C

XMLIndex Unstructured Component

Unlike a B-tree index, which you define for a specific database column that represents an individual XML element or attribute, or the `XMLIndex` structured component, which applies to specific, structured document parts, the unstructured component of an `XMLIndex` index is, by default, very general.

> **Note:**
>
> Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes. Oracle recommends that you recreate unstructured XML indexes as XML search indexes and use it alongside Transportable Binary XML.

Unless you specify a more narrow focus by detailing specific XPath expressions to use or not to use in indexing, an unstructured `XMLIndex` component applies to *all possible XPath expressions* for your XML data.

The unstructured component of an `XMLIndex` index has three logical parts:

- A **path index** – This indexes the XML *tags* of a document and identifies its various document *fragments*.
- An **order index** – This indexes the hierarchical *positions* of the nodes in an XML document. It keeps track of parent–child, ancestor–descendant, and sibling relations.
- A **value index** – This indexes the *values* of an XML document. It provides lookup by either value equality or value range. A value index is used for values in query predicates (`WHERE` clause).

The unstructured component of an `XMLIndex` index uses a path table and a set of (local) secondary indexes on the path table, which implement the logical parts described above. Two secondary indexes are created automatically:

- A **pikey index**, which implements the logical indexes for both path and order.
- A real **value index**, which implements the logical value index.

You can modify these two indexes or create additional secondary indexes. The path table and its secondary indexes are all owned by the owner of the base table upon which the `XMLIndex` index is created.

The pikey index handles paths and order relationships together, which gives the best performance in most cases. If you find in some particular case that the value index is not picked up when think it should be, you can replace the pikey index with separate indexes for the paths and order relationships. Such (optional) indexes are called **path id** and **order key** indexes, respectively. For best results, contact Oracle Support if you find that the pikey index is not sufficient for your needs in some case.

The path table contains one row for each indexed node in the XML document. For each indexed node, the **path table** stores:

- The corresponding *rowid* of the table that stores the document.

- A *locator*, which provides fast access to the corresponding document fragment. For binary XML storage of XML schema-based data, it also stores data-type information.
- An *order key*, to record the hierarchical position of the node in the document. You can think of this as a Dewey decimal key like that used in library cataloging and Internet protocol SNMP. In such a system, the key `3.21.5` represents the node position of the fifth child of the twenty-first child of the third child of the document root node.
- An identifier that represents an XPath *path* to the node.
- The effective *text value* of the node.

Table C-1 shows the main information[1] that is in the path table.

Table C-1 XMLIndex Path Table

Column	Data Type	Description
`PATHID`	`RAW(8)`	Unique identifier for the XPath path to the node.
`RID`	`ROWID`	Rowid of the table used to store the XML data.
`ORDER_KEY`	`RAW(1000)`	Decimal order key that identifies the hierarchical position of the node. (Document ordering is preserved.)
`LOCATOR`	`RAW(2000)`	Fragment-location information. Used for fragment extraction. For binary XML storage of XML schema-based data, data-type information is also stored here.
`VALUE`	`VARCHAR2(4000)`	Effective text value the node.

Tasks Involving XMLIndex Indexes with an Unstructured Component identifies the documentation for some user tasks involving `XMLIndex` indexes that have an *unstructured* component.

Table C-2 Tasks Involving XMLIndex Indexes with an Unstructured Component

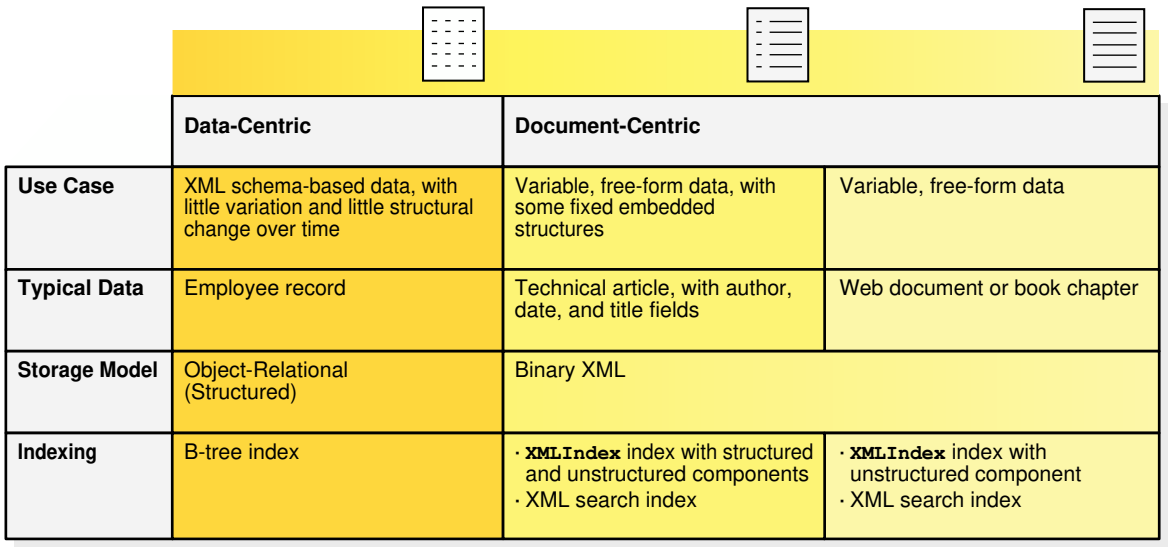
For information about how to...	See...
Create an `XMLIndex` index with an unstructured component	Example C-2, Example C-4, Example C-17, Example C-19, Example 6-17, Example 6-18, Example C-15
Drop the unstructured component of an `XMLIndex` index (drop the path table)	Example C-5
Name the path table when creating an `XMLIndex` index	Example C-2
Specify storage options when creating an `XMLIndex` index	Example C-4
Show all existing secondary indexes on an `XMLIndex` path table	Example C-6, Example C-14
Obtain the name of a path table for an `XMLIndex` index	Example C-3
Obtain the name of an XMLIndex index with an unstructured component, given its path table	Example C-7
Create a secondary index on an `XMLIndex` path table	Using XMLIndex with an Unstructured Component
Obtain information about all of the secondary indexes on an `XMLIndex` path table	Example C-14
Create a function-based index on a path-table `VALUE` column	Example C-9

[1] The actual path table implementation may be slightly different.

