

# Mastering XML DB Storage in Oracle Database 10g Release 2

*An Oracle White Paper*  
*March 2005*

# Mastering XML DB Storage in Oracle Database 10g Release 2

Introduction.....	1
Unstructured Storage of XML Documents.....	1
Creating Indexes on XMLType Columns Using CLOB Storage .....	1
Function-Based Index .....	2
CTXXPath Index .....	3
Usage Guidelines .....	4
Structured Storage of XML Documents .....	4
Annotating an XML Schema to Control Naming, Mapping, and Storage .....	5
Understanding XQuery and XPath Rewrite.....	5
Using Indexes to Improve Performance of XPath- and XQuery- Based Functions .....	5
Storage Options for Collections.....	7
CLOB.....	7
VARRAY as LOB .....	8
VARRAY as Nested Table .....	9
VARRAY of REF XMLType as LOB.....	10
VARRAY of REF XMLType as Nested Table .....	11
Conclusion .....	12

# Mastering XML DB Storage in Oracle Database 10g Release 2

As the volume of XML documents processed by web applications rapidly multiplies, optimal storage and retrieval of XML documents become critical for scalable and high performance web applications.

## INTRODUCTION

As the volume of XML data processed by web applications rapidly multiplies, optimal storage and retrieval of XML data become critical for scalable and high performance web applications. Built on the solid foundation of Oracle Database, Oracle XML DB offers an exceptional array of XML storage options to pillar the increasing load of these applications.

The heart of Oracle XML DB is the *XMLType*, a data type introduced since Oracle9i Release 1. As an object type, *XMLType* supports a comprehensive set of built-in methods for efficient processing of XML documents throughout their life cycles. Instances of an *XMLType* can be stored using unstructured- or structured storage. In the following sections, the most prominent XML storage approaches in Oracle XML DB will be delineated. Guidelines for their pertinent usage will also be provided.

## UNSTRUCTURED STORAGE OF XML DOCUMENTS

Unstructured storage provides the highest possible throughput when inserting and retrieving entire XML documents.

With Unstructured storage, an entire XML document is stored as a whole in a Character Large Object (CLOB). Unstructured storage provides the highest possible throughput when inserting and retrieving entire XML documents. Unstructured storage maintains document fidelity by preserving the original XML document, including white spaces. It also provides the greatest degree of flexibility in terms of the structure of the XML that can be stored in an *XMLType* table or column. These throughput and flexibility benefits come at the expense of certain aspects of intelligent processing. For example, XML fragment extraction and update will require DOM-level operations, which can be expensive with large XML documents or large number of small XML documents. However, CTXXPATH indexes and function-based indexes can be created for unstructured storage to optimize queries.

### Creating Indexes on XMLType Columns Using CLOB Storage

One of the issues with unstructured storage is its costly functional evaluation of XPath expressions on XML documents. Functional evaluation materializes XML documents into the memory to construct DOM trees before processing any XPath expressions in SQL statements.

To alleviate this drawback, there are two types of indexes (i.e., the function-based index and the CTXPath index) available for certain cases of processing XML documents stored in unstructured storage.

#### Function-Based Index

A function-based index is created by evaluating the specified functions of XPath expressions for each row in the table. Function-based indexes can be built only on XPath expressions returning a single value. A function-based index cannot be created if the XPath expression returns more than one value.

Given the table created in the example below using CLOB storage to store XML documents, the following CREATE INDEX statement will result in a function-based index being created on the value of the text node belonging to the Reference element. As the example shows, this index will enforce the unique constraint on the value of the text node associated with the Reference element.

```
create table PURCHASEORDER_CLOB of XMLTYPE
XMLType store as CLOB
ELEMENT
"http://localhost:8080/home/SCOTT/poSource/xsd/purchaseO
rder.xsd#PurchaseOrder";
```

Table created.

```
--
insert into PURCHASEORDER_CLOB
select object_value from PURCHASEORDER;
```

134 rows created.

```
--
create unique index IPURCHASEORDER_REFERENCE
on PURCHASEORDER_CLOB
(extractValue(object_value, '/PurchaseOrder/Reference'));
```

Index created.

```
--
insert into PURCHASEORDER_CLOB VALUES (
xmltype (bfilename('XMLDIR',
'EABEL-20021009123335791PDT.xml'),
nls_charset_id('AL32UTF8')));
insert into PURCHASEORDER_CLOB*
```

```
ERROR at line 1:
ORA-00001: unique constraint
(SCOTT.IPURCHASEORDER_REFERENCE) violated
```

Keep in mind when creating and using function-based indexes that the cost-based optimizer will only consider using the index when the function included in the WHERE clause is identical to the function used to create the index. Furthermore, function-base index and the corresponding query must be using the extractValue() function, not the existsNode() function.

