F

Extended Examples

The body of the *SQL Language Reference* contains examples for almost every reference topic. This appendix contains lengthy examples that are not appropriate in the context of a single SQL statement. These examples are intended to provide uninterrupted the series of steps that you would use to take advantage of particular Oracle functionality. They do not replace the syntax diagrams and semantics found for each individual SQL statement in the body of the reference. Use the cross-references provided to access additional information, such as privileges required and restrictions, as well as syntax.

This appendix contains the following sections:

- Using Extensible Indexing
- Using XML in SQL Statements

Using Extensible Indexing

This section provides examples of the steps entailed in a simple but realistic extensible indexing scenario.

Suppose you want to rank the salaries in the `HR.employees` table and then find those that rank between 10 and 20. You could use the `DENSE_RANK` function, as follows:

```
SELECT last_name, salary FROM
   (SELECT last_name, DENSE_RANK() OVER
      (ORDER BY salary DESC) rank_val, salary FROM employees)
   WHERE rank_val BETWEEN 10 AND 20;
```

> **See Also:**
>
> DENSE_RANK

This nested query is somewhat complex, and it requires a full scan of the `employees` table as well as a sort. An alternative would be to use extensible indexing to achieve the same goal. The resulting query will be simpler. The query will require only an index scan and a table access by rowid, and will therefore perform much more efficiently.

The first step is to create the implementation type `position_im`, including method headers for index definition, maintenance, and creation. Most of the type body uses PL/SQL, which is shown in italics.

The type must created with the `AUTHID CURRENT_USER` clause because of the `EXECUTE IMMEDIATE` statement inside the function `ODCIINDEXCREATE()`. By default that function runs with the definer rights. When the function is called in the subsequent creation of the domain index, the invoker does not have the same rights.