Table C-2 (Cont.) Tasks Involving XMLIndex Indexes with an Unstructured Component

For information about how to...	See...
Create a numeric index on a path-table `VALUE` column	Example C-11
Create a date index on a path-table `VALUE` column	Example C-12
Create an Oracle Text `CONTEXT` index on a path-table `VALUE` column	Example C-13
Exclude or include particular XPath expressions from use by an `XMLIndex` index	XMLIndex Path Subsetting: Specifying the Paths You Want to Index
Specify namespace prefixes for XPath expressions used for `XMLIndex`	XMLIndex Path Subsetting: Specifying the Paths You Want to Index
Exclude or include particular XPath expressions from use by an `XMLIndex` index	XMLIndex Path Subsetting: Specifying the Paths You Want to Index
Specify namespace prefixes for XPath expressions used for `XMLIndex`	XMLIndex Path Subsetting: Specifying the Paths You Want to Index

Figure C-1 XML Use Cases and XML Indexing



If you need to support ad-hoc XML queries that involve predicates, then you can use `XMLIndex` with an *unstructured component* – see XMLIndex Unstructured Component.

The pikey index uses path table columns `PATHID`, `RID`, and `ORDER_KEY` to represent the path and order indexes. An optional path id index uses columns `PATHID` and `RID` to represent the path index. A value index is an index on the `VALUE` column.

Example C-1 explores the contents of the path table for two purchase-order documents.

Example C-1 Path Table Contents for Two Purchase Orders

```
<PurchaseOrder>
 <Reference>SBELL-20021009123336O1PDT</Reference>
 <Actions>
  <Action>
   <User>SVOLLMAN</User>
```

```
  </Action>
 </Actions>
 . . .
</PurchaseOrder>

<PurchaseOrder>
 <Reference>ABEL-20021127121040897PST</Reference>
 <Actions>
  <Action>
   <User>ZLOTKEY</User>
  </Action>
  <Action>
   <User>KING</User>
  </Action>
 </Actions>
 . . .
</PurchaseOrder>
```

An `XMLIndex` index on an `XMLType` table or column storing these purchase orders includes a path table that has one row for each indexed node in the XML documents. Suppose that the system assigns the following `PATHID`s when indexing the nodes according to their XPath expressions:

PATHID	Indexed XPath
1	/PurchaseOrder
2	/PurchaseOrder/Reference
3	/PurchaseOrder/Actions
4	/PurchaseOrder/Actions/Action
5	/PurchaseOrder/Actions/Action/User

The resulting path table would then be something like this (column `LOCATOR` is not shown):

PATHID	RID	ORDER_KEY	VALUE
1	R1	1	SBELL-2002100912333601PDTSVOLLMAN
2	R1	1.1	SBELL-2002100912333601PDT
3	R1	1.2	SVOLLMAN
4	R1	1.2.1	SVOLLMAN
5	R1	1.2.1.1	SVOLLMAN
1	R2	1	ABEL-20021127121040897PSTZLOTKEYKING
2	R2	1.1	ABEL-20021127121040897PST
3	R2	1.2	ZLOTKEYKING
4	R2	1.2.1	ZLOTKEY
5	R2	1.2.1.1	ZLOTKEY
4	R2	1.2.2	KING
5	R2	1.2.2.1	KING

Guidelines for Using XMLIndex with an Unstructured Component

There are several guidelines that can help you use `XMLIndex` with an unstructured component.

These guidelines are applicable only when the two alternatives discussed return the same result set.

- Avoid prefixing `//` with ancestor elements. For example, use `//c`, *not* **`/a/b`**`//c`, provided these return the same result set.
- Avoid prefixing `/*` with ancestor elements. For example, use `/*/*/*`, *not* **`/a`**`/*/*`, provided these return the same result set.

- In a `WHERE` clause, use `XMLExists` rather than `XMLCast` of `XMLQuery`. This can allow optimization that, in effect, invokes a subquery against the path-table `VALUE` column. For example, use this:

```
SELECT count(*) FROM purchaseorder p
  WHERE
    XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="715515011020"]'
              PASSING OBJECT_VALUE AS "p");
```

  Do not use this:

```
SELECT count(*) FROM purchaseorder p
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/LineItems/LineItem/Part/@Id'
                         PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
                  AS VARCHAR2(14))
        = "715515011020";
```

- When possible, use `count(*)`, *not* `count(XMLCast(XMLQuery(...))`, in a `SELECT` clause. For example, if you know that a `LineItem` element in a purchase-order document has only one `Description` child, use this:

```
SELECT count(*) FROM po_binxml, XMLTable('//LineItem'
  PASSING OBJECT_VALUE);
```

  Do not use this:

```
SELECT count(li.value)
 FROM po_binxml p,
      XMLTable('//LineItem' PASSING p.OBJECT_VALUE
               COLUMNS value VARCHAR2(30) PATH 'Description') li;
```

- Reduce the number of XPath expressions used in a query `FROM` list as much as possible. For example, use this:

```
SELECT li.description
  FROM po_binxml p,
       XMLTable(
         'PurchaseOrder/LineItems/LineItem' PASSING p.OBJECT_VALUE
         COLUMNS description VARCHAR2(256) PATH 'Description') li;
```

  Do not use this:

```
SELECT li.description
  FROM po_binxml p,
       XMLTable('PurchaseOrder/LineItems' PASSING p.OBJECT_VALUE) ls,
       XMLTable('LineItems/LineItem'      PASSING ls.OBJECT_VALUE
                COLUMNS description VARCHAR2(256)
                                    PATH 'Description') li;
```

- If you use an XPath expression in a query to drill down inside a virtual table (created, for example, using SQL/XML function `XMLTable`), then create a secondary index on the order key of the path table using Oracle SQL function `sys_orderkey_depth`. Here is an example

of such a query; the selection navigates to element `Description` inside virtual line-item table `li`.

```
SELECT li.description
  FROM po_binxml p,
       XMLTable(
         'PurchaseOrder/LineItems/LineItem' PASSING p.OBJECT_VALUE
         COLUMNS description VARCHAR2(256) PATH 'Description') li;
```

Such queries are evaluated using function `sys_orderkey_depth`, which returns the depth of the order-key value. Because the order index uses two columns, the index needed is a *composite* index over columns `ORDER_KEY` and `RID`, as well as over function `sys_orderkey_depth` applied to the `ORDER_KEY` value. For example:

```
CREATE INDEX depth_ix ON my_path_table
  (RID, sys_orderkey_depth(ORDER_KEY), ORDER_KEY);
```

See also Example C-8.

Ignore the Path Table – It Is Transparent

Though you can create secondary indexes on path-table columns, you can generally ignore the path table itself.