### CTXXPath Index

Function-based index requires you to be aware in advance of the set of XPath expressions that will be used when searching XML content. Oracle XML DB also makes it possible to create a CTXXPATH index—a general purpose XPath-based index, using Oracle Text technology. CTXXPATH index can be used to improve the performance of the existsNode() SQL function, which is used for testing existence of node(s) using equality predicates. The CTXXPATH index is best used when only a small number of XML documents will match the supplied XPath expression.

The CTXXPATH index is based on Oracle Text Technology. It is designed to re-write the XPath expression supplied to existsNode(). Once re-written, underlying text index can be used as a primary filter to quickly locate a superset of the documents that match the supplied XPath expression. Each document identified by the text index is then checked, using a DOM-based evaluation, to ensure that it is a true match for the supplied XPath expression.

With CTXXPATH index, Data Manipulation Language (DML) operations such as inserting, updating, and deleting are asynchronous. You must use a special command to synchronize the DML operations, in a similar manner to Oracle Text index. Despite the asynchronous nature of DML operations, CTXXPATH indexing still follows the transactional semantics of existsNode() by also returning unindexed rows as part of its result set in order to guarantee its requirement of returning a superset of the valid results.

The example below uses CTXXPATH index and existsNode() for XPath Searching of XML documents.

```
create index PURCHASEORDER_CLOB_XPATH
on PURCHASEORDER_CLOB (object_value)
indextype is CTXSYS.CTXXPATH;
```

Index created.

```
--
explain plan for
select
extractValue(object_value, '/PurchaseOrder/Reference')
  from PURCHASEORDER_CLOB where existsNode(object_value,
'//LineItem/Part[@Id="715515011624"]') = 1;
```

Explained.

--

PLAN\_TABLE\_OUTPUT

-----  
Plan hash value: 2191955729

```
-----  
| Id | Operation | Name | Rows |  
-----  
| 0 | SELECT STATEMENT | | 1 |  
|* 1 | TABLE ACCESS BY INDEX ROWID | PURCHASEORDER_CLOB | 1 |  
-----
```

Predicate Information (identified by operation id):

```

1 -
filter (EXISTSNODE (SYS_MAKEXML ('3A7F7DBBEE5543A486567A908
C71D65A', 3664, "PURCHASEORDER_CLOB"."XMLDATA"),
'//LineItem/Part[@Id="715515011624"]')=1)

```

Note

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

19 rows selected.

### Usage Guidelines

Unstructured storage for XMLType is the best choice in the following cases:

- When you require storage and retrieval of XML documents in their entirety.
- When piece-wise updates on XML documents are not required.

### STRUCTURED STORAGE OF XML DOCUMENTS

Structured storage has numerous advantages in managing XML documents, including optimized memory management, reduced storage requirements, B-tree indexing, and in-place updates. For complex-structured XML documents, structured storage provides a number of options allowing optimal storage of collections according to actual usage scenarios. Structured storage does require somewhat increased processing of the corresponding XML schema during ingestion and retrieval of entire documents.

Structured storage of XML documents is based on decomposing the content of the document into a set of objects. These objects are based on the SQL 1999 standard. When an XML schema is registered with Oracle XML DB, the required type definitions are automatically generated from the XML schema.

A database-native type definition is generated from each `complexType` defined by the XML schema. Each element or attribute defined by the `complexType` becomes an attribute in the corresponding database-native type. To provide a database-native XML object model, Oracle XML DB automatically maps the scalar data types defined by the W3C XML Schema Recommendation to the database-native scalar datatypes.

The generated database-native types allow XML content, compliant with the XML schema, to be decomposed and stored in the database as a set of objects without any loss of information. When the document is ingested, the constructs defined by the XML schema are mapped directly to the appropriate database-native types. This allows Oracle XML DB to leverage the full power of Oracle Database when managing XML and can lead to significant reductions in the

For complex-structured XML documents, structured storage provides a number of options for optimal storage of collections according to actual usage scenarios.

amount of space required to store the document. It can also reduce the amount of memory required to query and update XML content.