> **See Also:**
>
> - CREATE TYPE and CREATE TYPE BODY
> - *Oracle Database Data Cartridge Developer's Guide* for complete information on the ODCI routines in this statement

```
CREATE OR REPLACE TYPE position_im AUTHID CURRENT_USER AS OBJECT
(
  curnum  NUMBER,
  howmany NUMBER,
  lower_bound NUMBER,
  upper_bound NUMBER,
/* lower_bound and upper_bound are used for the
index-based functional implementation */
  STATIC FUNCTION ODCIGETINTERFACES(ifclist OUT SYS.ODCIOBJECTLIST) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXCREATE
    (ia SYS.ODCIINDEXINFO, parms VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXTRUNCATE (ia SYS.ODCIINDEXINFO,
                                     env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXDROP(ia SYS.ODCIINDEXINFO,
                                env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXINSERT(ia SYS.ODCIINDEXINFO, rid ROWID,
                                  newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXDELETE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXUPDATE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXSTART(SCTX IN OUT position_im, ia SYS.ODCIINDEXINFO,
                                 op SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO,
                                 strt NUMBER, stop NUMBER, lower_pos NUMBER,
                                 upper_pos NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  MEMBER FUNCTION ODCIINDEXFETCH(SELF IN OUT position_im, nrows NUMBER,
                                 rids OUT SYS.ODCIRIDLIST, env SYS.ODCIEnv)
                                 RETURN NUMBER,
  MEMBER FUNCTION ODCIINDEXCLOSE(env SYS.ODCIEnv) RETURN NUMBER
);
/

CREATE OR REPLACE TYPE BODY position_im
IS
   STATIC FUNCTION ODCIGETINTERFACES(ifclist OUT SYS.ODCIOBJECTLIST)
       RETURN NUMBER IS
   BEGIN
       ifclist := SYS.ODCIOBJECTLIST(SYS.ODCIOBJECT('SYS','ODCIINDEX2'));
       RETURN ODCICONST.SUCCESS;
   END ODCIGETINTERFACES;
 STATIC FUNCTION ODCIINDEXCREATE (ia SYS.ODCIINDEXINFO, parms VARCHAR2, env SYS.ODCIEnv) RETURN
 NUMBER
  IS
   stmt   VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'Create Table ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
           '_STORAGE_TAB' || '(col_val, base_rowid, constraint pk PRIMARY KEY ' ||
           '(col_val, base_rowid)) ORGANIZATION INDEX AS SELECT ' ||
           ia.INDEXCOLS(1).COLNAME || ', ROWID FROM ' ||
           ia.INDEXCOLS(1).TABLESCHEMA || '.' || ia.INDEXCOLS(1).TABLENAME;
   EXECUTE IMMEDIATE stmt;
```

```
   RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXDROP(ia SYS.ODCIINDEXINFO, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'DROP TABLE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
   '_STORAGE_TAB';
/* Execute the statement */
   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXTRUNCATE(ia SYS.ODCIINDEXINFO, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'TRUNCATE TABLE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME || '_STORAGE_TAB';

   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXINSERT(ia SYS.ODCIINDEXINFO, rid ROWID,
                          newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'INSERT INTO ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
          '_STORAGE_TAB  VALUES (''' || newval || ''' , ''' || rid || ''' )';
/* Execute the SQL statement */
   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;

  STATIC FUNCTION ODCIINDEXDELETE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  env SYS.ODCIEnv)
     RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'DELETE FROM ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
          '_STORAGE_TAB WHERE col_val = ''' || oldval || ''' AND base_rowid = ''' || rid || '''';
/* Execute the statement */
   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXUPDATE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                          newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'UPDATE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
          '_STORAGE_TAB SET col_val = ''' || newval || ''' WHERE f2 = '''|| rid ||'''';
/* Execute the statement */
   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXSTART(SCTX IN OUT position_im, ia SYS.ODCIINDEXINFO,
                          op SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO,
                          strt NUMBER, stop NUMBER, lower_pos NUMBER,
                          upper_pos NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
    rid              VARCHAR2(5072);
    storage_tab_name VARCHAR2(65);
```

```
    lower_bound_stmt VARCHAR2(2000);
    upper_bound_stmt VARCHAR2(2000);
    range_query_stmt VARCHAR2(2000);
    lower_bound      NUMBER;
    upper_bound      NUMBER;
    cnum             INTEGER;
    nrows            INTEGER;

  BEGIN
/* Take care of some error cases.
    The only predicates in which position operator can appear are
       op() = 1     OR
       op() = 0     OR
       op() between 0 and 1
*/
    IF (((strt != 1) AND (strt != 0)) OR
        ((stop != 1) AND (stop != 0)) OR
        ((strt = 1) AND (stop = 0))) THEN
      RAISE_APPLICATION_ERROR(-20101,
                              'incorrect predicate for position_between operator');
    END IF;
    IF (lower_pos > upper_pos) THEN
      RAISE_APPLICATION_ERROR(-20101, 'Upper Position must be greater than or
      equal to Lower Position');
    END IF;
    IF (lower_pos <= 0) THEN
      RAISE_APPLICATION_ERROR(-20101, 'Both Positions must be greater than zero');
    END IF;
    storage_tab_name := ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
                        '_STORAGE_TAB';
    upper_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                        storage_tab_name || ') */ DISTINCT ' ||
                        'col_val FROM ' || storage_tab_name || ' ORDER BY ' ||
                        'col_val DESC) WHERE rownum <= ' || lower_pos;
    EXECUTE IMMEDIATE upper_bound_stmt INTO upper_bound;
    IF (lower_pos != upper_pos) THEN
      lower_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                          storage_tab_name || ') */ DISTINCT ' ||
                          'col_val FROM ' || storage_tab_name ||
                          ' WHERE col_val < ' || upper_bound || ' ORDER BY ' ||
                          'col_val DESC) WHERE rownum <= ' ||
                          (upper_pos - lower_pos);
      EXECUTE IMMEDIATE lower_bound_stmt INTO lower_bound;
    ELSE
      lower_bound := upper_bound;
    END IF;
    IF (lower_bound IS NULL) THEN
      lower_bound := upper_bound;
    END IF;
    range_query_stmt := 'Select base_rowid FROM ' || storage_tab_name ||
                        ' WHERE col_val BETWEEN ' || lower_bound || ' AND ' ||
                        upper_bound;
    cnum := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cnum, range_query_stmt, DBMS_SQL.NATIVE);
/* set context as the cursor number */
    SCTX := position_im(cnum, 0, 0, 0);
/* return success */
    RETURN ODCICONST.SUCCESS;
  END;
  MEMBER FUNCTION ODCIINDEXFETCH(SELF IN OUT position_im, nrows NUMBER,
                                 rids OUT SYS.ODCIRIDLIST, env SYS.ODCIEnv)
   RETURN NUMBER IS
```

```
    cnum    INTEGER;
    rid_tab DBMS_SQL.Varchar2_table;
    rlist   SYS.ODCIRIDLIST := SYS.ODCIRIDLIST();
    i       INTEGER;
    d       INTEGER;
  BEGIN
    cnum := SELF.curnum;
    IF self.howmany = 0 THEN
      dbms_sql.define_array(cnum, 1, rid_tab, nrows, 1);
      d := DBMS_SQL.EXECUTE(cnum);
    END IF;
    d := DBMS_SQL.FETCH_ROWS(cnum);
    IF d = nrows THEN
      rlist.extend(d);
    ELSE
      rlist.extend(d+1);
    END IF;
    DBMS_SQL.COLUMN_VALUE(cnum, 1, rid_tab);
    for i in 1..d loop
      rlist(i) := rid_tab(i+SELF.howmany);
    end loop;
    SELF.howmany := SELF.howmany + d;
    rids := rlist;
    RETURN ODCICONST.SUCCESS;
  END;
  MEMBER FUNCTION ODCIINDEXCLOSE(env SYS.ODCIEnv) RETURN NUMBER IS
    cnum INTEGER;
  BEGIN
    cnum := SELF.curnum;
    DBMS_SQL.CLOSE_CURSOR(cnum);
    RETURN ODCICONST.SUCCESS;
  END;
END;
/
```

