6

Indexes for XMLType Data

You can create indexes on your XML data, to focus on particular parts of it that you query often and thus improve performance. There are various ways that you can index `XMLType` data, whether it is XML schema-based or non-schema-based, and regardless of the `XMLType` storage model you use.

> **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 Unstructured XML Indexes is deprecated in 23ai and superseded by XML search indexes.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.
>
> The execution plans shown here are for illustration only. If you run the examples presented here in your environment then your execution plans might not be identical.

- Oracle XML DB Tasks Involving Indexes
  Common tasks involving indexes for XML data are described.
- Overview of Indexing XMLType Data
- XMLIndex
- XML Search Index: Indexing for Full Text Search and Ad-hoc Queries
  When you need full-text search, or range-search capabilities over large, unstructured documents stored as Transportable Binary XML, Oracle recommends that you use XML Search Index to index the XML data.
- Indexing XMLType Data Stored Object-Relationally
  You can effectively index `XMLType` data that is stored object-relationally by creating B-tree indexes on the underlying database columns that correspond to XML nodes.

> **See Also:**
>
> - *Oracle Database Concepts* for an overview of indexing
> - *Oracle Database Development Guide* for information about using indexes in application development

Oracle XML DB Tasks Involving Indexes

Common tasks involving indexes for XML data are described.

Table 6-1 identifies the documentation for some basic user tasks involving indexes for XML data.

Table 6-1 Basic XML Indexing Tasks

For information about how to...	See...
Choose an indexing approach	Overview of Indexing XMLType Data
Create, drop, or rename an `XMLIndex` index	Example 6-6, Example 6-8
Obtain the name of an `XMLIndex` index for a given table or column	Example 6-7
Determine whether a given `XMLIndex` index is used in evaluating a query	How to Tell Whether XMLIndex is Used
Turn off use of an `XMLIndex` index	Turning Off Use of XMLIndex
Creating and Using an XML Search Index	Creating and Using an XML Search Index
Queries using an XML Search Index	Queries using an XML Search Index
Index `XMLType` data stored object-relationally	Indexing XMLType Data Stored Object-Relationally, Guideline: Create indexes on ordered collection tables

Table 6-2 identifies the documentation for some user tasks involving `XMLIndex` indexes that have a *structured* component.

Table 6-2 Tasks Involving XMLIndex Indexes with a Structured Component

For information about how to...	See...
Create an `XMLIndex` index with a structured component	Example 6-11, Example 6-9
Drop the structured component of an `XMLIndex` index (drop all structure groups)	Example 6-13
Ensure data type correspondence between a query and an `XMLIndex` index with a structured component	Data Type Considerations for XMLIndex Structured Component
Create a B-tree index on a content table of an `XMLIndex` structured component	Example 6-14
Create an Oracle Text `CONTEXT` index on a content table of an `XMLIndex` structured component	Example D-6

Table 6-3 identifies the documentation for some other user tasks involving `XMLIndex` indexes.

Table 6-3 Tasks involving XML Search Index

For information about ...	See...
How to create and use an XML Search Index	Creating and Using an XML Search Index
How to maintain XML Search Indexes	Maintenance of XML Search Indexes
Default Preferences and Recommendations	Preference Defaults and Recommendations
How to query using an XML Search Index	Queries using an XML Search Index
Index Migration	Migrating to Use XML Search Index

Overview of Indexing XMLType Data

If your XML data contains islands of structured, predictable data, and your application only projects values from these islands of structured content:

- Use `XMLIndex` with a *structured component* to index the structured islands.
- A structured index component reflects the queries you use. You can change this set of known queries over time, provided you update the index definition accordingly. See XMLIndex Structured Component.

In addition, if you need to query content using full-text search within your XML data:

- Use an *XML search index*. See Queries using an XML Search Index.

Does your XML data contain *islands of data that is highly structured and predictable* ?

- If Yes: Use `XMLIndex` with a structured component to index the islands. See XMLIndex Structured Component.
- If No: Do you need to support *ad-hoc XML queries that involve predicates*?
  - If Yes: Use XML Search Index which can be customized to support both Full Text Search and Range-Search Predicates.
  - If No: Do not index your XML data.

> **Note:**
>
> XMLIndex and XML Search Index can both be created over the same XMLType column. See section Queries using an XML Search Index for more information about index usage during queries.

XMLIndex

> **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.

- Advantages of 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 either transportable binary XML or binary XML. That is the special domain of `XMLIndex`.
- XMLIndex Components
  XMLIndex is used to index XML data that may have some structure. It applies to XMLType data that is stored as both transportable binary XML and binary XML.
- XMLIndex Structured Component
  You create and use the structured component of an `XMLIndex` index for queries that project fixed, structured islands of XML content, even if the surrounding data is relatively unstructured.

- Creating, Dropping, Altering, and Examining an XMLIndex Index
  Basic operations on an `XMLIndex` index include creating it, dropping it, altering it, and examining it. Examples are presented.
- Use of XMLIndex with a Structured Component
  An `XMLIndex` structured component indexes specific islands of structure in your XML data.
- How to Tell Whether XMLIndex is Used
  To know whether a particular `XMLIndex` index has been used in resolving a query, you can examine an execution plan for the query.
- Turning Off Use of XMLIndex
  You can turn off the use of `XMLIndex` by using optimizer hint: `/*+ NO_XML_QUERY_REWRITE */` or optimizer hint `/*+ NO_XMLINDEX_REWRITE */`.
- Guidelines for Using XMLIndex with a Structured Component
  There are several guidelines that can help you use `XMLIndex` with a structured component.
- XMLIndex Partitioning and Parallelism
  If you partition an `XMLType` table, or a table with an `XMLType` column, using range, list, or hash partitioning, you can also create an `XMLIndex` index on the table. You can optionally ensure that index creation and maintenance are carried out in parallel.
- 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.
- Data Dictionary Static Public Views Related to XMLIndex
  Information about the standard database indexes is available in static public views `USER_INDEXES`, `ALL_INDEXES`, and `DBA_INDEXES`. Similar information about `XMLIndex` indexes is available in static public views `USER_XML_INDEXES`, `ALL_XML_INDEXES`, and `DBA_XML_INDEXES`.
- PARAMETERS Clause for CREATE INDEX and ALTER INDEX
  Creation or modification of an `XMLIndex` index often involves the use of a `PARAMETERS` clause with SQL statement `CREATE INDEX` or `ALTER INDEX`. You can use it to specify index characteristics in detail.

Advantages of 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 either transportable binary XML or 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 is effective in any part of a query; it is not limited to use in a `WHERE` clause. This is not the case for any of the other kinds of indexes you might use with XML data.
- You can use an `XMLIndex` index with either XML schema-based or non-schema-based XMLType data stored as binary XML. B-tree indexing is appropriate only for XML schema-based data that is stored object-relationally.

- You can use an `XMLIndex` index for searches with XPath expressions that target *collections*, that is, nodes that occur multiple times within a document. This is not the case for function-based indexes.
- If you have prior knowledge of the XPath expressions to be used in queries, then you can improve performance by using a *structured* `XMLIndex` component that targets fixed, structured islands of data that are queried often.
- `XMLIndex` indexing — both index creation and index maintenance — can be carried out in parallel, using multiple database processes. This is not the case for function-based indexes, which are deprecated.

XMLIndex Components

XMLIndex is used to index XML data that may have some structure. It applies to XMLType data that is stored as both transportable binary XML and binary XML.

> **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.

Semi-structured XML data can sometimes nevertheless contain islands of predictable, structured data. An `XMLIndex` index can therefore have two components: a **structured component**, used to index such islands, and an **unstructured component**, used to index data that has little or variable structure.

A structured component can help with queries that project and use islands of structured content. A typical example is a free-form specification with fixed fields author, date, and title. An unstructured component can help with queries that extract XML fragments. Either component can be omitted from a given `XMLIndex` index.

Unlike a structured component, an unstructured component is general and relatively untargeted. It is appropriate for general indexing of document-centric XML data. A typical example is an XML web document or a book chapter.

You can create an `XMLIndex` index with both structured and unstructured components. A typical use case is supporting queries that extract an XML fragment from a document whenever some structured data is also present. The unstructured component is used for the fragment extraction. The structured component is used for a query predicate that checks for the structured data (for example, in the SQL `WHERE` clause).

Though you can restrict an unstructured component to apply only to certain XPath subsets, its path table indexes node content that can be of different scalar types, which can require you to create multiple secondary indexes on the `VALUE` column to deal with the different data types — see Secondary Indexes on Column VALUE. Using an unstructured component alone can also lead to inefficiencies involving multiple probes and self-joins of its path table, for queries that project structured islands.

On the other hand, a structured component is not suited for queries that involve little structure or queries that extract XML fragments. Use a structured component to index structured islands of data; use an unstructured component to index data that has little structure.

The last row indicates the applicability of `XMLIndex` for different XML data use cases. It shows that `XMLIndex` is appropriate for semi-structured XML data, however it is stored (last two columns). And an `XMLIndex` index with a structured component is useful for document-centric data that contains structured islands.

Table 6-4 XML Use Cases and XML Indexing

	Data Centric	Document Centric
Use Case	XML schema-based data, with little variation and little structural change over time	• Variable, free-form data, with some fixed embedded structures • Variable, free-form data
Typical Data	Employee Record	• Technical article, with author, date, and title fields • Web document or book chapter
Storage Model	Object-Relational(Structured)	Transportable Binary XML
Indexing	B-Tree Index	• XMLIndex index with structured components • XML search index

Related Topics

- XMLIndex Structured Component
  You create and use the structured component of an `XMLIndex` index for queries that project fixed, structured islands of XML content, even if the surrounding data is relatively unstructured.
- 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.

> **See Also:**
>
> Advantages of XMLIndex for a summary of the advantages provided by each `XMLIndex` component type

XMLIndex Structured Component

You create and use the structured component of an `XMLIndex` index for queries that project fixed, structured islands of XML content, even if the surrounding data is relatively unstructured.

A structured `XMLIndex` component organizes such islands in a *relational* format. In this it is similar to SQL/XML *function* `XMLTable`, and the syntax you use to define the structured component reflects this similarity. The relational tables used to store the indexing data are data-type aware, and each column can be of a different scalar data type.

You can thus think of the act of creating the structured component of an `XMLIndex` index as *decomposing* a structured portion of your XML data into relational format. This differs from the object-relational storage model of `XMLType` in these ways:

- A structured index component *explicitly* decomposes particular *portions* of your data, which you specify — portions that you commonly query. Object-relational `XMLType` storage involves automatic decomposition of an entire `XMLType` table or column.
- The structured component of an `XMLIndex` index applies to both XML schema-based and non-schema-based data. Object-relational `XMLType` storage applies only to data that is based on an XML schema.
- The decomposed data for a structured `XMLIndex` component is stored in addition to the `XMLType` data, as an index, rather than being the storage model for the `XMLType` data itself.
- For a structured `XMLIndex` component, the same data can be projected multiple times, as columns of different data type.

The index content tables used for the structured component of an `XMLIndex` index are part of the index, but because they are normal relational tables you can, in turn, *index* them using any standard relational indexes, including indexes that satisfy primary-key and foreign-key constraints. You can also index them using domain indexes, such as an Oracle Text `CONTEXT` index.

Another way to look at the structured component of an `XMLIndex` index sees that it acts as a *generalized function-based index*. A function-based index is similar to a structured `XMLIndex` component that has only one relational column.

If you find that for a particular application you are creating multiple function-based indexes, then consider using an `XMLIndex` index with a structured component instead. Create also B-tree indexes on the columns of the structured index component.

> **Note:**
>
> - Queries that use SQL/XML function `XMLTable` can typically be automatically rewritten to use the relational indexing tables of an `XMLIndex` structured component. In particular, SQL `ORDER BY`, `GROUP BY`, and window constructs operating on columns of an `XMLTable` virtual table are rewritten to the same constructs operating on the real columns of the relational indexing tables of the structured `XMLIndex` component.
>
>   The relational tables used for `XMLIndex` structured indexing also contain some internal, system-defined columns. These internal columns might change in the future, so do not write code that depends on any assumptions about their existence or contents.
> - Queries that use Oracle SQL function `XMLSequence` within a SQL `TABLE` collection expression, that is, `TABLE (XMLSequence(...))`, are *not* rewritten to use the indexing tables of an `XMLIndex` structured component. Oracle SQL function `XMLSequence` is *deprecated* in Oracle Database 11g Release 2; use standard SQL/XML function `XMLTable` instead.
>
>   See *Oracle Database SQL Language Reference* for information about the SQL `TABLE` collection expression.

