emp table during 14:00 to 16:00
         Hrs‘);
      END IF;
   END IF;
END;
```

For Each Row Triggers

/*Row trigger for the EMP table which makes the entry in ename column in uppercase irrespective of the case in which the user enters the value */

FOR EACH ROW clause is used to create a row trigger

```
CREATE or REPLACE TRIGGER upper_trig
BEFORE INSERT or UPDATE of ename on emp FOR EACH ROW
Begin
   :new.ename :=UPPER(:old.ename);
End;
```

For every row, the row trigger will convert names of every employee into UPPER case

Restricting a Row Trigger

/* Row trigger for EMP table executes only when you insert a new row with job as ANALYST or when updated the salary of an existing ANALYST. The trigger should ensure that if a new row is inserted, commission amount should be zero and if a row is updated then commission should be equal to the old commission plus 10% of the increment. */

```
CREATE OR REPLACE TRIGGER new_emp_restrict_for_analyst
BEFORE INSERT OR UPDATE OF sal ON emp
FOR EACH ROW
When (new.job = 'ANALYST')
BEGIN
              IF INSERTING THEN
                  :new.comm :=0;
              ELSE
                  :new.comm := :old.comm+(:new.sal -
                  :old.salary) * 0.1;
              END IF;
END;
```

INSTEAD OF Triggers

- Tell Oracle what to do instead of performing the actions that caused the trigger to fire
- Trigger an invisible work in the background so the right actions take place
- Can write INSERT, UPDATE, or DELETE statements against a view
- Are useful when the associated view is based on more than one table

Syntax:

```
CREATE [OR REPLACE] TRIGGER <trigger_name> INSTEAD OF
<event1> [OR event2 OR event3]
ON <view_name>
REFERENCING OLD AS old/ NEW AS new]
[FOR EACH ROW]
BEGIN /* PL/SQL Block */
   ….
   ….
END;
```

Example:

/* create a view new_emp_dept */

```
Create view new_emp_dept as
Select empno, ename, emp.deptno, job, sal, dname, loc
From emp, dept
Where emp.deptno=dept.deptno;
```

/* Now suppose when a delete statement is executed for the view specifying a department number, following actions are to be performed simultaneously:
1. Delete all rows from the EMP table for a particular department number
2. Delete the corresponding department information from the DEPT table

```
CREATE OR REPLACE TRIGGER new_emp_dept_remove
INSTEAD OF DELETE on new_emp_dept
FOR EACH ROW
BEGIN
DELETE FROM emp
WHERE deptno = :old.deptno;
DELETE from dept
WHERE deptno = :old.deptno;
End;
```

Managing Triggers

- Triggers can be enabled, disabled, and recompiled.
  - To disable or enable one database trigger:
    ```
    ALTER TRIGGER <trigger_name> DISABLE | ENABLE
    ```
  - To disable or enable or alter all triggers for a table:
    ```
    ALTER TRIGGER <table_name> DISABLE | ENABLE ALL TRIGGERS
    ```
  - To recompile a trigger for a table:
    ```
    ALTER TRIGGER <trigger_name> COMPILE
    ```

Modifying Triggers

- Like a stored procedure, a trigger cannot be explicitly altered. It must be replaced with a new definition.
- When replacing a trigger, you must include the OR REPLACE option in the CREATE TRIGGER statement.
- *Note:*
  - *ALTER TRIGGER statement is used only to recompile, enable, or disable a trigger and not to modify the definition.*
  - *OR REPLACE option is provided to allow a new version of an existing trigger to replace the older version, without affecting any grants made for the original version of the trigger.*

Removing Triggers

- Triggers can be deleted from the system memory.
- They are removed using the DROP TRIGGER command.
- To drop a trigger one must either :
  - Own the trigger or
  - Have the DROP ANY TRIGGER system privilege

Syntax:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER emp_update_chk;
```

Example:

Write a trigger *validate_record* which prevents the record to be entered into StockAnalysis table if the year is not the current year and month is not the current month. Assume that functions get_current_year()andget_current_month()areavailable.

Stock Table is given below:

Year	Month	nav
2008	9	16.00
2008	10	16.50
2008	11	15.40
2009	1	15.40
2009	2	15.10
2009	3	15.10
2009	4	12.40
2009	5	13.40
2009	6	9.40
2009	7	9.00
2009	8	10.20
2009	9	10.40
2009	10	11.00
2009	11	11.40
2009	12	13.00

Example:

-- create a main table and its shadow table

```
CREATE TABLE parts (pnum NUMBER(4), pname VARCHAR2(15));
CREATE   TABLE   parts_log   (pnum   NUMBER(4),   pname
VARCHAR2(15));
```