The next step is to create the functional implementation `function_for_position_between` for the operator that will be associated with the indextype. (The PL/SQL blocks are shown in parentheses.)

This function is for use with an index-based function evaluation. Therefore, it takes an index context and scan context as parameters.

> **See Also:**
>
> - *Oracle Database Data Cartridge Developer's Guide* for information on creating index-based functional implementation
> - CREATE FUNCTION and *Oracle Database PL/SQL Language Reference*

```
CREATE OR REPLACE FUNCTION function_for_position_between
                           (col NUMBER, lower_pos NUMBER, upper_pos NUMBER,
                            indexctx IN SYS.ODCIIndexCtx,
                            scanctx IN OUT position_im,
                            scanflg IN NUMBER)
RETURN NUMBER AS
  rid              ROWID;
  storage_tab_name VARCHAR2(65);
  lower_bound_stmt VARCHAR2(2000);
```

```
  upper_bound_stmt VARCHAR2(2000);
  col_val_stmt     VARCHAR2(2000);
  lower_bound      NUMBER;
  upper_bound      NUMBER;
  column_value     NUMBER;
BEGIN
  IF (indexctx.IndexInfo IS NOT NULL) THEN
    storage_tab_name := indexctx.IndexInfo.INDEXSCHEMA || '.' ||
                        indexctx.IndexInfo.INDEXNAME || '_STORAGE_TAB';
    IF (scanctx IS NULL) THEN
/* This is the first call. Open a cursor for future calls.
   First, do some error checking
*/
      IF (lower_pos > upper_pos) THEN
        RAISE_APPLICATION_ERROR(-20101,
          'Upper Position must be greater than or equal to Lower Position');
      END IF;
      IF (lower_pos <= 0) THEN
        RAISE_APPLICATION_ERROR(-20101,
          'Both Positions must be greater than zero');
      END IF;
/* Obtain the upper and lower value bounds for the range we're interested in.
*/
      upper_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                          storage_tab_name || ') */ DISTINCT ' ||
                          'col_val FROM ' || storage_tab_name || ' ORDER BY ' ||
                          'col_val DESC) WHERE rownum <= ' || lower_pos;
      EXECUTE IMMEDIATE upper_bound_stmt INTO upper_bound;
      IF (lower_pos != upper_pos) THEN
        lower_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                            storage_tab_name || ') */ DISTINCT ' ||
                            'col_val FROM ' || storage_tab_name ||
                            ' WHERE col_val < ' || upper_bound || ' ORDER BY ' ||
                            'col_val DESC) WHERE rownum <= ' ||
                            (upper_pos - lower_pos);
        EXECUTE IMMEDIATE lower_bound_stmt INTO lower_bound;
      ELSE
        lower_bound := upper_bound;
      END IF;
      IF (lower_bound IS NULL) THEN
        lower_bound := upper_bound;
      END IF;
/* Store the lower and upper bounds for future function invocations for
   the positions.
*/
      scanctx := position_im(0, 0, lower_bound, upper_bound);
    END IF;
/* Fetch the column value corresponding to the rowid, and see if it falls
   within the determined range.
*/
    col_val_stmt := 'Select col_val FROM ' || storage_tab_name ||
                    ' WHERE base_rowid = ''' || indexctx.Rid || '''';
    EXECUTE IMMEDIATE col_val_stmt INTO column_value;
    IF (column_value <= scanctx.upper_bound AND
        column_value >= scanctx.lower_bound AND
        scanflg = ODCICONST.RegularCall) THEN
      RETURN 1;
    ELSE
      RETURN 0;
    END IF;
  ELSE
    RAISE_APPLICATION_ERROR(-20101, 'A column that has a domain index of' ||
```

```
                        'Position indextype must be the first argument');
  END IF;
END;
/
```

Next, create the `position_between` operator, which uses the `function_for_position_between` function. The operator takes an indexed `NUMBER` column as the first argument, followed by a `NUMBER` lower and upper bound as the second and third arguments.