- Ignore the Index Content Tables; They Are Transparent
  Although the index content tables of an `XMLIndex` structured component are normal relational tables, they are also *read-only*: you cannot add or drop their columns or modify (insert, update, or delete) their rows.

- Data Type Considerations for XMLIndex Structured Component
  The relational tables that are used for an `XMLIndex` structured component use SQL data types. XQuery expressions that are used in queries use XML data types (XML Schema data types and XQuery data types).
- Exchange Partitioning and XMLIndex
  In exchange partitioning, you exchange a table with a partition of another table. The first table must have the same structure as the partition of the second table, with which it is to be exchanged. The two tables must also be similar with respect to indexing with an `XMLIndex` index.

Related Topics

- Use of XMLIndex with a Structured Component
  An `XMLIndex` structured component indexes specific islands of structure in your XML data.
- SQL/XML Functions XMLQUERY, XMLTABLE, XMLExists, and XMLCast
  SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast` are defined by the SQL/XML standard as a general interface between the SQL and XQuery languages.

Ignore the Index Content Tables; They Are Transparent

Although the index content tables of an `XMLIndex` structured component are normal relational tables, they are also *read-only*: you cannot add or drop their columns or modify (insert, update, or delete) their rows.

You can thus generally ignore the relational index content tables. You cannot access them, other than to `DESCRIBE` them and create (secondary) indexes on them. You need never explicitly gather statistics on them. You need only collect statistics on the `XMLIndex` index itself or the base table on which the `XMLIndex` index is defined; statistics are collected and maintained on the index content tables 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.

Data Type Considerations for XMLIndex Structured Component

The relational tables that are used for an `XMLIndex` structured component use SQL data types. XQuery expressions that are used in queries use XML data types (XML Schema data types and XQuery data types).

XQuery typing rules can automatically change the data type of a subexpression, to ensure coherence and type-checking. For example, if a document that is queried using XPath expression `/PurchaseOrder/LineItem[@ItemNumber = 25]` is not XML schema-based, then the subexpression `@ItemNumber` is untyped, and it is then automatically cast to `xs:double` by the XQuery `=` comparison operator. To index this data using an `XMLIndex` structured component you must use `BINARY_DOUBLE` as the SQL data type.

This is a general rule. For an `XMLIndex` index with structured component to apply to a query, the data types must correspond. Table 6-5 shows the data-type correspondences.

Table 6-5 XML and SQL Data Type Correspondence for XMLIndex

XML Data Type	SQL Data Type
xs:decimal	INTEGER or NUMBER
xs:double	BINARY_DOUBLE
xs:float	BINARY_FLOAT
xs:date	DATE, TIMESTAMP WITH TIMEZONE
xs:dateTime	TIMESTAMP, TIMESTAMP WITH TIMEZONE
xs:dayTimeDuration	INTERVAL DAY TO SECOND
xs:yearMonthDuration	INTERVAL YEAR TO MONTH

Note:

If the XML data type is `xs:date` or `xs:dateTime`, and if you know that the data that you will query and for which you are creating an index will *not* contain a time-zone component, then you can increase performance by using SQL data type `DATE` or `TIMESTAMP`. If the data might contain a time-zone component, then you must use SQL data type `TIMESTAMP WITH TIMEZONE`.

If the XML and SQL data types involved do not have a built-in one-to-one correspondence, then you must make them correspond (according to Table 6-5), in order for the index to be picked up for your query. There are two ways you can do this:

- **Make the index correspond to the query –** Define (or redefine) the column in the structured index component, so that it corresponds to the XML data type. For example, if a query that you want to index uses the XML data type `xs:double`, then define the index to use the corresponding SQL data type, `BINARY_DOUBLE`.
- **Make the query correspond to the index –** In your query, explicitly cast the relevant parts of an XQuery expression to data types that correspond to the SQL data types used in the index content table.

Example 6-1 and Example 6-2 show how you can cast an XQuery expression in your query to match the SQL data type used in the index content table.

Notice that the number `25` plays a different role in these two examples, even though in both cases it is the purchase-order item number. In Example 6-1, `25` is a SQL number of data type `INTEGER`; in Example 6-2, `25` is an XQuery number of data type `xs:decimal`.

In Example 6-1, the `XMLQuery` result is cast to SQL type `INTEGER`, which is compared with the SQL value 25. In Example 6-2, the value of attribute `ItemNumber` is cast (in XQuery) to the XML data type `xs:decimal`, which is compared with the XQuery value 25 and which corresponds to the SQL data type (`INTEGER`) used for the index. There are thus two different kinds of data-type conversion in these examples, but they both convert query data to make it type-compatible with the index content table.

> **See Also:**
>
> Use DBMS_XMLSCHEMA to Map XML Schema Data Types to SQL Data Types for information about the built-in correspondence between XML Schema data types and SQL data types

Example 6-1 Making Query Data Compatible with Index Data – SQL Cast

```
SELECT count(*) FROM purchaseorder
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/LineItem/@ItemNumber'
                         PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
                AS INTEGER)
        = 25;
```

Example 6-2 Making Query Data Compatible with Index Data – XQuery Cast

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItem[xs:decimal(@ItemNumber) = 25]'
                  PASSING OBJECT_VALUE AS "p");
```

Exchange Partitioning and XMLIndex

In exchange partitioning, you exchange a table with a partition of another table. The first table must have the same structure as the partition of the second table, with which it is to be exchanged. The two tables must also be similar with respect to indexing with an `XMLIndex` index.

One of the following must be true:

- Neither table has an `XMLIndex` index.
- Both have an `XMLIndex` index, and one of the following is true:
  - Neither index has a structured component.
  - Both indexes have a structured component.

If none of those conditions holds then you *cannot* perform exchange partitioning.

If both tables have an `XMLIndex` index with a structured component then in the general case you must perform some preprocessing before invoking `ALTER TABLE EXCHANGE PARTITION`, and you must perform some postprocessing after invoking it. Otherwise, the exchange-partition operation raises an error.

You use PL/SQL procedures **`exchangePreProc`** and **`exchangePostProc`** in package `DBMS_XMLSTORAGE_MANAGE` to perform this preprocessing and postprocessing, as illustrated in Example 6-3. Each of the `XMLType` tables there, `table` and `exchange_table`, has an `XMLIndex` index that has a structured component.

In the special case of *reference*-partitioned tables there are foreign-key constraints involved, so things are a bit more complex. In this case, you use PL/SQL procedure **`refPartitionExchangeIn`** or **`refPartitionExchangeOut`**, to load data into (exchange-in) or out of (exchange-out) the partitioned tables, respectively.

Example 6-4 illustrates this, loading data from exchange tables `parent_ex` and `child_ex` into base tables `parent` and `child`. Example 6-5 shows the table and index definitions.

> **See Also:**
>
> - *Oracle Database SQL Language Reference*
> - *Oracle Database Data Cartridge Developer's Guide* for general information about using `ALTER TABLE EXCHANGE PARTITION` with tables that have domain indexes (`XMLIndex` is a domain index)
> - *Oracle Database PL/SQL Packages and Types Reference* for information about procedures `exchangePreProc`, `exchangePostProc`, `refPartitionExchangeIn`, and `refPartitionExchangeIOut` in package `DBMS_XMLSTORAGE_MANAGE`.

Example 6-3 Exchange-Partitioning Tables That Have an XMLIndex Structured Component

```
EXEC DBMS_XMLSTORAGE_MANAGE.exchangePreProc(USER, 'table');
EXEC DBMS_XMLSTORAGE_MANAGE.exchangePreProc(USER, 'exchange_table');

ALTER TABLE table EXCHANGE PARTITION partition WITH TABLE exchange_table
  WITH VALIDATION UPDATE INDEXES;

EXEC DBMS_XMLSTORAGE_MANAGE.exchangePostProc(USER, 'table');
EXEC DBMS_XMLSTORAGE_MANAGE.exchangePostProc(USER, 'exchange_table');
```

Example 6-4 Exchange-Partitioning Reference-Partitioned Tables That Use XMLIndex

In this example:

- **`parent`** is the partitioned base table.
- **`child`** is a *reference*-partitioned child table with `XMLType` column `xcol`.
- **`child_xidx`** is an `XMLIndex` index with a structured component, defined on column `xcol` of table `child`. This is a local index, which is partitioned.
- **`parent_ex`** is the exchange table for base table `parent`.
- **`child_ex`** is the exchange table for child table `child`.
- **`child_xidx_ex`** is an `XMLIndex` index with a structured component, defined on column `xcol` of table `child_ex`. This is *not* a local index (unlike the case for index `child_xidx`).
- **`USER`** is the owner (database schema) of the tables.

This example performs an exchange-in operation, loading data from the exchange tables into the partitioned tables. An exchange-out operations, which loads data out of the partitioned tables into the exchange tables, would look the same, except that it would use procedure `refPartitionExchangeOut` instead. The procedure is passed the relevant tables and the necessary `ALTER TABLE ... EXCHANGE` statements.

```
EXEC DBMS_XMLSTORAGE_MANAGE.refPartitionExchangeIn(
       USER, 'parent', 'child', 'parent_ex', 'child_ex',
       'ALTER TABLE parent EXCHANGE PARTITION part_all WITH TABLE parent_ex
          INCLUDING INDEXES WITH VALIDATION UPDATE INDEXES',
       'ALTER TABLE child  EXCHANGE PARTITION part_all WITH TABLE child_ex
          INCLUDING INDEXES WITH VALIDATION UPDATE INDEXES');
```

Example 6-5 Data Used in Example of Exchange-Partitioning for Reference-Partitioned Tables

This example shows the creation operations for the tables and indexes used in Example 6-4.

```
CREATE TABLE parent (id      NUMBER PRIMARY KEY,
                     created DATE)
  PARTITION BY RANGE (created)
    (PARTITION part_2014 VALUES LESS THAN (to_date('01-jan-2015', 'dd-mon-yyyy')),
     PARTITION part_all  VALUES LESS THAN (maxvalue));


                CREATE TABLE child (parent_id NUMBER NOT NULL,
                                    xcol      XMLType,
                                    CONSTRAINT child_tab_fk FOREIGN KEY (parent_id)
                                                            REFERENCES parent (id)
                                    ENABLE VALIDATE)
                  XMLType COLUMN xcol STORE AS BINARY XML PARTITION BY REFERENCE
                (child_tab_fk);


                CREATE INDEX child_xidx ON child p (xcol) INDEXTYPE IS XDB.XMLIndex
                  PARAMETERS ('XMLTable po_index_tab ''purchaseorder''
                               COLUMNS pid NUMBER(4) PATH ''@id''') LOCAL ;


                CREATE TABLE parent_ex (id      NUMBER PRIMARY KEY,
                                        created DATE);


                CREATE TABLE child_ex (parent_id NUMBER NOT NULL,
                                       xcol      XMLType,
                                       CONSTRAINT child_tab_fk1 FOREIGN KEY (parent_id)
                                                                REFERENCES parent_ex(id)
                                      ENABLE VALIDATE)
                  XMLType COLUMN xcol STORE AS BINARY XML;