You *cannot access* the path table, other than to `DESCRIBE` it and create (secondary) indexes on it. You need never explicitly gather statistics on the path table. You need only collect statistics on the `XMLIndex` index or the base table on which the `XMLIndex` index is defined; statistics are collected and maintained on the path table and its secondary indexes transparently.

Related Topics

- Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer
  The Oracle Database cost-based optimizer determines how to most cost-effectively evaluate a given query, including which indexes, if any, to use. For it to be able to do this accurately, you must collect statistics on various database objects.

Column VALUE of an XMLIndex Path Table

A secondary index on column `VALUE` is used with XPath expressions in a `WHERE` clause that have predicates involving string matches. For example:

```
/PurchaseOrder[Reference/text() = "SBELL-20021009123336O1PDT"]
```

Column `VALUE` stores the **effective text value** of an element or an attribute node — comments and processing instructions are ignored during indexing.

- For an *attribute*, the effective text value is the attribute value.
- For a *simple* element (an element that has no children), the effective text value is the concatenation of all of the text nodes of the element.
- For a *complex* element (an element that has children), the effective text value is the concatenation of (1) the text nodes of the element itself and (2) the effective text values of all of its simple-element descendants. (This is a recursive definition.)

The effective text value is limited (truncated), however, to 4000 bytes for a simple element or attribute and to 80 bytes for a complex element.

Column `VALUE` is a fixed size, `VARCHAR2(4000)`. Any overflow (beyond 4000 bytes) during index creation or update is truncated.

In addition to the 4000-byte limit for column `VALUE`, there is a limit on the size of a key for the secondary index created on this column. This is the case for B-tree and function-based indexes as well; it is not an `XMLIndex` limitation. The index-key size limit is a function of the block size for your database. It is this limit that determines how much of `VALUE` is indexed.

Thus, only the first 4000 bytes of the effective text value are stored in column `VALUE`, and only the first *N* bytes of column `VALUE` are indexed, where *N* is the index-key size limit (*N* < 4000). Because of the index-key size limit, the index on column `VALUE` acts only as a *preliminary filter* for the effective text value.

For example, suppose that your database block size requires that the `VALUE` index be no larger than 800 bytes, so that only the first 800 bytes of the effective text value is indexed. The first 800 bytes of the effective text value is first tested, using `XMLIndex`, and only if that text prefix matches the query value is the rest of the effective text value tested.

The secondary index on column `VALUE` is an index on SQL function `substr` (substring equality), because that function is used to test the text prefix. This function-based index is created automatically as part of the implementation of `XMLIndex` for column `VALUE`.

For example, the XPath expression `/PurchaseOrder[Reference/text() = :1]` in a query `WHERE` clause might, in effect, be rewritten to a test something like this:

```
substr(VALUE, 1 800) = substr(:1, 1, 800) AND VALUE = :1;
```

This conjunction contains two parts, which are processed from left to right. The first test uses the index on function `substr` as a preliminary filter, to eliminate text whose first 800 bytes do not match the first 800 bytes of the value of bind variable `:1`.

Only the first test uses an index — the full value of column `VALUE` is not indexed. After preliminary filtering by the first test, the second test checks the entire effective text value — that is, the full value of column `VALUE` — for full equality with the value of `:1`. This check does not use an index.

Even if only the first 800 bytes of text is indexed, it is important for query performance that up to 4000 bytes be stored in column `VALUE`, because that provides quick, direct access to the data, instead of requiring, for example, extracting it from deep within a `CLOB`-instance XML document. If the effective text value is greater than 4000 bytes, then the second test in the `WHERE`-clause conjunction requires accessing the base-table data.

Neither the `VALUE` column 4000-byte limit nor the index-key size affect query results in any way; they can affect only performance.

> **Note:**
>
> Because of the possibility of the `VALUE` column being truncated, an Oracle Text `CONTEXT` index created on the `VALUE` column might return incorrect results.

As mentioned, `XMLIndex` can be used with XML schema-based data. If an XML schema specifies a `defaultValue` value for a given element or attribute, and a particular document

does not specify a value for that element or attribute, then the `defaultValue` value is used for the `VALUE` column.

Secondary Indexes on Column VALUE

Even if you do not specify a secondary index for column `VALUE` when you create an `XMLIndex` index, a default secondary index is created on column `VALUE`. This default index has the default properties — in particular, it is an index for *text* (string-valued) data only.

You can, however, create a `VALUE` index of a different type. For example, you can create a number-valued index if that is appropriate for many of your queries. You can create multiple secondary indexes on the `VALUE` column. An index of a particular type is used only when it is appropriate. For example, a number-valued index is used only when the `VALUE` column is a number; it is ignored for other values. Secondary indexes on path-table columns are treated like any other secondary indexes — you can alter them, drop them, mark them unusable, and so on.

See Also:

- Using XMLIndex with an Unstructured Component for examples of creating secondary indexes on column `VALUE`
- PARAMETERS Clause for CREATE INDEX and ALTER INDEX for the syntax of the `PARAMETERS` clause

XPath Expressions That Are Not Indexed by an XMLIndex Unstructured Component

A few types of XPath expressions are *not* indexed by `XMLIndex`.

- Applications of XPath functions. In particular, user-defined XPath functions are *not* indexed.
- Axes other than `child`, `descendant`, and `attribute`, that is, axes `parent`, `ancestor`, `following-sibling`, `preceding-sibling`, `following`, `preceding`, and `ancestor-or-self`.
- Expressions using the union operator, `|` (vertical bar).

Using XMLIndex with an Unstructured Component

You can perform various operations on an `XMLIndex` index that has an unstructured component, including manipulating the path table and the secondary indexes of that component.

To include an unstructured component in an `XMLIndex` index, you can use a *`path_table_clause`* in the `PARAMETERS` clause when you create or modify the `XMLIndex` index — see *path_table_clause ::=*.

If you do not specify a *structured* component, then the index will have an unstructured component, even if you do not specify the path table. It is however generally a good idea to

specify the path table, so that it has a recognizable, user-oriented name that you can refer to in other XMLIndex operations.

Example C-2 shows how to name the path table ("my_path_table") when creating an XMLIndex index with an unstructured component.

If you do not name the path table then its name is generated by the system, using the index name you provide to CREATE INDEX as a base. Example C-3 shows this for the XMLIndex index created in Example 6-6.

By default, the storage options of a path table and its secondary indexes are derived from the storage properties of the base table on which the XMLIndex index is created. You can specify different storage options by using a PARAMETERS clause when you create the index, as shown in Example C-4. The PARAMETERS clause of CREATE INDEX (and ALTER INDEX) must be between single quotation marks (').

Because XMLIndex is a logical *domain* index, not a physical index, all physical attributes are either zero (0) or NULL.