> **See Also:**
>
> CREATE OPERATOR

```
CREATE OR REPLACE OPERATOR position_between
   BINDING (NUMBER, NUMBER, NUMBER) RETURN NUMBER
   WITH INDEX CONTEXT, SCAN CONTEXT position_im
   USING function_for_position_between;
```

In this `CREATE OPERATOR` statement, the `WITH INDEX CONTEXT, SCAN CONTEXT position_im` clause is included so that the index context and scan context are passed in to the functional evaluation, which is index based.

Now create the `position_indextype` indextype for the `position_operator`:

> **See Also:**
>
> CREATE INDEXTYPE

```
CREATE INDEXTYPE position_indextype
   FOR position_between(NUMBER, NUMBER, NUMBER)
   USING position_im;
```

The operator `position_between` uses an index-based functional implementation. Therefore, a domain index must be defined on the referenced column so that the index information can be passed into the functional evaluation. So the final step is to create the domain index `salary_index` using the `position_indextype` indextype:

> **See Also:**
>
> CREATE INDEX

```
CREATE INDEX salary_index ON employees(salary)
   INDEXTYPE IS position_indextype;
```

Now you can use the `position_between` operator function to rewrite the original query as follows:

```
SELECT last_name, salary FROM employees
   WHERE position_between(salary, 10, 20)=1
   ORDER BY salary DESC, last_name;

LAST_NAME                     SALARY
------------------------- ----------
Tucker                         10000
King                           10000
Baer                           10000
Bloom                          10000
Fox                             9600
Bernstein                       9500
Sully                           9500
Greene                          9500
Hunold                          9000
Faviet                          9000
McEwen                          9000
Hall                            9000
Hutton                          8800
Taylor                          8600
Livingston                      8400
Gietz                           8300
Chen                            8200
Fripp                           8200
Weiss                           8000
Olsen                           8000
Smith                           8000
Kaufling                        7900
```

Using XML in SQL Statements

This section describes some of the ways you can use `XMLType` data in the database.

XMLType Tables

The sample schema `oe` contains a table `warehouses`, which contains an `XMLType` column `warehouse_spec`. Suppose you want to create a separate table with the `warehouse_spec` information. The following example creates a very simple `XMLType` table with one `CLOB` column:

```
CREATE TABLE xwarehouses OF XMLTYPE
  XMLTYPE STORE AS CLOB;
```

You can insert into such a table using `XMLType` syntax, as shown in the next statement. (The data inserted in this example corresponds to the data in the `warehouse_spec` column of the sample table `oe.warehouses` where `warehouse_id` = 1.)

```
INSERT INTO xwarehouses VALUES
  (xmltype('<?xml version="1.0"?>
  <Warehouse>
    <WarehouseId>1</WarehouseId>
    <WarehouseName>Southlake, Texas</WarehouseName>
    <Building>Owned</Building>
    <Area>25000</Area>
    <Docks>2</Docks>
    <DockType>Rear load</DockType>
    <WaterAccess>true</WaterAccess>
    <RailAccess>N</RailAccess>
    <Parking>Street</Parking>
    <VClearance>10</VClearance>
  </Warehouse>'));
```

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for information on `XMLType` and its member methods

You can query this table with the following statement:

```
SELECT e.getClobVal() FROM xwarehouses e;
```

`CLOB` columns are subject to all of the restrictions on LOB columns. To avoid these restrictions, create an XMLSchema-based table. The XMLSchema maps the XML elements to their object-relational equivalents. The following example registers an XMLSchema locally. The XMLSchema (`xwarhouses.xsd`) reflects the same structure as the `xwarehouses` table. (XMLSchema declarations use PL/SQL and the `DBMS_XMLSCHEMA` package, so the example is shown in italics.)