-- create a trigger that inserts into the shadow table before each insert into the main table

```
CREATE TRIGGER parts_trig
BEFORE INSERT ON parts FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO parts_log VALUES(:new.pnum, :new.pname);
   COMMIT;
END;
```

-- insert a row into the main table, and then commit the insert

```
INSERT INTO parts VALUES (1040, ’Head Gasket’);
COMMIT;
```

-- insert another row, but then roll back the insert

```
INSERT INTO parts VALUES (2075, ’Oil Pan’);
ROLLBACK;
```

-- show that only committed inserts add rows to the main table

```
SELECT * FROM parts ORDER BY pnum;
```

-- show that both committed and rolled-back inserts add rows to the shadow table

```
SELECT * FROM parts_log ORDER BY pnum;
```

Question:

```
CREATE OR REPLACE TRIGGER minimum_age_check
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
   IF ADD_MONTHS (:new.birth_date, 18*12) > SYSDATE
   THEN
      RAISE_APPLICATION_ERROR(-20001, 'Employees must at
      least eighteen years of age.');
   END IF;
END;
```

Indexes

- An index for a database table is similar in concept to a book index.
- The downside of indexes is that when a row is added to the table, additional time is required to update the index for the new row.
- Generally, you should create an index on a column when you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is
  Create an index when a query retrieves <= 10 percent of the total rows in a table.
- This means the column for the index should contain a wide range of values. These types of indexes are called "B-tree" indexes
- An Oracle database automatically creates a B-tree index for the primary key of a table and for columns included in a unique constraint. For columns that contain a small range of values, you can use a "bitmap" index.

Creating a B-Tree Index

Syntax:

```
CREATE [UNIQUE] INDEX index_name ON
table_name(column_name[, column_name ...])
TABLESPACE tab_space;
```

where

- UNIQUE means that the values in the indexed columns must be unique.
- index_name is the name of the index.
- table_name is a database table.
- column_name is the indexed column. You can create an index on multiple columns (such an index is known as a composite index).
- tab_space is the tablespace for the index. If you don't provide a tablespace, the index is stored in the user's default tablespace.

Query to be executed

```
SELECT customer_id, first_name, last_name
FROM customers
WHERE last_name = 'Brown';
```

The following CREATE INDEX statement creates an index named i_customers_last_name on the last_name column of the customers table (I always put i_ at the start of index names):

```
CREATEINDEXi_customers_last_name                      ON
customers(last_name);
```

Once the index has been created, the previous query will take less time to complete. You can enforce uniqueness of column values using a unique index. For example, the following statement creates a unique index named i_customers_phone on the customers.phone column:

```
CREATE     UNIQUE     INDEX     i_customers_phone     ON
customers(phone);
```

You can also create a composite index on multiple columns. For example, the following statement creates a composite index named i_employees_first_last_name on the first_name and last_name columns of the employees table:

```
CREATE     INDEX     i_employees_first_last_name     ON
employees(first_name, last_name);
```

Creating a Function-Based Index

```
SELECT first_name, last_name
FROM customers
WHERE last_name = UPPER('BROWN');
```

Because this query uses a function—UPPER(), in this case—the i_customers_last_name index isn't used. If you want an index to be based on the results of a function, you must create a function-based index, such as:

```
CREATE INDEX i_func_customers_last_name
ON customers(UPPER(last_name));
```

CONNECT system/manager
ALTER SYSTEM SET QUERY_REWRITE_ENABLED=TRUE;

Retrieving Information on Indexes

Customers and employees tables:

```
COLUMN table_name FORMAT a15
COLUMN column_name FORMAT a15
SELECT index_name, table_name, column_name
FROM user_ind_columns
WHERE table_name IN ('CUSTOMERS', 'EMPLOYEES')
ORDER BY index_name;
```

```
INDEX_NAME TABLE_NAME COLUMN_NAME
------------------------------ --------------- ------------
CUSTOMERS_PK CUSTOMERS CUSTOMER_ID
EMPLOYEES_PK EMPLOYEES EMPLOYEE_ID
I_CUSTOMERS_LAST_NAME CUSTOMERS LAST_NAME
```

```
I_CUSTOMERS_PHONE CUSTOMERS PHONE
I_EMPLOYEES_FIRST_LAST_NAME EMPLOYEES LAST_NAME
I_EMPLOYEES_FIRST_LAST_NAME EMPLOYEES FIRST_NAME
I_FUNC_CUSTOMERS_LAST_NAME CUSTOMERS SYS_NC00006$
```

Modifying an Index

You modify an index using ALTER INDEX. The following example renames the i_customers_phone index to i_customers_phone_number:

```
ALTER INDEX i_customers_phone RENAME TO
i_customers_phone_number;
```

Dropping an Index

You drop an index using the DROP INDEX statement. The following example drops the I_customers_phone_number index:

```
DROP INDEX i_customers_phone_number;
```

****