If an XMLIndex index has both an unstructured and a structured component, then you can use ALTER INDEX to drop the unstructured component. To do this, you drop the path table. Example C-5 illustrates this. (This assumes that you also have a structured component — Example 6-11 results in an index with both structured and unstructured components.)

In addition to specifying storage options for the path table, Example C-4 names the secondary indexes on the path table.

Like the name of the path table, the names of the secondary indexes on the path-table columns are generated automatically using the index name as a base, unless you specify them in the PARAMETERS clause. Example C-6 illustrates this, and shows how you can determine these names using public view USER_IND_COLUMNS. It also shows that the pikey index uses three columns.

> **See Also:**
>
> Example C-14 for a similar, but more complex example

Example C-2 Naming the Path Table of an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('PATH TABLE my_path_table');
```

Example C-3 Determining the System-Generated Name of an XMLIndex Path Table

```
SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
  WHERE TABLE_NAME = 'PO_BINXML' AND INDEX_NAME = 'PO_XMLINDEX_IX';

PATH_TABLE_NAME
------------------------------
SYS67567_PO_XMLINDE_PATH_TABLE

1 row selected.
```

Example C-4 Specifying Storage Options When Creating an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS
```

```
('PATH TABLE po_path_table
  (PCTFREE 5 PCTUSED 90 INITRANS 5
   STORAGE (INITIAL 1k NEXT 2k MINEXTENTS 3 BUFFER_POOL KEEP)
   NOLOGGING ENABLE ROW MOVEMENT PARALLEL 3)
  PIKEY INDEX po_pikey_ix (LOGGING PCTFREE 1 INITRANS 3)
  VALUE INDEX po_value_ix (LOGGING PCTFREE 1 INITRANS 3)');
```

Example C-5 Dropping an XMLIndex Unstructured Component

```
ALTER INDEX po_xmlindex_ix PARAMETERS('DROP PATH TABLE');
```

Example C-6 Determining the Names of the Secondary Indexes of an XMLIndex Index

```
SELECT INDEX_NAME, COLUMN_NAME, COLUMN_POSITION FROM USER_IND_COLUMNS
  WHERE TABLE_NAME IN (SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
                       WHERE INDEX_NAME = 'PO_XMLINDEX_IX')
  ORDER BY INDEX_NAME, COLUMN_NAME;

INDEX_NAME                     COLUMN_NAME  COLUMN_POSITION
------------------------------ ------------ ---------------
SYS67563_PO_XMLINDE_PIKEY_IX   ORDER_KEY                  3
SYS67563_PO_XMLINDE_PIKEY_IX   PATHID                     2
SYS67563_PO_XMLINDE_PIKEY_IX   RID                        1
SYS67563_PO_XMLINDE_VALUE_IX   SYS_NC00006$               1

4 rows selected.
```

- Creating Additional Secondary Indexes on an XMLIndex Path Table
  You can add extra secondary indexes to an `XMLIndex` unstructured component.

Related Topics

- PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX
  The syntax for the `PARAMETERS` clause for `CREATE INDEX` and `ALTER INDEX` is defined.

Creating Additional Secondary Indexes on an XMLIndex Path Table

You can add extra secondary indexes to an `XMLIndex` unstructured component.

Examples Example C-9, Example C-11, Example C-12, and Example C-13 add extra secondary indexes to the `XMLIndex` index created in Example C-4.

You can create any number of additional secondary indexes on the `VALUE` column of the path table of an `XMLIndex` index. These can be of different types, including function-based indexes and Oracle Text indexes.

Whether or not a given index is used for a given element occurrence when processing a query is determined by whether it is of the appropriate type for that value and whether it is cost-effective to use it.

Example C-9 creates a function-based index on column `VALUE` of the path table using SQL function `substr`. This might be useful if your queries often use `substr` applied to the text nodes of XML elements.

If you have many elements whose text nodes represent numeric values, then it can make sense to create a numeric index on the column `VALUE`. However, doing so directly, in a manner analogous to Example C-9, raises an ORA-01722 error (invalid number) if some of the element values are *not* numbers. This is illustrated in Example C-10.

What is needed is an index that is used for numeric-valued elements but is ignored for element occurrences that do not have numeric values. Procedure `createNumberIndex` of package

DBMS_XMLINDEX exists specifically for this purpose. You pass it the names of the database schema, the XMLIndex index, and the numeric index to be created. Creation of a numeric index is illustrated in Example C-11.

Because such an index is specifically designed to ignore elements that do not have numeric values, its use does not detect their presence. If there are non-numeric elements and, for whatever reason, the XMLIndex index is not used in some query, then an ORA-01722 error is raised. However, if the index is used, no such error is raised, because the index ignores non-numeric data. As always, the use of an index never changes the result set — it never gives you different results, but use of an index can prevent you from detecting erroneous data.

Creating a date-valued index is similar to creating a numeric index; you use procedure DBMS_XMLINDEX.createDateIndex. Example C-12 shows this .

Example C-13 creates an Oracle Text CONTEXT index on column VALUE. This is useful for full-text queries on text values of XML elements. If a CONTEXT index is defined on column VALUE, then it is used during predicate evaluation. An Oracle Text index is independent of all other VALUE-column indexes.

The query in Example C-14 shows all of the secondary indexes created on the path table of an XMLIndex index. The indexes created explicitly are in bold. Note in particular that some indexes, such as the function-based index created on column VALUE, do not appear as such; the column name listed for such an index is a system-generated name such as SYS_NC00007$. You *cannot* see these columns by executing a query with COLUMN_NAME = 'VALUE' in the WHERE clause.

To know whether a particular XMLIndex index has been used in resolving a query, you can examine an execution plan for the query.

Similar to XMLIndex with Structured Component, it is at query compile time that Oracle Database determines whether or not a given XMLIndex index can be used, that is, whether the query can be rewritten into a query against the index.

For an unstructured XMLIndex component, if it cannot be determined at compile time that an XPath expression in the query is a subset of the paths you specified to be used for XMLIndex indexing, then the unstructured component of the index is not used.

You can examine the execution plan for a query to see whether a particular XMLIndex index has been used in resolving the query.

If the unstructured component of the index is used, then its path table, order key, or path id is referenced in the execution plan. The execution plan does not directly indicate that a domain index was used; it does not refer to the XMLIndex index by name. See Example C-8.

Given the name of a path table from an execution plan such as this, you can obtain the name of its XMLIndex index as shown in Example C-7

The unstructured component of an XMLIndex can be used for XPath expressions in the SELECT list, the FROM list, and the WHERE clause of a query, and it is useful for SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast. Unlike function-based indexes, which are deprecated for XMLType, XMLIndex indexes can be used when you extract data from an XML fragment in a document.