                CREATE INDEX child_ex_xidx ON child_ex p (xcol) INDEXTYPE IS XDB.XMLIndex
                  PARAMETERS ('XMLTable po_index_tab_ex ''purchaseorder''
                               COLUMNS pid NUMBER(4) PATH ''@id''');
```

Creating, Dropping, Altering, and Examining an XMLIndex Index

Basic operations on an `XMLIndex` index include creating it, dropping it, altering it, and examining it. Examples are presented.

> **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.

You create an `XMLIndex` index by declaring the index type to be `XDB.XMLIndex`, as illustrated in Example 6-6.

This creates an XMLIndex index named po_xmlindex_ix on XMLType table po_binxml. The index has only an unstructured component, no structured component.

You specify inclusion of a *structured* component in an XMLIndex index by including a *structured_clause* in the PARAMETERS clause. You specify inclusion of an *unstructured* component by including a *path_table_clause* in the PARAMETERS clause.

You can do this when you create the XMLIndex index or when you modify it. If, as in Example 6-6, you specify neither a *structured_clause* nor a *path_table_clause*, then *only* an unstructured component is included.

If an XMLIndex index has both an unstructured and a structured component, then you can drop either of these components using ALTER INDEX.

You can obtain the name of an XMLIndex index on a particular XMLType table (or column), as shown in Example 6-7. You can also select INDEX_NAME from DBA_INDEXES or ALL_INDEXES, as appropriate.

You rename or drop an XMLIndex index just as you would any other index, as illustrated in Example 6-8. This renaming changes the name of the XMLIndex index only. It does not change the name of the path table — you can rename the path table separately.

Similarly, you can change other index properties using other ALTER INDEX options, such as REBUILD. XMLIndex is no different from other index types in this respect.

The RENAME clause of an ALTER INDEX statement for XMLIndex applies only to the XMLIndex index itself. To rename the path table and secondary indexes, you must determine the names of these objects and use appropriate ALTER TABLE or ALTER INDEX statements on them directly. Similarly, to retrieve the physical properties of the secondary indexes or alter them in any other way, you must obtain their names, as in Example C-6.

See Also:

- *structured_clause ::=*
- *path_table_clause ::=*
- *drop_path_table_clause ::=*
- *alter_index_group_clause ::=*

Example 6-6 Creating an XMLIndex Index

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

Example 6-7 Obtaining the Name of an XMLIndex Index on a Particular Table

```
SELECT INDEX_NAME FROM USER_INDEXES
  WHERE TABLE_NAME = 'PO_BINXML' AND ITYP_NAME = 'XMLINDEX';

INDEX_NAME
---------------
PO_XMLINDEX_IX
```

```
1 row selected.
```

Example 6-8 Renaming and Dropping an XMLIndex Index

```
ALTER INDEX po_xmlindex_ix RENAME TO new_name_ix;

DROP INDEX new_name_ix;
```

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.

Use of XMLIndex with a Structured Component

An XMLIndex structured component indexes specific islands of structure in your XML data.

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

A *structured_clause* specifies the structured islands that you want to index. You use the keyword GROUP to specify each structured island: an island thus corresponds syntactically to a structure **group**. If you specify no group explicitly, then the predefined group DEFAULT_GROUP is used. For ALTER INDEX, you precede the GROUP keyword with the modification operation keyword: ADD_GROUP specifies a new group (island); DROP_GROUP deletes a group.

Why have multiple groups within a single index, instead of simply using multiple XMLIndex indexes? The reason is that XMLIndex is a domain index, and you can create only one domain index of a given type on a given database column.

The syntax for defining a structure group, that is, indexing a structured island, is similar to the syntax for invoking SQL/XML *function* XMLTable: you use keywords XMLTable and COLUMNS to define relational columns, and you use multilevel chaining of XMLTable to handle collections. To simplify the creation of such an index, you can use PL/SQL function DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView to provide exactly the XMLTable expression needed for creating the index.

- Using Namespaces and Storage Clauses with an XMLIndex Structured Component
  When you create an XMLIndex index that has a structured component you can specify XML namespaces and storage options to use.
- Adding a Structured Component to an XMLIndex Index
  You can use ALTER INDEX to add a structured component to an existing XMLIndex index.
- Using Non-Blocking ALTER INDEX with an XMLIndex Structured Component
  You can prevent ALTER INDEX from blocking when you add a group or column for the structured component of an XMLIndex index, so that queries that use the index do not need to wait.
- Modifying the Data Type of a Structured XMLIndex Component
  If an error is raised because some of your data does not match the data type used for the corresponding column of the structured XMLIndex component, you can in some cases simply modify the index by passing keyword MODIFY_COLUMN_TYPE to ALTER INDEX.

- Dropping an XMLIndex Structured Component
  If an `XMLIndex` index has both an unstructured and a structured component, then you can use `ALTER INDEX` to drop the structured component. You do this by dropping *all* of the structure groups that compose the structured component.
- Indexing the Relational Tables of a Structured XMLIndex Component
  Because the tables used for the structured component of an `XMLIndex` index are normal relational tables, you can index them using any standard relational indexes.

Related Topics

- Using a Registered PARAMETERS Clause for XMLIndex
  The string value used for the `PARAMETERS` clause of a `CREATE INDEX` or `ALTER INDEX` statement has a 1000-character limit. To get around this limitation, you can use PL/SQL procedures `registerParameter` and `modifyParameter` in package `DBMS_XMLINDEX`.
- Data Type Considerations for XMLIndex Structured Component
  The relational tables that are used for an `XMLIndex` structured component use SQL data types. XQuery expressions that are used in queries use XML data types (XML Schema data types and XQuery data types).

See Also:

- Indexing Binary XML Data Exposed Using a Relational View for information about using `DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView`
- Indexing XML Data for Full-Text Queries (pre-23ai)
- *structured_clause ::=*
- Usage of XMLIndex_xmltable_clause for information about an `XMLType` column in an `XMLTable` clause
- Usage of column_clause for information about keywords `COLUMNS` and `VIRTUAL`

Using Namespaces and Storage Clauses with an XMLIndex Structured Component

When you create an `XMLIndex` index that has a structured component you can specify XML namespaces and storage options to use.

Example 6-9 shows the creation of an `XMLIndex` index that has only a structured component (no path-table clause) and that uses the `XMLNAMESPACES` clause to specify namespaces. It specifies that the index data be compressed and use tablespace `USERTBS1`. The example assumes a binary XML table `po_binxml` with non XML schema-based data.

Each of the (identical) `TABLESPACE` clauses in Example 6-9 applies at the table level (tables `po_ptab` and `li_tab`).

In general you can specify storage options at both the table level and the partition level. A specification at the partition level overrides one at the table level. A `TABLESPACE` clause can also be specified at the *index* level, that is, so that it applies to all of the partitions and tables used for the index. If `TABLESPACE` is specified at more than one level, the partition level overrides the table level, which overrides the index level.

Example 6-10 specifies the same `TABLESPACE` for each of the tables used in the index. This commonality can be factored out by specifying the `TABLESPACE` at the index level, as shown in Example 6-10.

Example 6-9 XMLIndex with a Structured Component, Using Namespaces and Storage Options

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('XMLTable po_ptab
                 (TABLESPACE "USERTBS1" COMPRESS FOR OLTP)
                  XMLNAMESPACES (DEFAULT ''http://www.example.com/po''),
                 ''/purchaseOrder''
                 COLUMNS orderdate   DATE          PATH ''@orderDate'',
                         id          BINARY_DOUBLE PATH ''@id'',
                         items       XMLType       PATH ''items/item'' VIRTUAL
               XMLTable li_tab
                 (TABLESPACE "USERTBS1" COMPRESS FOR OLTP)
                  XMLNAMESPACES (DEFAULT ''http://www.example.com/po''),
                 ''/item'' PASSING items
                 COLUMNS partnum     VARCHAR2(15)  PATH ''@partNum'',
                         description CLOB          PATH ''productName'',
                         usprice     BINARY_DOUBLE PATH ''USPrice'',
                         shipdat     DATE          PATH ''shipDate''');
```

Example 6-10 XMLIndex with a Structured Component, Specifying TABLESPACE at the Index Level

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('XMLTable po_ptab,
                  XMLNAMESPACES (DEFAULT ''http://www.example.com/po''),
                 ''/purchaseOrder''
                 COLUMNS orderdate   DATE          PATH ''@orderDate'',
                         id          BINARY_DOUBLE PATH ''@id'',
                         items       XMLType       PATH ''items/item'' VIRTUAL
               XMLTable li_tab,
                  XMLNAMESPACES (DEFAULT ''http://www.example.com/po''),
                 ''/item'' PASSING items
                 COLUMNS partnum     VARCHAR2(15)  PATH ''@partNum'',
                         description CLOB          PATH ''productName'',
                         usprice     BINARY_DOUBLE PATH ''USPrice'',
                         shipdat     DATE          PATH ''shipDate''
               TABLESPACE "USERTBS1" COMPRESS FOR OLTP)');
```

Adding a Structured Component to an XMLIndex Index

You can use `ALTER INDEX` to add a structured component to an existing `XMLIndex` index.

Example 6-11 shows the creation of an `XMLIndex` index with only an unstructured component. An unstructured component is created because the `PARAMETERS` clause explicitly names the path table.

Example 6-11 then uses `ALTER INDEX` to add a structured component (group) named `po_item`. This structure group includes two relational tables, each specified with keyword `XMLTable`.

The top-level table, `po_idx_tab`, has columns `reference`, `requestor`, `username`, and `lineitem`. Column `lineitem` is of type `XMLType`. It represents a collection, so it is passed to the second `XMLTable` construct to form the second-level relational table, `po_index_lineitem`, which has columns `itemno`, `description`, `partno`, `quantity`, and `unitprice`.

The keyword VIRTUAL is *required* for an XMLType column. It specifies that the XMLType column itself is not materialized: its data is stored in the XMLIndex index only in the form of the relational columns specified by its corresponding XMLTable table.

You cannot create more than one XMLType column in a given XMLTable clause. To achieve that effect, you must instead define an additional group.

Example 6-11 also illustrates the use of a registered parameter string in the PARAMETERS clause. It uses PL/SQL procedure DBMS_XMLINDEX.registerParameter to register the parameters string named myparam. Then it uses ALTER INDEX to update the index parameters to include those in the string myparam.

Example 6-11 XMLIndex Index: Adding a Structured Component

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

BEGIN
  DBMS_XMLINDEX.registerParameter(
    'myparam',
    'ADD_GROUP GROUP po_item
       XMLTable po_idx_tab ''/PurchaseOrder''
         COLUMNS reference   VARCHAR2(30)  PATH ''Reference'',
                 requestor   VARCHAR2(30)  PATH ''Requestor'',
                 username    VARCHAR2(30)  PATH ''User'',
                 lineitem    XMLType       PATH ''LineItems/LineItem'' VIRTUAL
       XMLTable po_index_lineitem ''/LineItem'' PASSING lineitem
         COLUMNS itemno      BINARY_DOUBLE PATH ''@ItemNumber'',
                 description VARCHAR2(256) PATH ''Description'',
                 partno      VARCHAR2(14)  PATH ''Part/@Id'',
                 quantity    BINARY_DOUBLE PATH ''Part/@Quantity'',
                 unitprice   BINARY_DOUBLE PATH ''Part/@UnitPrice''');
END;
/

ALTER INDEX po_xmlindex_ix PARAMETERS('PARAM myparam');
```

Using Non-Blocking ALTER INDEX with an XMLIndex Structured Component

You can prevent ALTER INDEX from blocking when you add a group or column for the structured component of an XMLIndex index, so that queries that use the index do not need to wait.

When you use ALTER INDEX to add a group or a column for the structured component of an XMLIndex index, this index-maintenance operation obtains an exclusive DDL lock on the base table and the index.

The base table is locked to DML operations, and the index cannot be used for queries until the ALTER INDEX operation is finished. This means that during this index maintenance the index cannot be used by other sessions that query or perform DML operations on the base table. The duration of the ALTER INDEX operation and the attendant locking depends on the volume of data in the base XMLType column.

You can avoid or work around this problem as follows:

1. Use keyword **NONBLOCKING** before ADD_GROUP or ADD_COLUMN in the PARAMETERS clause of the ALTER INDEX statement that creates the structured-component group or column.

This updates the index as needed, but it does not index any base-table data. Because it does not depend on the base-table data it is quick regardless of the base-table size.

2. Invoke PL/SQL procedure `DBMS_XMLINDEX.process_pending`.

   This procedure indexes rows of the base table and populates tables of the index, just as if keyword `NONBLOCKING` were absent. However, in this case only a few rows are locked at a time while they are processed and the changes committed. Rows that have already been locked for some other purpose are skipped. This can significantly reduce lock contention and allow indexing of some rows to proceed at the same time as querying or DML on other rows.

   When procedure `process_pending` finishes it returns, as `OUT` parameters:

   - The number of rows that it could not index. This is either because they were *locked* for another purpose or because an error was raised (this number includes the number returned as the other `OUT` parameter).

     After you think those locks have been removed, invoke procedure `process_pending` again to try to process those pending rows.

   - The number of rows that it could not index because an *error* was raised. (This should be rare.)

     Check table `SYS_AIXSXI_`*`index_number`*`_ERRORTAB` for information about those errors, then take action to fix the underlying problems. *`index_number`* is the object number of the index.

3. Repeat step 2 as many times as necessary until procedure `process_pending` indicates that all rows have been successfully indexed or you encounter an insurmountable problem and decide to cancel the indexing operation altogether.