### **Annotating an XML Schema to Control Naming, Mapping, and Storage**

The W3C XML Schema Recommendation defines an annotation mechanism that allows vendor-specific information to be added to an XML schema. Although annotations are not required by the Oracle XML DB to register schemas, Oracle XML DB allows application developers and DBAs to control the mapping between the XML schema and the database-native object model by using annotations.

Annotating an XML schema allows control over the naming of the database-native objects and attributes created. Annotations can also be used to override the default mapping between the XML schema data types and database-native data types and to specify which table should be used to store the data.

### **Understanding XQuery and XPath Rewrite**

Rewrite of XQuery and XPath expressions is the key to improving the performance of SQL/XML functions containing XQuery and XPath expressions. It converts the functions into compiled database internal query representation. This insulates the database optimizer from having to understand the XQuery and XPath expressions and the XML data model. The database optimizer can efficiently process the compiled query representation in the same manner as any other compiled SQL statement. In this way, it can derive an execution plan based on conventional relational algebra. This leads to the execution of SQL statements with XQuery and XPath expressions with near relational performance. As a result, XML DB applications can reach optimal scalability.

For XQuery and XPath rewrite to take place, the following conditions must be satisfied:

- The `XMLType` column or table containing the XML documents must be based on a registered XML schema.
- The `XMLType` column or table must be stored using structured (object-relational) storage techniques.
- It must be possible to map the nodes referenced by the XQuery or the XPath expression to attributes of the underlying database-native object model.

### **Using Indexes to Improve Performance of XPath- and XQuery-Based Functions**

For structured storage, Oracle XML DB supports the creation of three kinds of index on XML content:

- **B-Tree indexes.** When the `XMLType` table or column is based on structured storage techniques, conventional B-Tree indexes can be created on underlying database-native types.

- **Function-based indexes.** These can be created on any XMLType table or column.
- **Text-based indexes.** These can be created on any XMLType table or column.

Indexes are typically created by using SQL function `extractValue`, although it is also possible to create indexes based on other functions such as `existsNode`. During the index creation process Oracle XML DB uses XPath rewrite to determine whether it is possible to map between the nodes referenced in the XPath expression used in the `CREATE INDEX` statement and the attributes of the underlying database-native types. If the nodes in the XPath expression can be mapped to attributes of the database-native types, then the index is created as a conventional B-Tree index on the underlying database-native objects. If the XPath expression cannot be rewritten then a function-based index is created.

The example below shows creation of index `purchaseorder_user_index` on the value of the `User` element text node.

```
CREATE INDEX purchaseorder_user_index
  ON purchaseorder(extractValue(OBJECT_VALUE,
  '/PurchaseOrder/User'));
```

At first glance, the index appears to be a function-based index. However, where the XMLType table or column being indexed is based on structured storage, XPath rewrite determines whether the index can be re-stated as an index on the underlying SQL types. In this example, the `CREATE INDEX` statement results in the index being created on the `userid` attribute of the `purchaseorder_t` object.

The following `EXPLAIN PLAN` is generated when the same query is executed after the index has been created. It shows that the query plan will make use of the newly created index. The new execution plan is much more scalable.

```
EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE,
  '/PurchaseOrder/Reference') "Reference"
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
  '/PurchaseOrder[User="SBELL"]') = 1;
```

Explained.

PLAN\_TABLE\_OUTPUT

Plan hash value: 713050960

Id	Operation	Name	Rows
0	SELECT STATEMENT		1
1	TABLE ACCESS BY INDEX ROWID	PURCHASEORDER	1
* 2	INDEX RANGE SCAN	PURCHASEORDER_USER_INDEX	1

Predicate Information (identified by operation id):

```
-----  
      2 - access("PURCHASEORDER"."SYS_NC00022$"='SBELL')
```

18 rows selected.

One key benefit of the relational database is that you do not need to change your application logic when the indexes change. This is also true for XML applications that leverage Oracle XML DB capabilities. The optimizer automatically uses the index whenever it is appropriate.

### Storage Options for Collections

When structured storage is selected, collections (elements which have `maxOccurs > 1`, allowing them to appear multiple times) are mapped into `VARRAY` values.