See Also:

- Column VALUE of an XMLIndex Path Table for information about the possibility of an Oracle Text `CONTEXT` index created on the `VALUE` column returning incorrect results
- *Oracle Text Reference* for information about `CREATE INDEX` parameter `TRANSACTIONAL`
- *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL procedures `createNumberIndex` and `createDateIndex` in package `DBMS_XMLINDEX`

Example C-7 Obtaining the Name of an XMLIndex Index from Its Path-Table Name

```
SELECT INDEX_NAME FROM USER_XML_INDEXES
  WHERE PATH_TABLE_NAME = 'MY_PATH_TABLE';

INDEX_NAME
------------------------------
PO_XMLINDEX_IX

1 row selected.
```

Example C-8 Extracting Data from an XML Fragment Using XMLIndex

```
SET AUTOTRACE ON EXPLAIN


SELECT li.description, li.itemno
  FROM po_binxml, XMLTable('/PurchaseOrder/LineItems/LineItem'
                           PASSING OBJECT_VALUE
                           COLUMNS "DESCRIPTION" VARCHAR(40) PATH 'Description',
                                   "ITEMNO"      INTEGER     PATH '@ItemNumber') li
  WHERE XMLExists('/PurchaseOrder[Reference="SBELL-20021009123336O1PDT"]'
                  PASSING OBJECT_VALUE);

DESCRIPTION                                ITEMNO
---------------------------------------- ----------
A Night to Remember                               1
The Unbearable Lightness Of Being                 2
Sisters                                           3

3 rows selected.


Execution Plan

----------------------------------------------------------------------------------------------
----------------------
| Id  | Operation                          | Name                           | Rows  | Bytes
|Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
----------------------
|   0 | SELECT STATEMENT                   |                                |     1 |  1546
|   30   (4)|00:00:01 |
|*  1 |  FILTER                            |                                |       |
|           |          |
|*  2 |   TABLE ACCESS BY INDEX ROWID      | MY_PATH_TABLE                  |     1 |  3524
```

```
|    3   (0)|00:00:01 |
|*  3 |    INDEX RANGE SCAN               | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |
|    2   (0)|00:00:01 |
|*  4 |  FILTER                           |                              |       |
|           |         |
|*  5 |   TABLE ACCESS BY INDEX ROWID     | MY_PATH_TABLE                |     1 |  3524
|    3   (0)|00:00:01 |
|*  6 |    INDEX RANGE SCAN               | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |
|    2   (0)|00:00:01 |
|   7 |  NESTED LOOPS                     |                              |       |
|           |         |
|   8 |   NESTED LOOPS                    |                              |     1 |  1546
|   30   (4)|00:00:01 |
|   9 |    NESTED LOOPS                   |                              |     1 |    24
|   28   (4)|00:00:01 |
|  10 |     VIEW                          | VW_SQ_1                      |     1 |    12
|   26   (0)|00:00:01 |
|  11 |      HASH UNIQUE                  |                              |     1 |  5046
|           |         |
|  12 |       NESTED LOOPS                |                              |     1 |  5046
|   26   (0)|00:00:01 |
|* 13 |        TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                |     1 |  3524
|   24   (0)|00:00:01 |
|* 14 |         INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_VALUE_IX |    73 |
|    1   (0)|00:00:01 |
|* 15 |        TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                |     1 |  1522
|    2   (0)|00:00:01 |
|* 16 |         INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |
|    1   (0)|00:00:01 |
|  17 |     TABLE ACCESS BY USER ROWID    | PO_BINXML                    |     1 |    12
|    1   (0)|00:00:01 |
|* 18 |    INDEX RANGE SCAN               | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |
|    1   (0)|00:00:01 |
|* 19 |   TABLE ACCESS BY INDEX ROWID     | MY_PATH_TABLE                |     1 |  1522
|    2   (0)|00:00:01 |
-------------------------------------------------------------------------------------------
-----------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(:B1<SYS_ORDERKEY_MAXCHILD(:B2))
   2 - filter(SYS_XMLI_LOC_ISNODE("SYS_P2"."LOCATOR")=1)
   3 - access("SYS_P2"."RID"=:B1 AND "SYS_P2"."PATHID"=HEXTORAW('28EC')  AND
"SYS_P2"."ORDER_KEY">:B2 AND
              "SYS_P2"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD(:B3))
       filter(SYS_ORDERKEY_DEPTH("SYS_P2"."ORDER_KEY")=SYS_ORDERKEY_DEPTH(:B1)+1)
   4 - filter(:B1<SYS_ORDERKEY_MAXCHILD(:B2))
   5 - filter(SYS_XMLI_LOC_ISNODE("SYS_P5"."LOCATOR")=1)
   6 - access("SYS_P5"."RID"=:B1 AND "SYS_P5"."PATHID"=HEXTORAW('60E0')  AND
"SYS_P5"."ORDER_KEY">:B2 AND
              "SYS_P5"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD(:B3))
       filter(SYS_ORDERKEY_DEPTH("SYS_P5"."ORDER_KEY")=SYS_ORDERKEY_DEPTH(:B1)+1)
  13 - filter("SYS_P10"."VALUE"='SBELL-20021009123336O1PDT' AND
"SYS_P10"."PATHID"=HEXTORAW('4F8C')  AND
              SYS_XMLI_LOC_ISNODE("SYS_P10"."LOCATOR")=1)
  14 - access(SUBSTRB("VALUE",1,1599)='SBELL-20021009123336O1PDT')
  15 - filter(SYS_XMLI_LOC_ISNODE("SYS_P8"."LOCATOR")=1)
  16 - access("SYS_P10"."RID"="SYS_P8"."RID" AND "SYS_P8"."PATHID"=HEXTORAW('4E36')  AND
              "SYS_P8"."ORDER_KEY"<"SYS_P10"."ORDER_KEY")
       filter("SYS_P10"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD("SYS_P8"."ORDER_KEY") AND
              SYS_ORDERKEY_DEPTH("SYS_P8"."ORDER_KEY")
```

```
+1=SYS_ORDERKEY_DEPTH("SYS_P10"."ORDER_KEY"))
  18 - access("PO_BINXML".ROWID="SYS_ALIAS_4"."RID" AND
"SYS_ALIAS_4"."PATHID"=HEXTORAW('3748') )
  19 - filter(SYS_XMLI_LOC_ISNODE("SYS_ALIAS_4"."LOCATOR")=1)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Example C-9 Creating a Function-Based Index on Path-Table Column VALUE