   You can cancel the indexing at any time (before step 2) by using keywords **`NONBLOCKING ABORT`** in the `PARAMETERS` clause of a separate `ALTER INDEX` statement for the same `XMLIndex` index.

4. If all rows have been successfully indexed then use keywords **`NONBLOCKING COMPLETE`** in the `PARAMETERS` clause of a separate `ALTER INDEX` statement for the same `XMLIndex` index.

Example 6-12 illustrates this.

Just as table `SYS_AIXSXI_`*`index_number`*`_ERRORTAB` reports errors, so table `SYS_AIXSXI_`*`index_number`*`_PENDINGTAB` records the current status of each base-table row: whether or not it has been indexed. A row might not yet be indexed because it is locked by for some other purpose or because trying to index it raised an error. In the latter case, consult `SYS_AIXSXI_`*`index_number`*`_ERRORTAB` for specific information about the error.

> **See Also:**
>
> *alter_index_group_clause ::=*

Example 6-12 Using DBMS_XMLINDEX.PROCESS_PENDING To Index XML Data

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('XMLTable po_idx_tab
                 ''/PurchaseOrder''
                 COLUMNS reference   VARCHAR2(30)  PATH ''Reference'',
                         requestor   VARCHAR2(30)  PATH ''Requestor'',
                         username    VARCHAR2(30)  PATH ''User'',
```

```
                  lineitem    XMLType       PATH ''LineItems/LineItem'' VIRTUAL
              XMLTable po_index_lineitem
                ''/LineItem'' PASSING lineitem
                COLUMNS itemno      BINARY_DOUBLE PATH ''@ItemNumber'',
                        description VARCHAR2(256) PATH ''Description'',
                        partno      VARCHAR2(14)  PATH ''Part/@Id'',
                        quantity    BINARY_DOUBLE PATH ''Part/@Quantity'',
                        unitprice   BINARY_DOUBLE PATH ''Part/@UnitPrice''');

ALTER INDEX po_struct
  PARAMETERS('NONBLOCKING ADD_GROUP GROUP po_action_group
              XMLTABLE po_idx_tab
                ''/PurchaseOrder''
                COLUMNS actions       XMLType       PATH  ''Actions/Action'' VIRTUAL
              XMLTABLE po_idx_action
                ''/Action'' PASSING actions
                COLUMNS actioned_by   VARCHAR2(10)  PATH  ''User'',
                        date_actioned TIMESTAMP     PATH  ''Date''');

DECLARE
  num_pending NUMBER := 0;
  num_errored NUMBER := 0;
BEGIN
 DBMS_XMLINDEX.process_pending('oe', 'po_struct', num_pending, num_errored);
 DBMS_OUTPUT.put_line('Number of rows still pending = ' || num_pending);
 DBMS_OUTPUT.put_line('Number of rows with errors   = ' || num_errored);
END;
/
Number of rows still pending = 0
Number of rows with errors   = 0

PL/SQL procedure successfully completed.

ALTER INDEX po_struct PARAMETERS('NONBLOCKING COMPLETE');
```

Modifying the Data Type of a Structured XMLIndex Component

If an error is raised because some of your data does not match the data type used for the corresponding column of the structured `XMLIndex` component, you can in some cases simply modify the index by passing keyword `MODIFY_COLUMN_TYPE` to `ALTER INDEX`.

You can, for example, expand a `VARCHAR2(30)` column to, say, `VARCHAR2(40)` if it needs to accommodate data that is up to 40 characters. This is simpler and more efficient than dropping the column and then adding a new column. The new data type must be compatible with the old one: the same restrictions apply as apply for `ALTER TABLE MODIFY COLUMN`.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about `ALTER TABLE MODIFY COLUMN`
> - *modify_column_type_clause ::==*

Dropping an XMLIndex Structured Component

If an `XMLIndex` index has both an unstructured and a structured component, then you can use `ALTER INDEX` to drop the structured component. You do this by dropping *all* of the structure groups that compose the structured component.

Example 6-13 shows how to drop the structured component that was added in Example 6-11, by dropping its only structure group, `po_item`.

Example 6-13 Dropping an XMLIndex Structured Component

```
ALTER INDEX po_xmlindex_ix PARAMETERS('DROP_GROUP GROUP po_item');
```

Indexing the Relational Tables of a Structured XMLIndex Component

Because the tables used for the structured component of an `XMLIndex` index are normal relational tables, you can index them using any standard relational indexes.

This is explained in section XMLIndex Structured Component. It is illustrated by Example 6-14, which creates a B-tree index on the `reference` column of the index content table (structured fragment) for the `XMLIndex` index of Example 6-11.

Example 6-14 Creating a B-tree Index on an XMLIndex Index Content Table

```
CREATE INDEX idx_tab_ref_ix ON po_idx_tab (reference);
```

How to Tell Whether XMLIndex is Used

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

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 example, if the path `/PurchaseOrder/LineItems//*` is included for indexing, then a query with `/PurchaseOrder/LineItems/LineItem/Description` can use the index, but a query with `//Description` cannot. The latter also matches potential `Description` elements that are not children of `/PurchaseOrder/LineItems`, and it is not possible at compile time to know if such additional `Description` elements are present in the data.

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 *structured* component of the index is used, then one or more of its index content tables is called out in the execution plan. See Example 6-15.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide*

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('GROUP po_item
```

```
                XMLTable po_idx_tab ''/PurchaseOrder''
                  COLUMNS reference   VARCHAR2(30)  PATH ''Reference'',
                          requestor   VARCHAR2(30)  PATH ''Requestor'',
                          username    VARCHAR2(30)  PATH ''User'',
                          lineitem    XMLType       PATH ''LineItems/LineItem'' VIRTUAL
                XMLTable po_index_lineitem ''/LineItem'' PASSING lineitem
                  COLUMNS itemno      BINARY_DOUBLE PATH ''@ItemNumber'',
                          description VARCHAR2(256) PATH ''Description'',
                          partno      VARCHAR2(14)  PATH ''Part/@Id'',
                          quantity    BINARY_DOUBLE PATH ''Part/@Quantity'',
                          unitprice   BINARY_DOUBLE PATH ''Part/@UnitPrice''');
```

Example 6-15 shows an execution plan that indicates that the same `XMLIndex` index is also picked up for a query that uses multilevel XMLTable chaining.

The execution plan shows direct access to the relational index content tables, `po_idx_tab` and `po_index_lineitem`.

Example 6-15 Using a Structured XMLIndex Component for a Query with Multilevel Chaining

```
EXPLAIN PLAN FOR
  SELECT po.reference, li.*
    FROM po_binxml p,
         XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
                  COLUMNS reference   VARCHAR2(30)  PATH 'Reference',
                          lineitem    XMLType       PATH 'LineItems/LineItem') po,
         XMLTable('/LineItem' PASSING po.lineitem
                  COLUMNS itemno      BINARY_DOUBLE PATH '@ItemNumber',
                          description VARCHAR2(256) PATH 'Description',
                          partno      VARCHAR2(14)  PATH 'Part/@Id',
                          quantity    BINARY_DOUBLE PATH 'Part/@Quantity',
                          unitprice   BINARY_DOUBLE PATH 'Part/@UnitPrice') li
    WHERE po.reference = 'SBELL-20021009123335280PDT';
```

```
-------------------------------------------------------------------------------
------------------------
| Id  | Operation                    | Name                  | Rows  | Bytes
| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
------------------------
|   0 | SELECT STATEMENT             |                       |    17 | 20366
|     8   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                       |       |
|           |          |
|   2 |   NESTED LOOPS               |                       |    17 | 20366
|     8   (0)| 00:00:01 |
|   3 |    NESTED LOOPS              |                       |     1 |   539
|     3   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL        | PO_IDX_TAB            |     1 |   529
|     3   (0)| 00:00:01 |
|*  5 |     INDEX UNIQUE SCAN        | SYS_C007442           |     1 |    10
```

```
|     0   (0)| 00:00:01 |
|*  6 |    INDEX RANGE SCAN          | SYS86751_86759_PKY_IDX |    17 |
|     1   (0)| 00:00:01 |
|   7 |   TABLE ACCESS BY INDEX ROWID| PO_INDEX_LINEITEM      |    17 | 11203
|     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------
------------------------

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

   4 - filter("SYS_SXI_2"."REFERENCE"='SBELL-20021009123335280PDT')
   5 - access("P"."SYS_NC_OID$"="SYS_SXI_2"."OID")
   6 - access("SYS_SXI_2"."KEY"="SYS_SXI_3"."PKEY")

Note
-----
   - dynamic sampling used for this statement

25 rows selected.
```

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.

Turning Off Use of XMLIndex

You can turn off the use of `XMLIndex` by using optimizer hint: `/*+ NO_XML_QUERY_REWRITE */` or optimizer hint `/*+ NO_XMLINDEX_REWRITE */`.

Each of these hints turns off the use of *all* `XMLIndex` indexes. In addition to turning off the use of `XMLIndex`, `NO_XML_QUERY_REWRITE` turns off all XQuery optimization (`XMLIndex` is part of XPath rewrite).

Example 6-16 shows the use of these optimizer hints.

> **Note:**
>
> The `NO_INDEX` optimizer hint does not apply to `XMLIndex`.

> **See Also:**
>
> XQuery Optional Features for information about XQuery pragmas `ora:no_xmlquery_rewrite` and `ora:xmlquery_rewrite`, which you can use for fine-grained control of XQuery optimization

Example 6-16 Turning Off XMLIndex Using Optimizer Hints

```
SELECT /*+ NO_XMLINDEX_REWRITE */
  count(*) FROM po_binxml WHERE XMLExists('$p/*' PASSING OBJECT_VALUE AS "p");

SELECT /*+ NO_XML_QUERY_REWRITE */
  count(*) FROM po_binxml WHERE XMLExists('$p/*' PASSING OBJECT_VALUE AS "p");
```

Guidelines for Using XMLIndex with a Structured Component

There are several guidelines that can help you use `XMLIndex` with a structured component.

- Use `XMLIndex` with a structured component to project and index XML data as relational columns. Do not use function-based indexes; they are deprecated for use with XML.
- Ensure data type correspondence between a query and an `XMLIndex` index that has a structured component. See Data Type Considerations for XMLIndex Structured Component.
- If you create a relational view over `XMLType` data (for example, using SQL function `XMLTable`), then consider also creating an `XMLIndex` index with a structured component that targets the same relational columns. See Relational Views over XML Data.
- Instead of using a single XQuery expression for both fragment extraction and value filtering (search), use SQL/XML function `XMLQuery` in the `SELECT` clause to extract fragments and `XMLExists` in the `WHERE` clause to filter values.

  This lets Oracle XML DB evaluate fragment extraction functionally or by using streaming evaluation. For value filtering, this lets Oracle XML DB pick up an `XMLIndex` index that has a relevant structured component.
- To order query results, use a SQL `ORDER BY` clause, together with SQL/XML function `XMLTable`. Avoid using the XQuery `order by` clause. This is particularly pertinent if you use an `XMLIndex` index with a structured component.

XMLIndex Partitioning and Parallelism

If you partition an `XMLType` table, or a table with an `XMLType` column, using range, list, or hash partitioning, you can also create an `XMLIndex` index on the table. You can optionally ensure that index creation and maintenance are carried out in parallel.

To ensure parallel index creation and maintenance, you use a `PARALLEL` clause (with optional degree) when creating or altering an `XMLIndex` index.

If you use the keyword `LOCAL` when you create the `XMLIndex` index, then the index and all of its storage tables are locally equipartitioned with respect to the base table.

If you do not use the keyword `LOCAL` then you cannot create an `XMLIndex` index on a partitioned table. Also, if you composite-partition a table, then you cannot create an `XMLIndex` index on it.

If you use a `PARALLEL` clause and the base table is partitioned or enabled for parallelism, then this can improve the performance for both DML operations (`INSERT`, `UPDATE`, `DELETE`) and index DDL operations (`CREATE`, `ALTER`, `REBUILD`).

Specifying parallelism for an index can also consume more storage, because storage parameters apply separately to each query server process. For example, an index created with an `INITIAL` value of 5M and a parallelism degree of 12 consumes at least 60M of storage during index creation.