Schema annotations can be used to control how collections in an XML document are stored in the database. These storage options allow you to tune the performance of applications that use `XMLType` datatypes to store XML in the database. There are currently five options:

- Character Large Object (CLOB). The entire set of elements is persisted as XML text stored in a CLOB column.
- `VARRAY` in a LOB. Each element in the collection is converted into a database-native object. The collection of database-native objects is serialized and stored in a LOB column.
- `VARRAY` as Nested Table. Each element in the collection is converted into a database-native object. The collection of database-native objects is stored as a set of rows in an Index Organized Table (IOT).
- `VARRAY` of `REF XMLType` as LOB. Each element in the collection is treated as a separate `XMLType` value. The collection of `XMLType` values is stored as a set of rows in an `XMLType` table. The `VARRAY` of `REF XMLType` is stored as a LOB.
- `VARRAY` of `REF XMLType` as Nested Table. Similar to the above option, each element in the collection is treated as a separate `XMLType` value. The collection of `XMLType` values is stored as a set of rows in an `XMLType` table. The main difference from the above option is that the `VARRAY` of `REF XMLType` is now stored as Nested Table.

### CLOB

With the CLOB storage option, the entire set of elements is persisted as XML text stored in a CLOB column. This storage option for collections is similar to the unstructured storage for XML documents. If the usage scenario of an application only fetches collections in their entirety, CLOB storage will have

lower memory and processing requirements. It reduces complexity while also providing fast ingestion and retrieval of collections.

However, evaluating XPath expressions of collections stored as CLOBs relies on costly DOM-based functional evaluation. Again, similar to unstructured storage for XML documents, function-base indexes and CTXPath indexes can be used to make some improvements in certain cases. Another disadvantage of CLOB storage option arises when updates of XML fragments inside collections are required. An update of a small fragment inside a large collection will still require the entire collection to be parsed and rewritten.

#### **XML Schema Annotation for CLOB storage**

The XML Schema annotation below is used to specify CLOB storage:

```
xdb:SQLType="CLOB"
```

When this annotation is used with global elements, the entire XML document will be stored as a CLOB. If it is used with local elements and complexTypes, the element and all its descendants will be stored as a CLOB. The parent database-native type will contain an attribute of CLOB data type. The example below demonstrate how ListItem collection is stored in a CLOB.

```
<xs:complexType name="LineItemsType"
  xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="ListItem"
      type="ListItemType"
      maxOccurs="unbounded"
      xdb:SQLName="LINEITEM"
      xdb:SQLType="CLOB"/>
  </xs:sequence>
</xs:complexType>
```

```
SQL> describe LINEITEMS_T
LINEITEMS_T is NOT FINAL
Name                               Null?                                Type
-----
SYS_XDBPD$                          XDB.XDB$RAW_LIST_T
LINEITEM                              CLOB
```

#### **Usage Guidelines**

CLOB storage for collections is the best choice in the following cases:

- You need to reduce parsing and generation overhead associated with a subtree in the document.
- You need to avoid the 1000-column limit by storing all of the elements and attributes as a single column.

#### **VARRAY as LOB**

By default, the annotation `SQLInline` has the value "true", the entire contents of a VARRAY is serialized using a single LOB column. This storage

model provides for optimal ingestion and retrieval of the entire document. It does have significant limitations when it is necessary to index, update, or retrieve individual members of the collection. The name of the `VARRAY` type is either explicitly specified by the user using `SQLCollType` attribute or obtained by mangling the element name.

The example below shows how the `LineItems` collection in the `po.xsd` schema is mapped by default to a `VARRAY` as LOB in SQL. In addition, the schema annotation `xdb:SQLCollType="LINEITEM_V"` defines the typename of the `VARRAY`.

```
create table PURCHASEORDER of XMLTYPE
xmlSchema "http://myhost:8080/home/SCOTT/xsd/po.xsd"
element "PurchaseOrder"
/
SQL> describe LINEITEM_V
LINEITEM_V VARRAY(2147483647) OF LINEITEM_T
LINEITEM_T is NOT FINAL
Name                               Null?                                Type
-----
SYS_XDBPD$                          XDB.XDB$RAW_LIST_T
ITEMNUMBER                           NUMBER(38)
DESCRIPTION                          VARCHAR2(256 CHAR)
PART                                   PART_T
```

#### **VARRAY as Nested Table**

`VARRAY` can also be stored as Nested Table to avoid some of the disadvantages of being stored as LOB. In this case, each member of the collection is persisted as a separate row in a Nested Table. Nested tables can in turn be parents of other nested tables to allow collections within collections.