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for information on creating XMLSchemas

```
begin
 dbms_xmlschema.registerSchema(
  'http://www.example.com/xwarehouses.xsd',
  '<schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.example.com/xwarehouses.xsd"
       xmlns:who="http://www.example.com/xwarehouses.xsd"
       version="1.0">

  <simpleType name="RentalType">
   <restriction base="string">
    <enumeration value="Rented"/>
    <enumeration value="Owned"/>
   </restriction>
  </simpleType>

  <simpleType name="ParkingType">
   <restriction base="string">
    <enumeration value="Street"/>
    <enumeration value="Lot"/>
   </restriction>
  </simpleType>

  <element name = "Warehouse">
    <complexType>
     <sequence>
      <element name = "WarehouseId"   type = "positiveInteger"/>
      <element name = "WarehouseName" type = "string"/>
      <element name = "Building"      type = "who:RentalType"/>
      <element name = "Area"          type = "positiveInteger"/>
      <element name = "Docks"         type = "positiveInteger"/>
      <element name = "DockType"      type = "string"/>
      <element name = "WaterAccess"   type = "boolean"/>
      <element name = "RailAccess"    type = "boolean"/>
      <element name = "Parking"       type = "who:ParkingType"/>
      <element name = "VClearance"    type = "positiveInteger"/>
```

```
     </sequence>
    </complexType>
  </element>
</schema>',
   TRUE, TRUE, FALSE, FALSE);
end;
/
```

Now you can create an XMLSchema-based table, as shown in the following example:

```
CREATE TABLE xwarehouses OF XMLTYPE
   XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
   ELEMENT "Warehouse";
```

By default, Oracle stores this as an object-relational table. Therefore, you can insert into it as shown in the example that follows. (The data inserted in this example corresponds to the data in the `warehouse_spec` column of the sample table `oe.warehouses` where `warehouse_id` = 1.)

```
INSERT INTO xwarehouses VALUES(   xmltype.createxml('<?xml version="1.0"?>
   <who:Warehouse xmlns:who="http://www.example.com/xwarehouses.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.example.com/xwarehouses.xsd
   http://www.example.com/xwarehouses.xsd">
      <WarehouseId>1</WarehouseId>
      <WarehouseName>Southlake, Texas</WarehouseName>
      <Building>Owned</Building>
      <Area>25000</Area>
      <Docks>2</Docks>
      <DockType>Rear load</DockType>
      <WaterAccess>true</WaterAccess>
      <RailAccess>false</RailAccess>
      <Parking>Street</Parking>
      <VClearance>10</VClearance>
      </who:Warehouse>'));
...
```

You can define constraints on an XMLSchema-based table. To do so, you use the `XMLDATA` pseudocolumn to refer to the appropriate attribute within the `Warehouse` XML element:

```
ALTER TABLE xwarehouses ADD (PRIMARY KEY(XMLDATA."WarehouseId"));
```

Because the data in `xwarehouses` is stored object relationally, Oracle rewrites queries to this `XMLType` table to go to the underlying storage when possible. Therefore the following queries would use the index created by the primary key constraint in the preceding example:

```
SELECT * FROM xwarehouses x
   WHERE EXISTSNODE(VALUE(x), '/Warehouse[WarehouseId="1"]',
   'xmlns:who="http://www.example.com/xwarehouses.xsd"') = 1;

SELECT * FROM xwarehouses x
   WHERE EXTRACTVALUE(VALUE(x), '/Warehouse/WarehouseId',
   'xmlns:who="http://www.example.com/xwarehouses.xsd"') = 1;
```

You can also explicitly create indexes on XMLSchema-based tables, which greatly enhance the performance of subsequent queries. You can create object-relational views on `XMLType` tables, and you can create `XMLType` views on object-relational tables.

> **See Also:**
>
> - XMLDATA Pseudocolumn for information on the `XMLDATA` pseudocolumn
> - "Creating an XMLType View: Example"
> - Creating an Index on an XMLType Table: Example

XMLType Columns

The sample table `oe.warehouses` was created with a `warehouse_spec` column of type `XMLType`. The examples in this section create a shortened form of the `oe.warehouses` table, using two different types of storage.

The first example creates a table with an `XMLType` table stored as a `CLOB`. This table does not require an XMLSchema, so the content structure is not predetermined:

```
CREATE TABLE xwarehouses (
   warehouse_id        NUMBER,
   warehouse_spec      XMLTYPE)
   XMLTYPE warehouse_spec STORE AS CLOB
   (TABLESPACE example
    STORAGE (INITIAL 6144)
    CHUNK 4000
    NOCACHE LOGGING);
```

The following example creates a similar table, but stores the `XMLType` data in an object-relational `XMLType` column whose structure is determined by the specified XMLSchema:

```
CREATE TABLE xwarehouses (
   warehouse_id    NUMBER,
   warehouse_spec  XMLTYPE)
   XMLTYPE warehouse_spec STORE AS OBJECT RELATIONAL
      XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
      ELEMENT "Warehouse";
```