The syntax for the parallelism clause for `CREATE INDEX` and `ALTER INDEX` is the same as for other domain indexes:

```
{ NOPARALLEL | PARALLEL [ integer ] }
```

Example 6-17 creates an `XMLIndex` index with a parallelism degree of 10. If the base table is partitioned, then this index is equipartitioned.

In Example 6-17, the path table and the secondary indexes are created with the same parallelism degree as the `XMLIndex` index itself, 10, by inheritance. You can specify different parallelism degrees for these by using separate `PARALLEL` clauses. Example 6-18 demonstrates this. Again, because of keyword `LOCAL`, if the base table is partitioned, then this index is equipartitioned.

In Example 6-18, the `XMLIndex` index itself is created serially, because of `NOPARALLEL`. The secondary index `po_pikey_ix` is also populated serially, because no parallelism is specified explicitly for it; it inherits the parallelism of the `XMLIndex` index. The path table itself is created with a parallelism degree of 10, and the secondary index value column, `po_value_ix`, is populated with a degree of 5, due to their explicit parallelism specifications.

Any parallelism you specify for an `XMLIndex` index, its path table, or its secondary indexes is exploited during subsequent DML operations and queries.

There are two places where you can specify parallelism for `XMLIndex`: within the `PARAMETERS` clause parenthetical expression and after it.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information on the `CREATE INDEX` parallel clause

Example 6-17 Creating an XMLIndex Index in Parallel

```
CREATE INDEX po_xmlindex_ix ON sale_info (sale_po_clob)
  INDEXTYPE IS XDB.XMLIndex
  LOCAL PARALLEL 10;
```

Example 6-18 Using Different PARALLEL Degrees for XMLIndex Internal Objects

```
CREATE INDEX po_xmlindex_ix ON sale_info (sale_po_clob)
  INDEXTYPE IS XDB.XMLIndex
  LOCAL NOPARALLEL PARAMETERS ('PATH TABLE po_path_table (PARALLEL 10)
                                PIKEY INDEX po_pikey_ix
                                VALUE INDEX po_value_ix (PARALLEL 5)');
```

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.
- XMLIndex Components
  XMLIndex is used to index XML data that may have some structure. It applies to XMLType data that is stored as both transportable binary XML and binary XML.

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.

> **Note:**
>
> The following applies only to procedures in package `DBMS_STATS`; it does not apply to `ANALYZE INDEX`.

For `XMLIndex`, you normally need to collect statistics on only the base table on which the `XMLIndex` index is defined (using, for example, procedure `DBMS_STATS.gather_table_stats`). This automatically collects statistics for the `XMLIndex` index itself, as well as the path table, its secondary indexes, and any structured component content tables and their secondary indexes.

If you delete statistics on the base table (using procedure `DBMS_STATS.delete_table_stats`), then statistics on the other objects are also deleted. Similarly, if you collect statistics on the `XMLIndex` index (using procedure `DBMS_STATS.gather_index_stats`), then statistics are also collected on the path table, its secondary indexes, and any structured component content tables and their secondary indexes.

Example 6-19 collects statistics on the base table `po_binxml`. Statistics are automatically collected on the `XMLIndex` index, its path table, and the secondary path-table indexes.

> **See Also:**
>
> Data Dictionary Static Public Views Related to XMLIndex for information about database views that record statistics information for an `XMLIndex` index

Example 6-19 Automatic Collection of Statistics on XMLIndex Objects

```
CALL DBMS_STATS.gather_table_stats(USER, 'PO_BINXML', ESTIMATE_PERCENT => NULL);
```

Data Dictionary Static Public Views Related to XMLIndex

Information about the standard database indexes is available in static public views `USER_INDEXES`, `ALL_INDEXES`, and `DBA_INDEXES`. Similar information about `XMLIndex` indexes is available in static public views `USER_XML_INDEXES`, `ALL_XML_INDEXES`, and `DBA_XML_INDEXES`.

Table 6-6 describes the columns in each of these views.

Table 6-6 XMLIndex Static Public Views

Column Name	Type	Description
ASYNC	VARCHAR2	Asynchronous index updating specification. See Asynchronous (Deferred) Maintenance of XMLIndex Indexes.

Table 6-6 (Cont.) XMLIndex Static Public Views

Column Name	Type	Description
EX_OR_INCLUDE	VARCHAR2	Path subsetting: • FULLY_IX – The index uses no path subsetting. • EXCLUDE – The index uses only exclusion subsetting. • INCLUDE – The index uses only inclusion subsetting.
INDEX_NAME	VARCHAR2	Name of the XMLIndex index.
INDEX_OWNER	VARCHAR2	Owner of the index. Not available for USER_XML_INDEXES.
INDEX_TYPE	VARCHAR2	The types of components the index is composed of: STRUCTURED, UNSTRUCTURED, or STRUCTURED AND UNSTRUCTURED.
PARAMETERS	XMLType	Information from the PARAMETERS clause that was used to create the index. If a structured component is present, the PARAMETERS clause includes the name of the structure group and the table definitions provided by XMLTable, including the XQuery expressions that define the columns. If an unstructured XMLIndex component is present, the PARAMETERS clause can include the set of XPath paths defining path-subsetting and the name of a scheduler job for synchronization.
PATH_TABLE_NAME	VARCHAR2	Name of the XMLIndex path table.
PEND_TABLE_NAME	VARCHAR2	Name of the table that records base-table DML operations since the last index synchronization. See Asynchronous (Deferred) Maintenance of XMLIndex Indexes.
TABLE_NAME	VARCHAR2	Name of the base table on which the index is defined.
TABLE_OWNER	VARCHAR2	Owner of the base table on which the index is defined.

These views provide information about an XMLIndex index, but there is no single static data dictionary view that provides information about the statistics gathered for an XMLIndex index. This statistics information is distributed among the following views:

- USER_INDEXES, ALL_INDEXES, DBA_INDEXES – Column LAST_ANALYZED provides the date when the XMLIndex index was last analyzed.
- USER_TAB_STATISTICS, ALL_TAB_STATISTICS, DBA_TAB_STATISTICS – Column TABLE_NAME can be used to identify the tables conforming an XMLIndex index. To retrieve statistics information on these tables, query using the name of the XMLTable table as TABLE_NAME.
- USER_IND_STATISTICS, ALL_IND_STATISTICS, DBA_IND_STATISTICS – Column INDEX_NAME provides information about each of the secondary indexes for an XMLIndex index. for information about a given secondary index, query using the name of that secondary index as INDEX_NAME.

PARAMETERS Clause for CREATE INDEX and ALTER INDEX

Creation or modification of an `XMLIndex` index often involves the use of a `PARAMETERS` clause with SQL statement `CREATE INDEX` or `ALTER INDEX`. You can use it to specify index characteristics in detail.

You can use PL/SQL procedures `registerParameter` and `modifyParameter` in package `DBMS_XMLINDEX` to bypass the 1000-character `PARAMETERS` clause limit.

- Using a Registered PARAMETERS Clause for XMLIndex
  The string value used for the `PARAMETERS` clause of a `CREATE INDEX` or `ALTER INDEX` statement has a 1000-character limit. To get around this limitation, you can use PL/SQL procedures `registerParameter` and `modifyParameter` in package `DBMS_XMLINDEX`.
- PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX
  The syntax for the `PARAMETERS` clause for `CREATE INDEX` and `ALTER INDEX` is defined.
- Usage of XMLIndex_parameters_clause
  When you create an `XMLIndex` index, if there is no *`XMLIndex_parameters_clause`*, then the new index has only an unstructured component. If there is an *`XMLIndex_parameters_clause`*, but the `PARAMETERS` argument is empty (`''`), then the result is the same: an index with only an unstructured component.
- Usage of XMLIndex_parameters
  Certain considerations apply to using *`XMLIndex_parameters`*.
- Usage of groups_clause and alter_index_group_clause
  Clause *`groups_clause`* is used only with `CREATE INDEX` (or following `ADD GROUP` in clause *`alter_index_group_clause`*). Clause *`alter_index_group_clause`* is used only with `ALTER INDEX`.
- Usage of XMLIndex_xmltable_clause
  Certain considerations apply to using *`XMLIndex_xmltable_clause`*.
- Usage of column_clause
  Certain considerations apply to using *`column_clause`*.

See Also:

- PARAMETERS Clause for CREATE INDEX and ALTER INDEX in Unstructured Index
- *Oracle Database SQL Language Reference* for the syntax of *index_attributes*
- *Oracle Database SQL Language Reference* for the syntax of *segment_attributes_clause*
- *Oracle Database SQL Language Reference* for the syntax of *table_properties*
- *Oracle Database SQL Language Reference* for the syntax of *parallel_clause*
- *Oracle Database SQL Language Reference* for additional information about the syntax and semantics of `CREATE INDEX`
- *Oracle Database SQL Language Reference* for additional information about the syntax and semantics of `ALTER INDEX`
- *Oracle Database PL/SQL Packages and Types Reference*, section "Calendaring Syntax", for the syntax of *repeat_interval*

Using a Registered PARAMETERS Clause for XMLIndex

The string value used for the `PARAMETERS` clause of a `CREATE INDEX` or `ALTER INDEX` statement has a 1000-character limit. To get around this limitation, you can use PL/SQL procedures `registerParameter` and `modifyParameter` in package `DBMS_XMLINDEX`.

For each of these procedures, you provide a string of parameters (unlimited in length) and an identifier under which the string is registered. Then, in the index `PARAMETERS` clause, you provide the identifier preceded by the keyword `PARAM`, instead of a literal string.

The identifier must already have been registered before you can use it in a `CREATE INDEX` or `ALTER INDEX` statement.

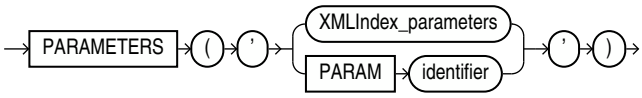
See Also:

Example 6-11

PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX

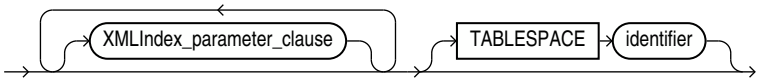
The syntax for the `PARAMETERS` clause for `CREATE INDEX` and `ALTER INDEX` is defined.

XMLIndex_parameters_clause ::=



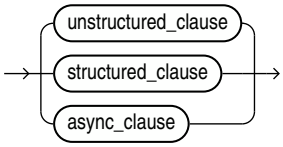
> **See Also:**
>
> Usage of XMLIndex_parameters_clause

XMLIndex_parameters ::=

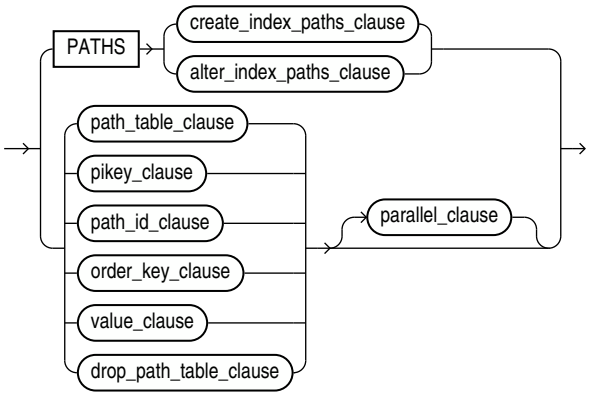


> **See Also:**
>
> Usage of XMLIndex_parameters

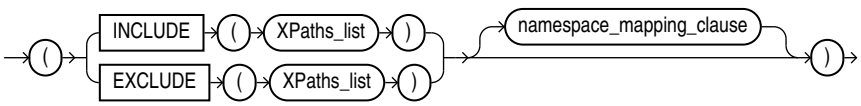
XMLIndex_parameter_clause ::=



unstructured_clause ::=

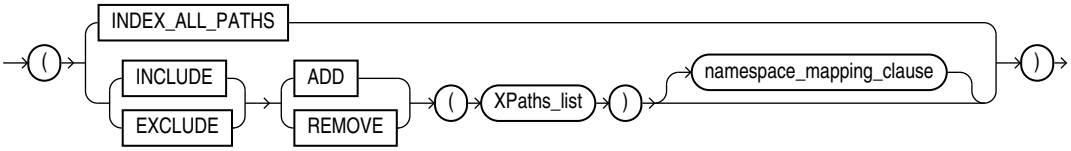


create_index_paths_clause ::=



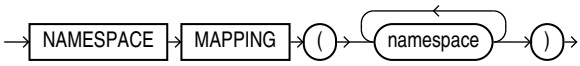
> **See Also:**
>
> - Usage of PATHS Clause
> - Usage of create_index_paths_clause and alter_index_paths_clause

alter_index_paths_clause ::=

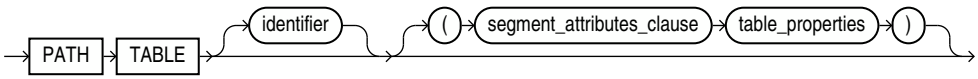


> **See Also:**
>
> - Usage of PATHS Clause
> - Usage of create_index_paths_clause and alter_index_paths_clause