With this storage option, XPath expressions can now be evaluated through XPath re-write. To further improve performance, a number of indexing schemes are available for this storage option. For example, B-Tree indexes can be created on columns of the Nested Table; function-based indexes can be created for commonly used XPath expressions; and CtxXPath indexes can also be created to optimize the processing of `existsNode()` processing.

#### **XML Schema Annotation for Nested Table Storage**

The XML Schema annotation below is used to specify Nested Table storage:

```
xdb:storeVarrayAsTable="true"
```

This is specified in the root element of the XML schema to force all collections to be stored as Nested Tables. The default use of LOBs to persist `VARRAYs` is overridden. A Nested Table is created for each element that specifies `maxOccurs > 1`. The Nested Tables will be created with system-generated names when tables are generated by `registerSchema()`.

### Renaming Nested Tables

Since system-generated Nested Tables may be used in many other situations (e.g., indexing), they can be renamed to be more meaningful for end users. In the example below, we first identify Nested Tables in the user schema and later rename them into more meaningful names.

```
SQL> select table_name, table_type_name,
parent_table_column
       2  from user_nested_tables
       3  connect by PRIOR TABLE_NAME = PARENT_TABLE_NAME
       4  start with PARENT_TABLE_NAME = 'PURCHASEORDER'
/
LEVEL TABLE_NAME  TABLE_TYPE_NAME  PARENT_TABLE_COLUMN
-----
1  "SYS_NTDFLWRMq=="  ACTION_V          "XMLDATA"."ACTIONS"."ACTION"
1  "SYS_NTDFLwYKWq=="  LINEITEM_V       "XMLDATA"."LINEITEMS"."LINEITEM"

SQL> rename "SYS_NTDFLWRMq==" to ACTION_TABLE
/

SQL> rename "SYS_NTDFLwYKWq==" to LINEITEM_TABLE
/
```

### Usage Guidelines

- When XPath expressions in DML or DDL statements frequently access children elements of member elements of a collection.
- When the usage scenario requires frequent updates of XML fragments of children elements of member elements of a collection.

### VARRAY of REF XMLType as LOB

Another storage option to treat each element in the collection as a separate XMLType value. The collection of XMLType values is stored as a set of rows in an XMLType table. This is also known as ‘Out-Of-Line’ storage. The parent table contains a VARRAY of REF XMLType that manages the members of the collection. Each member of the VARRAY of REF values to XMLType will point at one row in the Out-Of-Line table. The VARRAY of REF XMLType is stored as a LOB by default.

With elements of a collection stored in an XMLType table, XPath rewrite will be enabled by the XML DB to evaluate XPath expressions. For partial updates of an XML document, the XPath expression in the update statement can also be rewritten to change only the specific element(s). Query performance will also benefit from various indexing options (e.g., B-Tree index, function-based index).

### XML Schema Annotation for VARRAY of REF XMLType as LOB

If `SQLInline="false"`, then the database-native type is set to `XDB.XDB$XMLTYPE_REF_LIST_T`, a predefined type representing a VARRAY

of REF values to XMLType. Furthermore, xdb:defaultTable can be used to provide an explicit name for the generated XMLType table.

In the example below, the XML schema is annotated to produce the mapping of the collection of LineItem elements into a VARRAY of REF values to XMLType.

```
<xs:complexType name="LineItemsType"
xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType"
      maxOccurs="unbounded" xdb:SQLName="LINEITEM"
      xdb:SQLInline="false"
      xdb:defaultTable="LINEITEM_TABLE"/>
  </xs:sequence>
</xs:complexType>
```

```
SQL> describe LINEITEMS_T
LINEITEMS_T is NOT FINAL
Name                               Type
-----
SYS_XDBPD$                         XDB.XDB$RAW_LIST_T
LINEITEM                            XDB.XDB$XMLTYPE_REF_LIST_T
```

```
SQL> describe XDB.XDB$XMLTYPE_REF_LIST_T
XDB.XDB$XMLTYPE_REF_LIST_T VARRAY(2147483647) OF REF
XMLTYPE
```

```
SQL> describe LINEITEM_TABLE
TABLE of SYS.XMLTYPE ( XMLSchema "PurchaseOrder.xsd"
Element "LineItem")
STORAGE Object-relational TYPE "LINEITEM_T"
```

#### **Usage Guidelines**

Since VARRAY of REF values to XMLType is stored as a single BLOB by default, the out-of-line storage model does not support reverse tree navigation. We cannot create an index to support finding the parent for a given child.