```
CREATE INDEX fn_based_ix ON po_path_table (substr(VALUE, 1, 100));
```

Example C-10 Trying to Create a Numeric Index on Path-Table Column VALUE Directly

```
CREATE INDEX direct_num_ix ON po_path_table (to_binary_double(VALUE));
CREATE INDEX direct_num_ix ON po_path_table (to_binary_double(VALUE))
                                            *
ERROR at line 1:
ORA-01722: invalid number
```

Example C-11 Creating a Numeric Index on Column VALUE with Procedure createNumberIndex

```
CALL DBMS_XMLINDEX.createNumberIndex('OE', 'PO_XMLINDEX_IX', 'API_NUM_IX');
```

Example C-12 Creating a Date Index on Column VALUE with Procedure createDateIndex

```
CALL DBMS_XMLINDEX.createDateIndex('OE', 'PO_XMLINDEX_IX', 'API_DATE_IX',
                                   'dateTime');
```

Example C-13 Creating an Oracle Text CONTEXT Index on Path-Table Column VALUE

```
CREATE INDEX po_otext_ix ON po_path_table (VALUE)
  INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS('TRANSACTIONAL');
```

Example C-14 Showing All Secondary Indexes on an XMLIndex Path Table

```
SELECT c.INDEX_NAME, c.COLUMN_NAME, c.COLUMN_POSITION, e.COLUMN_EXPRESSION
  FROM USER_IND_COLUMNS c LEFT OUTER JOIN USER_IND_EXPRESSIONS e
    ON (c.INDEX_NAME = e.INDEX_NAME)
  WHERE c.TABLE_NAME IN (SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
                         WHERE INDEX_NAME = 'PO_XMLINDEX_IX')
  ORDER BY c.INDEX_NAME, c.COLUMN_NAME;

INDEX_NAME           COLUMN_NAME  COLUMN_POSITION COLUMN_EXPRESSION
-------------------- ------------ --------------- ----------------------
API_DATE_IX          SYS_NC00009$               1 SYS_EXTRACT_UTC(SYS_XMLCONV("V
                                                  ALUE",3,8,0,0,181))
API_NUM_IX           SYS_NC00008$               1 TO_BINARY_DOUBLE("VALUE")
FN_BASED_IX          SYS_NC00007$               1 SUBSTR("VALUE",1,100)
PO_OTEXT_IX          VALUE                      1
PO_PIKEY_IX          ORDER_KEY                  3
PO_PIKEY_IX          PATHID                     2
PO_PIKEY_IX          RID                        1
PO_VALUE_IX          SYS_NC00006$               1 SUBSTRB("VALUE",1,1599)

8 rows selected.
```

Related Topics

- Indexing XML Data for Full-Text Queries (pre-23ai)
  When you need full-text search over XML data, Oracle recommends that you store your `XMLType` data as binary XML and you use XQuery Full Text (XQFT). You use an XML search index for this. This is the topic of this section.

Asynchronous (Deferred) Maintenance of XMLIndex Indexes

You can defer the cost of maintaining an `XMLIndex` index that has *only* an *unstructured* component, performing maintenance only at commit time or when database load is reduced. This can improve DML performance, and it can enable bulk loading of unsynchronized index rows when an index is synchronized.

This feature applies to an `XMLIndex` index that has *only* an *unstructured* component. If you specify asynchronous maintenance for an `XMLIndex` index that has a *structured* component (even if it also has an unstructured component), then an error is raised.

By default, `XMLIndex` indexing is updated (maintained) at each DML operation, so that it remains in sync with the base table. In some situations, you might not require this, and using possibly stale indexes might be acceptable. In that use case, you can decide to defer the cost of index maintenance, performing at commit time only or at some time when database load is reduced. This can improve DML performance. It can also improve index maintenance performance by enabling bulk loading of unsynchronized index rows when an index is synchronized.

Using a stale index has no effect, other than performance, on DML operations. It can have an effect on query results, however: If the index is not up-to-date at query time, then the query results might not be up-to-date either. Even if only one column of a base table is of data type `XMLType`, all queries on that table reflect the database data as of the last synchronization of the `XMLIndex` index on the `XMLType` column.

You can specify index maintenance deferment using the parameters clause of a `CREATE INDEX` or `ALTER INDEX` statement.

Be aware that even if you defer synchronization for an `XMLIndex` index, the following database operations automatically synchronize the index:

- Any DDL operation on the index – `ALTER INDEX` or creation of secondary indexes
- Any DDL operation on the base table – `ALTER TABLE` or creation of another index

Table C-3 lists the synchronization options and the `ASYNC` clause syntax you use to specify them. The `ASYNC` clause is used in the `PARAMETERS` clause of a `CREATE INDEX` or `ALTER INDEX` statement for `XMLIndex`.

Table C-3 Index Synchronization

When to Synchronize	ASYNC Clause Syntax
Always	`ASYNC (SYNC ALWAYS)` This is the default behavior. You can specify it explicitly, to cancel a previous `ASYNC` specification.
Upon commit	`ASYNC (SYNC ON COMMIT)`

Table C-3 (Cont.) Index Synchronization

When to Synchronize	ASYNC Clause Syntax
Periodically	`ASYNC (SYNC EVERY "repeat_interval")` *repeat_interval* is the same as for the calendaring syntax of `DBMS_SCHEDULER` To use `EVERY`, you must have the `CREATE JOB` privilege.
Manually, on demand	`ASYNC (SYNC MANUAL)` You can manually synchronize the index using PL/SQL procedure `DBMS_XMLINDEX.syncIndex`.

Optional `ASYNC` syntax parameter `STALE` is intended for possible future use; you need never specify it explicitly. It has value `FALSE` whenever `ALWAYS` is used; otherwise it has value `TRUE`. Specifying an explicit `STALE` value that contradicts this rule raises an error.

Example C-15 creates an XMLIndex index that is synchronized every Monday at 3:00 pm, starting tomorrow.

Example C-16 manually synchronizes the index created in Example C-15.

When `XMLIndex` index synchronization is deferred, all DML changes (inserts, updates, and deletions) made to the base table since the last index synchronization are recorded in a pending table, one row per DML operation. The name of this table is the value of column `PEND_TABLE_NAME` of static public views `USER_XML_INDEXES`, `ALL_XML_INDEXES`, and `DBA_XML_INDEXES`.

You can examine this table to determine when synchronization might be appropriate for a given `XMLIndex` index. The more rows there are in the pending table, the more the index is likely to be in need of synchronization.

If the pending table is large, then setting parameter `REINDEX` to `TRUE` when calling `syncIndex`, as in Example C-16, can improve performance. When `REINDEX` is `TRUE`, all of the secondary indexes are dropped and then re-created after the pending table data is bulk-loaded.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference*, section "Calendaring Syntax", for the syntax of *repeat_interval*
> - *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL procedure `DBMS_XMLINDEX.syncIndex`

Example C-15 Specifying Deferred Synchronization for XMLIndex

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('ASYNC (SYNC EVERY "FREQ=HOURLY; INTERVAL = 1")');
```