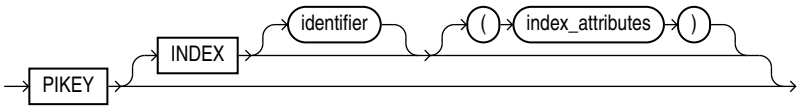
namespace_mapping_clause ::=



path_table_clause ::=

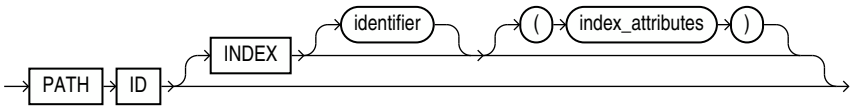


pikey_clause ::=



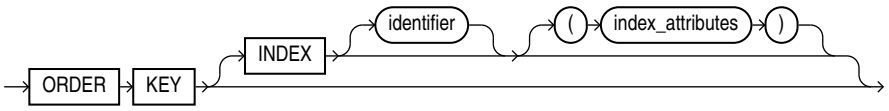
> **See Also:**
>
> Usage of pikey_clause, path_id_clause, and order_key_clause

path_id_clause ::=



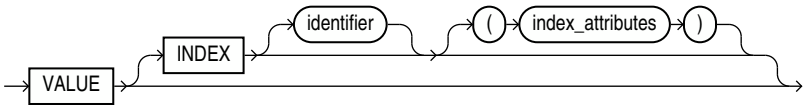
> **See Also:**
>
> Usage of pikey_clause, path_id_clause, and order_key_clause

order_key_clause ::=



> **See Also:**
>
> Usage of pikey_clause, path_id_clause, and order_key_clause

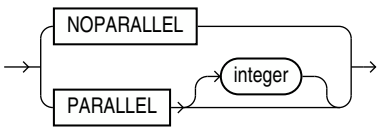
value_clause ::=



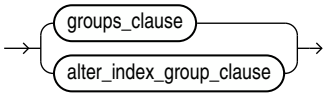
> **See Also:**
>
> Usage of value_clause

drop_path_table_clause ::=

parallel_clause ::=

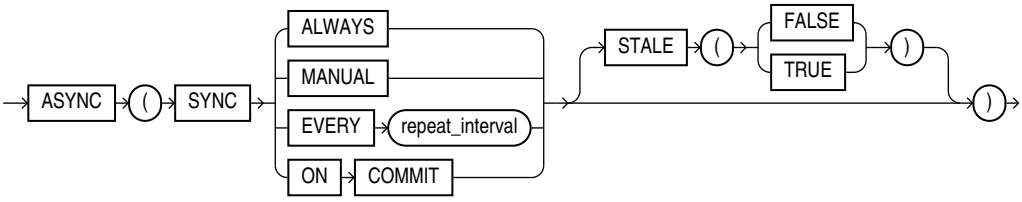


structured_clause ::=



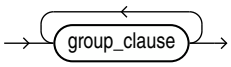
> **See Also:**
>
> Usage of groups_clause and alter_index_group_clause

async_clause ::=



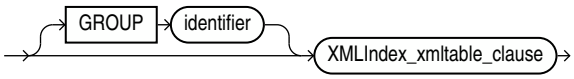
> **See Also:**
>
> Usage of async_clause

groups_clause ::=



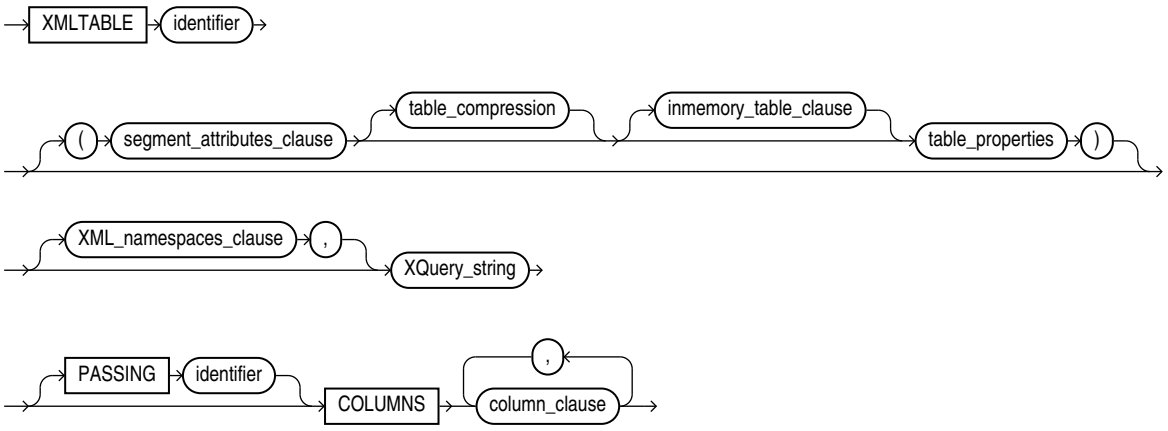
> **See Also:**
>
> Usage of groups_clause and alter_index_group_clause

group_clause ::=



> **See Also:**
>
> Usage of groups_clause and alter_index_group_clause

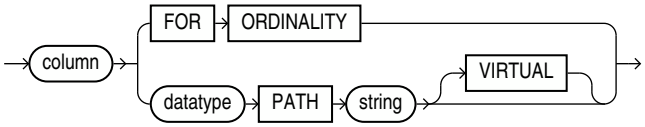
XMLIndex_xmltable_clause ::=



Syntax elements `XML_namespaces_clause` and `XQuery_string` are the same as for SQL/XML function `XMLTable`.

> **See Also:**
>
> - Usage of XMLIndex_xmltable_clause
> - XMLTABLE SQL/XML Function in Oracle XML DB

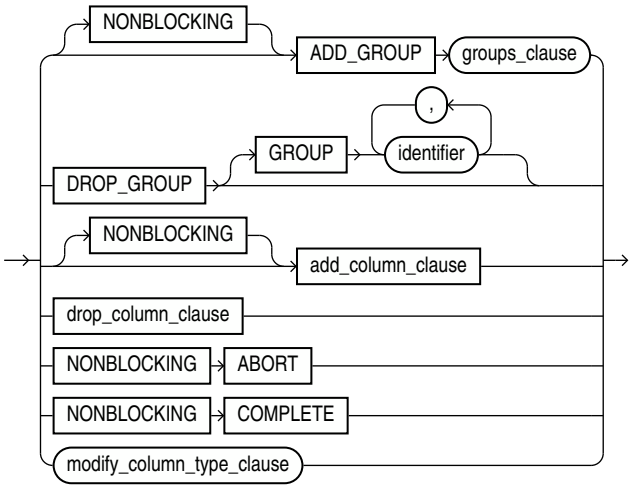
column_clause ::=



Syntax element `column_clause` is similar, but not identical, to `XML_table_column` in SQL/XML function `XMLTable`.

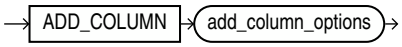
> **See Also:**
>
> - Usage of column_clause
> - XMLTABLE SQL/XML Function in Oracle XML DB

alter_index_group_clause ::=

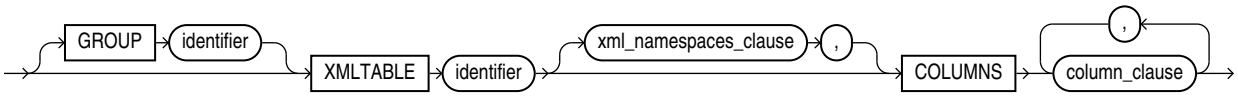


> **See Also:**
>
> Usage of groups_clause and alter_index_group_clause

add_column_clause :==

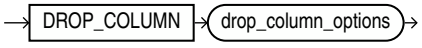


add_column_options :==

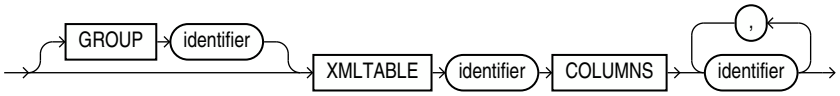


Syntax element `XML_namespaces_clause` is the same as for SQL/XML function `XMLTable`. See XMLTABLE SQL/XML Function in Oracle XML DB.

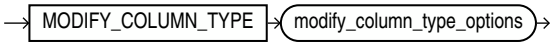
drop_column_clause :==



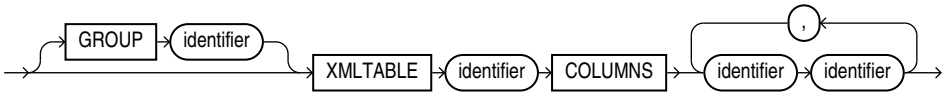
drop_column_options :==



modify_column_type_clause :==



modify_column_type_options :==



Usage of XMLIndex_parameters_clause

When you create an `XMLIndex` index, if there is no *`XMLIndex_parameters_clause`*, then the new index has only an unstructured component. If there is an *`XMLIndex_parameters_clause`*, but the `PARAMETERS` argument is empty (`''`), then the result is the same: an index with only an unstructured component.

> **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.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the use context for *`XMLIndex_parameters_clause`* in `CREATE INDEX`
> - *Oracle Database SQL Language Reference* for information about the use context for *`XMLIndex_parameters_clause`* in `ALTER INDEX`

Usage of XMLIndex_parameters

Certain considerations apply to using *`XMLIndex_parameters`*.

> **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.

- There can be at most one *XMLIndex_parameter_clause* of each type in *XMLIndex_parameters*. For example, there can be at most one PATHS clause, at most one *path_table_clause*, and so on.
- If there is no *structured_clause* when you create an XMLIndex index, then the new index has only an unstructured component. If there is only a *structured_clause*, then the new index has only a structured component.

Usage of groups_clause and alter_index_group_clause

Clause *groups_clause* is used only with CREATE INDEX (or following ADD GROUP in clause *alter_index_group_clause*). Clause *alter_index_group_clause* is used only with ALTER INDEX.

Usage of XMLIndex_xmltable_clause

Certain considerations apply to using *XMLIndex_xmltable_clause*.

- The *XQuery_string* expression in *XMLIndex_xmltable_clause* must not use the XQuery functions ora:view (*desupported*), fn:doc, or fn:collection.
- Oracle XML DB raises an error if a given *XMLIndex_xmltable_clause* contains more than one *column_clause* of data type XMLType. To achieve the effect of defining two such virtual columns, you must instead add a separate *group_clause*.
- The PASSING clause in *XMLIndex_xmltable_clause* is optional. If not present, then an XMLType column is passed implicitly, as follows:
  - For the first *XMLIndex_xmltable_clause* in a parameters clause, the XMLType column being indexed is passed implicitly. (When indexing an XMLType table, pseudocolumn OBJECT_VALUE is passed.)
  - For each subsequent *XMLIndex_xmltable_clause*, the VIRTUAL XMLType column of the preceding *XMLIndex_xmltable_clause* is passed implicitly.

Usage of column_clause

Certain considerations apply to using *column_clause*.

When you use multilevel chaining of XMLTable in an XMLIndex index, the XMLTable table at one level corresponds to an XMLType column at the previous level. The syntax description shows keyword VIRTUAL as optional. In fact, it is used only for such an XMLType column, in which case it is *required*. It is an error to use it for a non-XMLType column. VIRTUAL specifies that the XMLType column itself is not materialized, meaning that its data is stored in the index only in the form of the relational columns specified by its corresponding XMLTable table.

XML Search Index: Indexing for Full Text Search and Ad-hoc Queries

When you need full-text search, or range-search capabilities over large, unstructured documents stored as Transportable Binary XML, Oracle recommends that you use XML Search Index to index the XML data.

The XMLExists expression supports querying XMLType data that is stored as Transportable Binary XML with XQuery Full Text (XQFT) predicates. If you use an XQFT full-text predicate in

an XMLExists expression within a SQL WHERE clause, then you must create an XML Search Index. This section describes the creation and use of such an index.

Oracle recommends that you store your data in the Transportable Binary XML format. When your data is not using TBX storage, you can still index XML data by creating an XQuery Full Text CONTEXT index or an Oracle Text Index. For more information on how to create an XQuery Full Text CONTEXT index, see Indexing XML Data for Full-Text Queries (pre-23ai).

- Creating and Using an XML Search Index
  To create an XML search index, specify the FOR XML clause in the `CREATE SEARCH INDEX` statement. You can create an XML search index only on a column with the `SYS.XMLType` data type that stores documents using the TRANSPORTABLE BINARY XML (TBX) storage option, which is the 23ai default.
- Maintenance of XML Search Indexes
  XML Search Indexes are asynchronous indexes. New documents are not immediately reflected by the index until the new documents are synchronized.
- Preference Defaults and Recommendations
- Queries using an XML Search Index
- Migrating to Use XML Search Index
  XML Search Indexes can only be used when the XMLType is stored as Transportable Binary XML.

> **See Also:**
>
> - Oracle Text Reference for more information about CONTEXT indexes and Oracle Text Search indexes.
> - Indexing XML Data for Full-Text Queries (pre-23ai) for information on how to index XML Data using XQuery Full Text Indexes.

Creating and Using an XML Search Index

To create an XML search index, specify the FOR XML clause in the `CREATE SEARCH INDEX` statement. You can create an XML search index only on a column with the `SYS.XMLType` data type that stores documents using the TRANSPORTABLE BINARY XML (TBX) storage option, which is the 23ai default.

An XQuery Full Text query can use an XML search index to improve performance. To create an XML Search Index you must be granted database role `CTXAPP`. More generally, this role is needed to create and maintain Oracle Text indexes, to set Oracle Text index preferences, or to use Oracle Text PL/SQL packages.

To create an XML Search Index, specify the FOR XML clause in the `CREATE SEARCH INDEX` statement. You can create an XML search index only on a column with the `SYS.XMLType` data type that stores documents using the `TRANSPORTABLE BINARY XML` (TBX) storage option.

If you run the `CREATE SEARCH INDEX` command without the FOR XML clause on a column that is of type `SYS.XMLTYPE` the type of the created index will depend on the storage option of the column. When the storage option of the column is TBX, and XML Search Index is created. Otherwise, an Oracle Text index is created.

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (document);
```

You may use the FOR XML clause explicitly if you have ensured that the storage option of the column is TBX.

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML;
```

When creating an XML Search Index, you must decide the types of queries that you expect to take advantage of the index.

Type of Query	SEARCH_ON argument
XQuery Full Text	`TEXT`
Range-search on Numbers	`VALUE(BINARY_FLOAT)` or `VALUE(NUMBER)`
Range-search on Timestamp	`VALUE(TIMESTAMP)`
Range-search on Keywords	`VALUE(VARCHAR2)`

If you will only use XQuery Full Text predicates in an `XMLExists` operator, then specify `SEARCH_ON TEXT` on the index's parameter string. This is the default if no `SEARCH_ON` clause is specified:

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document)
FOR XML PARAMETERS ('SEARCH_ON TEXT');
```

Multiple `SEARCH_ON` clauses can be specified as part of the `PARAMETERS` clause:

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document)
FOR XML PARAMETERS ('SEARCH_ON VALUE(BINARY_DOUBLE) SEARCH_ON VALUE(VARCHAR2)');
```

Equivalently, it can be combined into a single clause as below:

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document)
FOR XML PARAMETERS ('SEARCH_ON VALUE(BINARY_DOUBLE, VARCHAR2)');
```

> **Note:**
>
> You cannot specify the same data type multiple times.

If you plan on enabling the index for both range-search and full-text search queries, then specify `SEARCH_ON TEXT_VALUE`.

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document)
FOR XML PARAMETERS ('SEARCH_ON TEXT_VALUE(BINARY_DOUBLE, VARCHAR2, TIMESTAMP)');
```

You can index XML data that is not stored using the TBX option by creating an XQuery Full Text `CONTEXT` index.

Syntax for XML Search Index:

```
CREATE SEARCH INDEX [schema.]index ON [schema.]table(xml_column)
FOR XML
[LOCAL]
PARAMETERS(
  [SEARCH_ON (TEXT | TEXT_VALUE(data_types) | VALUE(data_types))]
  [STORAGE storage_pref]
  [MEMORY memsize]
  [SYNC (MANUAL | EVERY "interval-string" | ON COMMIT)]
```

```
  [MAINTENANCE AUTO | MAINTENANCE MANUAL]
  [OPTIMIZE (MANUAL | EVERY "interval-string" | AUTO_DAILY)]
)
[PARALLEL N]
[UNUSABLE];
```

where:

- `[schema.]index` specifies the name of the XML search index to create.
- `[schema.]table(xml_column)` specifies the names of table and column to index. `xml_column` is the name of the column on which the index is created.

> **See Also:**
>
> *Oracle Text Reference* for more information about these parameters.

Maintenance of XML Search Indexes

XML Search Indexes are asynchronous indexes. New documents are not immediately reflected by the index until the new documents are synchronized.

When maintaining an XML Search Index, the index will synchronize unindexed documents in the background in a system-controlled manner by default. The index synchronization can also be triggered on demand by invoking procedure `CTX_DDL.SYNC_INDEX`.

Index synchronization can be tuned to happen on transaction commit or periodically after at least an interval of time has passed. To change the method of index synchronization, specify the `SYNC` option to:

- `SYNC (ON COMMIT)` – The index will be synchronized during transaction commit.
- `SYNC (EVERY "interval information")` – The index will be synchronized periodically in the background by verifying if there are documents pending indexing.

To create an XML Search Index that synchronizes during commit time:

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document)
FOR XML PARAMETERS ('SYNC (ON COMMIT) SEARCH_ON TEXT_VALUE(BINARY_DOUBLE, VARCHAR2)');
```

To create an XML Search Index that synchronizes every two hours:

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document)
FOR XML PARAMETERS ('SYNC (EVERY "FREQ=HOURLY;INTERVAL=2") SEARCH_ON
TEXT_VALUE(BINARY_DOUBLE, VARCHAR2)');
```

> **See Also:**
>
> Oracle Text Reference for details on index synchronization and maintenance

Preference Defaults and Recommendations

When you create an XML Search Index with no Oracle Text preferences, the following preferences are enabled by default:

Preference	Attribute	Value
N/A	MAINTENANCE	AUTO
BASIC_STORAGE	STAGE_ITAB	TRUE
BASIC_STORAGE	STAGE_ITAB_MAX_ROWS	10000
BASIC_STORAGE	STAGE_ITAB_AUTO_OPT	TRUE
BASIC_STORAGE	XML_SAVE_COPY	TRUE

You can specify your own Text index preferences when creating an XML Search Index. Oracle recommends that the following preferences are used:

- D_TABLE_CLAUSE – Specify SECUREFILE storage for column DOC of index data table $D, which contains information about the structure of your XML documents. Specify caching and medium compression.
- I_TABLE_CLAUSE – Specify SECUREFILE storage for column TOKEN_INFO of index data table $I, which contains information about full-text tokens and their occurrences in the indexed documents. Specify caching (but not compression).
- STAGE_ITAB - Specify TRUE to enable the staging table feature. The staging table decreases the amount of fragmentation in the index after maintenance operations by merging the contents of the staging table with the index only after a specific row threshold is reached. Data in the staging table is still part of the index.
- STAGE_ITAB_MAX_ROWS - Specify a value of 10000 or more. This value should be tuned according to how often the index will be maintained due to DML. Larger values will merge the contents of the staging table less often while keeping the staging table fragmented.
- STAGE_ITAB_AUTO_OPT - Specify TRUE to allow the merging of the staging table to happen in the background.
- STAGE_ITAB_PARALLEL - Specify a parallel degree when merging the staging table. By default, the merge operation will use parallel degree 4.

You can use the interfaces defined in the CTX_DDL package and the PARAMETERS clause to specify non-default values for these options:

```
BEGIN
  CTX_DDL.create_preference('xml_sto', 'BASIC_STORAGE');
  CTX_DDL.set_attribute('xml_sto',
                        'D_TABLE_CLAUSE',
                        'TABLESPACE my_ts
                         LOB(DOC) STORE AS SECUREFILE
                         (TABLESPACE my_ts COMPRESS MEDIUM CACHE)');
  CTX_DDL.set_attribute('xml_sto',
                        'I_TABLE_CLAUSE',
                        'TABLESPACE my_ts
                         LOB(TOKEN_INFO) STORE AS SECUREFILE
                         (TABLESPACE my_ts NOCOMPRESS CACHE)');
  CTX_DDL.set_attribute('xml_sto', 'STAGE_ITAB', 'TRUE');
  CTX_DDL.set_attribute('xml_sto', 'STAGE_ITAB_MAX_ROWS', '10000');
  CTX_DDL.set_attribute('xml_sto', 'STAGE_ITAB_AUTO_OPT', 'TRUE');
  CTX_DDL.set_attribute('xml_sto', 'STAGE_ITAB_PARALLEL', '4');

END;
```

```
/

CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETERS ('STORAGE
xml_sto SEARCH_ON TEXT_VALUE(NUMBER, VARCHAR2)');
```

Note:

- Oracle Text Reference for information about procedure `CTX_DDL.set_sec_grp_attr`
- Oracle Text Reference for information about procedure `CTX_DDL.create_preference`
- Oracle Text Reference for information about procedure `CTX_DDL.set_attribute`
- Oracle Text Reference for information about preference `BASIC_STORAGE`, `D_TABLE_CLAUSE`, and `I_TABLE_CLAUSE`

Queries using an XML Search Index

An XML Search Index is considered for usage when there are XMLExists predicates in your query with the following operators:

- XQuery Full Text Predicate Operators
  - `CONTAINS TEXT`
  - `CONTAINS TEXT … FTAND`
  - `CONTAINS TEXT … FTOR`
  - `CONTAINS TEXT … FTAND FTNOT`
  - `CONTAINS TEXT … WINDOW n WORDS`
- Range Search Predicates over scalar values of numeric, datatime, or string data type using relational operators.

Note:

Which predicates inside an `XMLExists` operator pick up the index are determined by the index's setting for `SEARCH_ON`. If you use XQuery Full Text Predicate Operators for an index that is not Full-Text enabled, then compile time error ORA-18177 will be raised.

The following restrictions need to be considered when writing queries to pick up an XML Search Index:

- The expression that computes the XML nodes for the search context must be an XPath expression whose steps are only along forward and descendent axes.
- You can pass only one XMLType instance as a SQL expression in the PASSING clause of SQL/XML function XMLExists, and each of the other, non-XMLType SQL expressions in

that clause must be either a compile-time constant of a SQL built-in data type or a bind variable that is bound to an instance of such a data type.

Example 6-20 XQuery Full Text Query

```
CREATE TABLE PO_BINXML(doc xmltype) xmltype doc STORE AS transportable binary xml;

CREATE SEARCH INDEX PO_CTX_IDX ON
      PO_BINXML(doc) FOR XML PARAMETERS ('search_on text sync (on
    commit)');

SELECT XMLQuery('for $i in /PurchaseOrder/LineItems/LineItem/Description
                   where $i[. contains text "Big" ftand "Street"]
               return <Title>{$i}</Title>'
               PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM po_binxml
  WHERE XMLExists('/PurchaseOrder/LineItems/LineItem/Description
                  [. contains text "Big" ftand "Street"]'
                 PASSING OBJECT_VALUE);
```

Example 6-21 Execution Plan for XQuery Full Text Query

```
-------------------------------------------------------------------------------------------
-
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time
|
-------------------------------------------------------------------------------------------
-
|   0 | SELECT STATEMENT             |            |     1 |  2014 |     4   (0)| 00:00:01
|
|   1 |  TABLE ACCESS BY INDEX ROWID| PO_BINXML  |     1 |  2014 |     4   (0)| 00:00:01
|
|*  2 |   DOMAIN INDEX               | PO_CTX_IDX |       |       |     4   (0)| 00:00:01
|
-------------------------------------------------------------------------------------------
-

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

   2 - access("CTXSYS"."CONTAINS"(SYS_MAKEXML(0,"XMLDATA"),'<query><textquery
             grammar="CONTEXT" lang="english"> ( ( {Big} ) and ( {Street} ) )  INPATH
             (/PurchaseOrder/LineItems/LineItem/Description)</textquery></query>')>0)
```

Here are some more full text query examples:

Retrieving records where description contains text "Julius"

```
select id FROM po_binxml
where xmlexists('/PurchaseOrder/LineItems/LineItem/Description[. contains text
"Julius"]' PASSING doc);
```

Retrieving records where address contains text "Oracle" or "Austin"

```
select id FROM po_binxml where
xmlexists('/PurchaseOrder/ShippingInstructions/address[. contains text "Oracle" ftor
"Austin"]' PASSING doc);
```

In addition to the restrictions for any query that will use an XML Search Index, when using range-search predicates in your query, the following restrictions are applied:

- The comparison value must be a compile time constant or a bind variable that was bound prior to executing the cursor.
- The data type of the compile time constant or bind variable must be enabled in the index.
- The relational predicate must not be `!=`.
- The predicate must not be negated by using the `NOT()`.