Example C-16 Manually Synchronizing an XMLIndex Index Using SYNCINDEX

```
EXEC DBMS_XMLINDEX.syncIndex('OE', 'PO_XMLINDEX_IX', REINDEX => TRUE);
```

- Syncing an XMLIndex Index in Case of Error ORA-08181
  If a query raises error ORA-08181, check whether the base `XMLType` table of the query has an `XMLIndex` index with an unstructured component. If so, then manually synchronize the `XMLIndex` index using `DBMS_XMLINDEX.syncIndex`.

Syncing an XMLIndex Index in Case of Error ORA-08181

If a query raises error ORA-08181, check whether the base `XMLType` table of the query has an `XMLIndex` index with an unstructured component. If so, then manually synchronize the `XMLIndex` index using `DBMS_XMLINDEX.syncIndex`.

This applies only if error ORA-08181 is raised in the following situation:

1. In a pluggable database, *PDB1*, you created an `XMLType` table or column *XTABCOL*, which you indexed using an `XMLIndex` index that has an unstructured component.
2. You plugged *PDB1* into a container database.
3. You cloned *PDB1* to a new pluggable database, *PDB2*.
4. Error ORA-08181 is raised when you query *XTABCOL* in *PDB2*.

If the error is raised even after synchronizing then seek another cause. Error ORA-08181 is a general error that can be raised in various situations, of which this is only one.

Related Topics

- Oracle XML DB and Database Consolidation
  Each pluggable database has its own Oracle XML DB Repository, and its own Oracle XML DB configuration file, `xdbconfig.xml`.

Advantages of Unstructured XMLIndex

B-tree indexes can be used advantageously with object-relational `XMLType` storage — they provide sharp focus by targeting the underlying objects directly. They are generally ineffective, however, in addressing the detailed structure (elements and attributes) of an XML document stored using binary XML. That is the special domain of `XMLIndex`.

`XMLIndex` is a *domain* index; it is designed specifically for the domain of XML data. It is a *logical* index. An `XMLIndex` index can be used for SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast`.

XMLIndex presents the following advantages over other indexing methods:

- An `XMLIndex` index with an unstructured component can speed access to both `SELECT` list data and `FROM` list data, making it useful for XML *fragment* extraction, in particular. Function-based indexes, which are deprecated, cannot be used to extract document fragments.
- You need no prior knowledge of the XPath expressions that might be used in queries. The unstructured component of an `XMLIndex` index can be completely general. This is not the case for function-based indexes.

Data Dictionary Static Public Views Related to Unstructured XMLIndex

Data Dictionary views reporting information about Unstructured XMLIndex indexes are shared with Structured XMLIndex indexes through public views `USER_XML_INDEXES`, `ALL_XML_INDEXES`, and `DBA_XML_INDEXES`.

Similar to Structured XMLIndex, statistics information is shared across multiple views.

When querying `USER_TAB_STATISTICS`, `ALL_TAB_STATISTICS`, `DBA_TAB_STATISTICS`, statistics over the Path-Table can be queried by filtering over the `TABLE_NAME` column using the Path-Table name.

> **See Also:**
>
> Data Dictionary Static Public Views Related to XMLIndex

XMLIndex Path Subsetting: Specifying the Paths You Want to Index

If you know which XPath expressions you are most likely to query then you can narrow the focus of `XMLIndex` indexing and thus improve performance.

One of the advantages of an `XMLIndex` index with an unstructured component is that it is very general: you need not specify which XPath locations to index; you need no prior knowledge of the XPath expressions that will be queried. By default, an unstructured `XMLIndex` component indexes all possible XPath locations in your XML data.

However, if you are aware of the XPath expressions that you are most likely to query, then you can narrow the focus of `XMLIndex` indexing and thus improve performance. Having fewer indexed nodes means less space is required for indexing, which improves index maintenance during DML operations. Having fewer indexed nodes improves DDL performance, and having a smaller path table improves query performance.

You narrow the focus of indexing by pruning the set of XPath expressions (paths) corresponding to XML fragments to be indexed, specifying a subset of all possible paths. You can do this in two alternative ways:

- Exclusion – Start with the default behavior of including all possible XPath expressions, and exclude some of them from indexing.
- Inclusion – Start with an empty set of XPath expressions to be used in indexing, and add paths to this inclusion set.

You can specify path subsetting either when you create an `XMLIndex` index using `CREATE INDEX` or when you modify it using `ALTER INDEX`. In both cases, you provide the subsetting information in the `PATHS` parameter of the statement's `PARAMETERS` clause. For exclusion, you use keyword `EXCLUDE`. For inclusion, you use keyword `INCLUDE` for `ALTER INDEX` and no keyword for `CREATE INDEX` (list the paths to include). You can also specify namespace mappings for the nodes targeted by the `PATHS` parameter.

For `ALTER INDEX`, keyword `INCLUDE` or `EXCLUDE` is followed by keyword `ADD` or `REMOVE`, to indicate whether the list of paths that follows the keyword is to be added or removed from the inclusion or exclusion list. For example, this statement adds path `/PurchaseOrder/Reference` to the list of paths to be excluded from indexing:

```
ALTER INDEX po_xmlindex_ix REBUILD
  PARAMETERS ('PATHS (EXCLUDE ADD (/PurchaseOrder/Reference))');
```

To alter an `XMLIndex` index so that it *includes all* possible paths, use keyword `INDEX_ALL_PATHS`. See *alter_index_paths_clause ::=*.

> **Note:**
>
> If you create an `XMLIndex` index that has both structured and unstructured components, then, by default, any nodes indexed in the structured component are also indexed in the unstructured component; that is, they are *not* automatically *excluded* from the unstructured component. If you do not want unstructured `XMLIndex` indexing to apply to them, then you must explicitly use path subsetting to exclude them.

- Examples of XMLIndex Path Subsetting
  Some examples are presented of defining `XMLIndex` indexes on subsets of XPath expressions.
- XMLIndex Path-Subsetting Rules
  Rules that apply to `XMLIndex` path subsetting are described.

Related Topics

- PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX
  The syntax for the `PARAMETERS` clause for `CREATE INDEX` and `ALTER INDEX` is defined.

Examples of XMLIndex Path Subsetting

Some examples are presented of defining `XMLIndex` indexes on subsets of XPath expressions.

Example C-17 XMLIndex Path Subsetting with CREATE INDEX

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE)
  INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('PATHS (INCLUDE (/PurchaseOrder/LineItems//*
                               /PurchaseOrder/Reference))');
```

This statement creates an index that indexes only top-level element `PurchaseOrder` and some of its children, as follows:

- All `LineItems` elements and their descendants
- All `Reference` elements

It does that by including the specified paths, starting with an empty set of paths to be used for the index.

Example C-18 XMLIndex Path Subsetting with ALTER INDEX

```
ALTER INDEX po_xmlindex_ix REBUILD
  PARAMETERS ('PATHS (INCLUDE ADD (/PurchaseOrder/Requestor
                                   /PurchaseOrder/Actions/Action//*))');
```

This statement adds two more paths to those used for indexing. These paths index element `Requestor` and descendants of element `Action` (and their ancestors).

Example C-19 XMLIndex Path Subsetting Using a Namespace Prefix

If an XPath expression to be used for `XMLIndex` indexing uses namespace prefixes, you can use a `NAMESPACE MAPPING` clause to the `PATHS` list, to specify those prefixes. Here is an example:

```
CREATE INDEX po_xmlindex_ix ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('PATHS (INCLUDE (/PurchaseOrder/LineItems//*   /PurchaseOrder/ipo:Reference)
                    NAMESPACE MAPPING (xmlns="http://xmlns.oracle.com"
                                       xmlns:ipo="http://xmlns.oracle.com/ipo"))');
```

XMLIndex Path-Subsetting Rules

Rules that apply to `XMLIndex` path subsetting are described.

- The paths must reference only child and descendant axes, and they must test only element and attribute nodes or their names (possibly using wildcards). In particular, the paths must not involve predicates.
- You cannot specify both path exclusion and path inclusion; choose one or the other.
- If an index was created using path exclusion (inclusion), then you can modify it using only path exclusion (inclusion) — index modification must either further restrict or further extend the path subset. For example, you cannot create an index that includes certain paths and subsequently modify it to exclude certain paths.

PARAMETERS Clause for CREATE INDEX and ALTER INDEX in Unstructured Index

- Usage of PATHS Clause
  Certain considerations apply to using the `PATHS` clause.
- Usage of create_index_paths_clause and alter_index_paths_clause
  Certain considerations apply to using *`create_index_paths_clause`* and *`alter_index_paths_clause`*.
- Usage of pikey_clause, path_id_clause, and order_key_clause
  Syntactically, each of the clauses `pikey_clause`, `path_id_clause`, and `order_key_clause` is optional. A pikey index is created even if you do not specify a `pikey_clause`. To create a path id index or an order-key index, you must specify a `path_id_clause` or an `order_key_clause`, respectively.
- Usage of value_clause
  Certain considerations apply to using *`value_clause`*.
- Usage of async_clause
  Certain considerations apply to using the `ASYNC` clause.

Usage of PATHS Clause

Certain considerations apply to using the `PATHS` clause.

- There can be at most one `PATHS` clause in a `CREATE INDEX` statement. That is, there can be at most one occurrence of `PATHS` followed by *`create_index_paths_clause`*.

- Clause *create_index_paths_clause* is used only with CREATE INDEX; *alter_index_paths_clause* is used only with ALTER INDEX.

Usage of create_index_paths_clause and alter_index_paths_clause

Certain considerations apply to using *create_index_paths_clause* and *alter_index_paths_clause*.

- The INDEX_ALL_PATHS keyword rebuilds the index to include all paths. This keyword is available only for *alter_index_paths_clause*, not *create_index_paths_clause*.
- An explicit list of paths to index can include wildcards and //.
- *XPaths_list* is a list of one or more XPath expressions, each of which includes only child axis, descendant axis, name test, and wildcard (*) constructs.
- If *XPaths_list* is omitted from *create_index_paths_clause*, all paths are indexed.
- For each unique namespace prefix that is used in an XPath expression in *XPaths_list*, a standard XML *namespace* declaration is needed, to provide the corresponding namespace information.
- You can change an index in ways that are not reflected directly in the syntax by dropping it and then creating it again as needed. For example, to change an index that was defined by including paths to one that is defined by excluding paths, drop it and then create it using EXCLUDE.

Usage of pikey_clause, path_id_clause, and order_key_clause

Syntactically, each of the clauses pikey_clause, path_id_clause, and order_key_clause is optional. A pikey index is created even if you do not specify a pikey_clause. To create a path id index or an order-key index, you must specify a path_id_clause or an order_key_clause, respectively.

Usage of value_clause

Certain considerations apply to using *value_clause*.

- Column VALUE is created as VARCHAR2(4000).
- If clause *value_clause* consists only of the keyword VALUE, then the value index is created with the usual default attributes.
- If clause *path_id_clause* consists only of the keywords PATH ID, then the path-id index is created with the usual default attributes.
- If clause *order_key_clause* consists only of the keywords ORDER KEY, then the order-key index is created with the usual default attributes.

Usage of async_clause

Certain considerations apply to using the ASYNC clause.

- Use this feature only with an XMLIndex index that has *only* an *unstructured* component. If you specify an ASYNC clause for an XMLIndex index that has a *structured* component, then an error is raised.
- ALWAYS means automatic synchronization occurs for each DML statement.

- `MANUAL` means no automatic synchronization occurs. You must manually synchronize the index using `DBMS_XMLINDEX.syncIndex`.
- `EVERY` *`repeat_interval`* means automatically synchronize the index at interval *`repeat_interval`*. The syntax of *`repeat_interval`* is the same as that for PL/SQL package `DBMS_SCHEDULER`, and it must be enclosed in double quotation marks (`"`). To use `EVERY` you must have the `CREATE JOB` privilege.
- `ON COMMIT` means synchronize the index immediately after a commit operation. The commit does not return until the synchronization is complete. Since the synchronization is performed as a separate transaction, there can be a short period when the data is committed but index changes are not yet committed.
- `STALE` is optional. A value of `TRUE` means that query results might be stale; a value of `FALSE` means that query results are always up-to-date. The default value, and the only permitted explicitly specified value, is as follows.
  - For `ALWAYS`, `STALE` is `FALSE`.
  - For any other `ASYNC` option besides `ALWAYS`, `STALE` is `TRUE`.