To ensure that the correct data type is picked up for the index, data type constructors like `xs:double`, `xs:decimal`, or `xs:dateTime` can be used to allow the correct data type to be inferred.

Example 6-22 A Range Search Query in an XMLExists predicate answered by an XML Search Index

```
CREATE SEARCH INDEX XSIDX ON xtab(doc)
FOR xml parameters ('search_on value(number, varchar2, timestamp) sync (on commit)');

SELECT XMLQuery('for $i in /PurchaseOrder
                 return <Details>{$i/User}{$i/Reference}</Details>'
               PASSING doc RETURNING CONTENT)
  FROM xtab
  WHERE XMLExists('/PurchaseOrder/LineItems/LineItem/Part[UnitPrice < 15.0]'
                  PASSING doc);
```

This query returns User and Reference of Purchase orders that contain at least one item that is cheaper than $15.00.

Example 6-23 Execution Plan for a Range Search Query in an XMLExists predicate answered by an XML Search Index

```
---------------------------------------------
| Id  | Operation                    | Name  |
---------------------------------------------
|   0 | SELECT STATEMENT             |       |
|   1 |  SORT AGGREGATE              |       |
|*  2 |   FILTER                     |       |
|   3 |    FAST DUAL                 |       |
|   4 |  NESTED LOOPS                |       |
|   5 |   TABLE ACCESS BY INDEX ROWID| XTAB  |
|*  6 |    DOMAIN INDEX              | XSIDX |
|   7 |   XPATH EVALUATION           |       |
---------------------------------------------

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

   2 - filter(:B1 IS NOT NULL)
   6 - access("CTXSYS"."CONTAINS"(SYS_MAKEXML(131072,"SYS_NC00002$"),'<query><
              textquery grammar="CONTEXT"
lang="english">sdata(FNUM_284549FF0C91A6F6A8AABBEE
              2B5798E5_UnitPrice  < 15.0 )</textquery></query>')>0)
```

Retrieving orders after Feb 5

```
SELECT id from xtab where
      xmlexists('/PurchaseOrder[@orderDate > xs:dateTime("2024-02-05")]' PASSING
    doc);
```

Retrieving orders with Description > "Light"

```
SELECT id from xtab where
      xmlexists('/PurchaseOrder/LineItems/LineItem[Description > "Lava"]' PASSING
    doc);
```

Which index is used on a query?

When the only index on an XMLType column is an XML Search Index, queries will attempt to use the XML Search Index by default.

However, when there is an Structured XMLIndex on the same column as the XML Search Index, usage of the Structured XMLIndex will take priority over the XML Search index.

XQuery Full Text queries will continue to pick up the XML Search Index, meanwhile, range-search queries will use the Structured XMLIndex instead.

This behavior can be overridden by using the `ora:use_xmltext_idx` pragma to force the XML Search Index to be picked up over the Structured XML Index.

```
SELECT XMLQuery('for $i in /PurchaseOrder
                 return <Details>{$i/User}{$i/Reference}</Details>'
               PASSING doc RETURNING CONTENT)
  FROM xtab
  WHERE XMLExists('(#ora:use_xmltext_idx #)){/PurchaseOrder/LineItems/LineItem/
Part[UnitPrice < 15.0]}'
                  PASSING doc);
```

Furthermore, the `NO_INDEX` hint can be used to avoid considering the XML Search Index at all.

Migrating to Use XML Search Index

XML Search Indexes can only be used when the XMLType is stored as Transportable Binary XML.

If you have legacy queries for XMLType data stored as Non Transportable Binary XML that use SQL function `CONTAINS` and an Oracle Text index that is not XML-enabled the data can be migrated to use Transportable Binary XML Storage and use an XML Search Index.

Furthermore, if you have legacy queries for XMLType data stored as Non Transportable Binary XML alongside an XQuery Full Text Context index, you can also migrate this data to use Transportable Binary XML and use XML Search Index.

Oracle recommends that you use XML Search Index for indexing of unstructured sections of XML documents. XML Search Indexes allow for much greater flexibility in data partitioning. CONTEXT indexes and XQuery Full Text indexes can only be locally partitioned over range-partitioned tables.

When migrating from a CONTEXT index to an XML Search Index, the usage of the `CONTAINS` function should be replaced with `XMLEXISTS` predicates.

When migrating from an XQFT index to an XML Search Index, the `ora:use_xmltext_idx` pragma used within some of the XMLExists predicates is only required to force the XML Search Index to be picked up when there is an Structured XML Index on the same column.

Migrating from using Oracle Text Index to XML Search Index

The table below provides examples of typical queries using Oracle Text specific operators and their replacement when using XML Search Index:

Original Example	Replacement Example
`CONTAINS(t.x, 'HASPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description}' PASSING t.x AS "d")`
`CONTAINS(t.x, 'Big INPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description [. contains text "Big"]' PASSING t.x AS "d")`
`CONTAINS(t.x, '(Big) AND (Street) INPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description [. contains text "Big" ftand "Street"]' PASSING t.x AS "d")`
`CONTAINS(t.x, '(Big) OR (Street) INPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description [. contains text "Big" ftor "Street"]' PASSING t.x AS "d")`
`CONTAINS(t.x, '({Big}) NOT ({Street}) INPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description [. contains text "Big" ftand ftnot "Street"]' PASSING t.x AS "d")`
`CONTAINS(t.x, '({Street}) MNOT ({Big Street}) INPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description [. contains text "Street" not in "Big Street"]' PASSING t.x AS "d")`
`CONTAINS(t.x, '(NEAR (({Big}, {Street}), 3) INPATH (/P/LIs/LI/Description)') > 0`	`XMLExists('$d/P/LIs/LI/Description [. contains text "Big" ftand "Street" window 3 words]' PASSING t.x AS "d")`
(Not applicable – Oracle Text queries are not XML namespace aware.)	`XMLExists('declare namespace ipo="http://www.example.com/IPO"; /ipo:P/ipo:LIs/ipo:LI/ipo:Description [. contains text "Big"]' PASSING t.x AS "d")`

The path test can contain a predicate expression, which is the same for both the original query (with HASPATH) and its replacement. For example:

```
/PurchaseOrder/LineItems/LineItem/Part[@Id < "31415927"]
```

Migrating from using XQuery Full Text Index to XML Search Index

The main purpose of an XQuery Full Text Index is to efficiently search data in XML documents using XQuery Full Text predicates.

Creating an XQuery Full Text Index requires then the creation of a Section Group and Storage preferences to configure the underlying CONTEXT index.

The equivalent index creation statement when using XML Search Index is:

```
CREATE SEARCH INDEX xml_ft_idx ON xmldoctab (tbx_document) FOR XML PARAMETESRS
('SEARCH_ON TEXT');
```

Indexing XMLType Data Stored Object-Relationally

You can effectively index `XMLType` data that is stored object-relationally by creating B-tree indexes on the underlying database columns that correspond to XML nodes.

If the data to be indexed is a *singleton*, that is, if it can occur only once in any XML instance document, then you can use a *shortcut* of ostensibly creating a function-based index, where the expression defining the index is a functional application, with an XPath-expression argument that targets the singleton data. A shortcut is defined for `XMLCast` applied to `XMLQuery`, and another shortcut is defined for (deprecated) Oracle SQL function `extractValue`.

In many cases, Oracle XML DB then automatically creates appropriate indexes on the underlying object-relational tables or columns; it does *not* create a function-based index on the targeted `XMLType` data as the `CREATE INDEX` statement would suggest.

In the case of the `extractValue` shortcut, the index created is a B-tree index. In the case of `XMLCast` applied to `XMLQuery`, the index created is a function-based index on the scalar value resulting from the functional expression.

If the data to be indexed is a *collection*, then you cannot use such a shortcut; you must create the B-tree indexes manually.

- Indexing Non-Repeating Text Nodes or Attribute Values
  Table `purchaseorder` in sample database schema `OE` is stored object-relationally. Each purchase-order document has a single `Reference` element; this element is a singleton. You can thus use a shortcut to create an index on the underlying object-relational data.
- Indexing Repeating (Collection) Elements
  In `XMLType` data stored object-relationally, a collection is stored as an ordered collection table (OCT) of an `XMLType` instance, which means that you can directly access its members. Because object-relational storage directly reflects the fine-grained structure of the XML data, you can create indexes that target individual collection members.

Indexing Non-Repeating Text Nodes or Attribute Values

Table `purchaseorder` in sample database schema `OE` is stored object-relationally. Each purchase-order document has a single `Reference` element; this element is a singleton. You can thus use a shortcut to create an index on the underlying object-relational data.

Example 6-24 shows a `CREATE INDEX` statement that ostensibly tries to create a function-based index using `XMLCast` applied to `XMLQuery`, targeting the text content of element `Reference`. (The content of this element is only text, so targeting the element is the same as targeting its text node using XPath node test `text()`.)

Example 6-25 ostensibly tries to create a function-based index using (deprecated) Oracle SQL function `extractValue`, targeting the same data.

In reality, in both Example 6-24 and Example 6-25 no function-based index is created on the targeted `XMLType` data. Instead, Oracle XML DB rewrites the `CREATE INDEX` statements to create indexes on the underlying scalar data.

> **See Also:**
>
> Example 19-7 and Example 19-8 for information about XPath rewrite as it applies to such `CREATE INDEX` statements

In some cases when you use either of these shortcuts, the `CREATE INDEX` statement is not able to create an index on the underlying scalar data as described, and it instead actually does create a function-based index on the referenced `XMLType` data. (This is so, even if the *value* of the index might be a scalar.)

If this happens, drop the index, and create instead an `XMLIndex` index with a structured component that targets the same XPath. As a general rule, Oracle recommends against using a function-based index on `XMLType` data.

This is an instance of a general rule for `XMLType` data, regardless of the storage method used: Use an `XMLIndex` with a structured component instead of a function-based index. This rule applies starting with Oracle Database 11g Release 2 (11.2). Respecting this rule obviates the overhead associated with maintenance operations on function-based indexes, and it can increase the number of situations in which the optimizer can correctly select the index.

Example 6-24 CREATE INDEX Using XMLCAST and XMLQUERY on a Singleton Element

```
CREATE INDEX po_reference_ix ON purchaseorder
  (XMLCast(XMLQuery ('$p/PurchaseOrder/Reference' PASSING po.OBJECT_VALUE AS "p"
                                                  RETURNING CONTENT)
              AS VARCHAR2(128)));
```

Example 6-25 CREATE INDEX Using EXTRACTVALUE on a Singleton Element

```
CREATE INDEX po_reference_ix ON purchaseorder
  (extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference'));
```

Indexing Repeating (Collection) Elements

In `XMLType` data stored object-relationally, a collection is stored as an ordered collection table (OCT) of an `XMLType` instance, which means that you can directly access its members. Because object-relational storage directly reflects the fine-grained structure of the XML data, you can create indexes that target individual collection members.

You must create such indexes manually. The special feature of automatically creating B-tree indexes when you ostensibly create a function-based index for (deprecated) Oracle SQL function `extractValue` does *not* apply to collections (the XPath expression passed to `extractValue` must target a singleton).

To create B-tree indexes for a collection, you must understand the structure of the SQL object that is used to manage the collection. Given this information, you can use conventional object-relational SQL code to created the indexes directly on the appropriate SQL-object attributes. Refer to Guideline: Create indexes on ordered collection tables for an example of how to do this.