#### **VARRAY of REF XMLType as Nested Table**

As a slight variation to the previous storage option, instead of storing the VARRAY of REF values to XMLType as a LOB by default, we can annotate the XML schema to store the VARRAY as Nested Table. With this storage option, efficient reverse tree navigation via an index becomes possible.

In this storage structure, each row in the Nested Table consists of a NESTED\_TABLE\_ID column and a REF XMLType column (COLUMN\_VALUE). All of the REF XMLType values point to the same target XMLType table. We can further specify the scoping of the REF XMLType to enable the join operation for tree navigation. Creating an index on COLUMN\_VALUE will provide further optimization of reversed tree navigation.

### **XML Schema Annotation for VARRAY of REF XMLType as Nested Table**

The same as what we have described for the Nested Table storage option, the XML Schema annotation below is used to specify Nested Table storage:

```
xdb:storeVarrayAsTable="true"
```

This is specified in the root element of the XML schema to force all collections to be stored as Nested Tables. The default use of LOBs to persist VARRAYs is overridden. A Nested Table is created for each element that specifies maxOccurs > 1. The Nested Tables will be created with system-generated names when tables are generated by registerSchema().

In the example below, a scope is added for the REF XMLType column.

```
SQL> select table_name, table_type_name,
parent_table_column
       2   from user_nested_tables
       3   connect by PRIOR TABLE_NAME = PARENT_TABLE_NAME
       4   start with PARENT_TABLE_NAME = 'PURCHASEORDER'
```

```
LEVEL TABLE_NAME TABLE_TYPE_NAME PARENT_TABLE_COLUMN
-----
1 "SYS_NTDfLwRMq==" XDB$XMLTYPE_REF_LIST_T
"XMLDATA"."ACTIONS"."ACTION"
1 "SYS_NTDfLwYKWq==" XDB$XMLTYPE_REF_LIST_T
"XMLDATA"."LINEITEMS"."LINEITEM"
```

```
SQL> rename "SYS_NTDfLwYKWq==" to LINEITEM_REF_TABLE
```

```
SQL> desc LINEITEM_REF_TABLE
```

```
      Name                                         Type
-----
COLUMN_VALUE                                     REF OF XMLTYPE
```

```
SQL> alter table LINEITEM_REF_TABLE
      add (scope for (COLUMN_VALUE) is LINEITEM_TABLE)
```

```
SQL> create LINEITEM_INDEX on LINEITEM_REF_TABLE
      (COLUMN_VALUE, NESTED_TABLE_ID)
```

### **Usage Guidelines**

This storage option should be used if reversed tree navigation is required. To further improve performance of tree navigation in XPath expressions, REFS can be scoped to the out-of-line XMLType table.

### **CONCLUSION**

XML DB in Oracle Database 10g offers an exceptional array of storage options for optimal storage and retrieval of XML documents in diverse usage scenarios.

XML DB in Oracle Database 10g offers an exceptional array of storage options for optimal storage and retrieval of XML documents in diverse usage scenarios. The table below summarizes advantages and disadvantages to consider when selecting Oracle XML DB storage options for your specific usage scenarios.

<b>Feature</b>	<b>Unstructured Storage (with Oracle Text Index)</b>	<b>Structured Storage (with B*Tree index)</b>
Database schema	Very flexible when schemas change.	Limited flexibility for schema changes. Similar to the ALTER

Feature	Unstructured Storage (with Oracle Text Index)	Structured Storage (with B*Tree index)
flexibility		TABLE restrictions.
Data integrity and accuracy	Maintains the original XML content fidelity, important in some applications.	Trailing new lines, whitespace within tags, and data format are lost. DOM fidelity is maintained.
Performance	Mediocre DML performance.	Excellent DML performance.
Access to SQL	Some accessibility to SQL features.	Good accessibility to existing SQL features, such as constraints, indexes, etc.
Space needed	Can consume considerable space.	Needs less space in particular when used with a registered XML schema.

For Oracle XML DB to efficiently process the collection members, it is crucial to select the correct storage structure for collections.

Furthermore, the majority of XML documents contain collections of repeating elements. For Oracle XML DB to efficiently process the collection members, it is crucial to select the correct storage structure for collections. A correct storage structure makes it possible to index elements within the collection and to perform direct operations on individual elements within the collection.

Oracle XML DB offers the following ways to manage the members of a collection:

- When a collection is stored as a CLOB value, you cannot directly access its members. Storing the members as XML text in a CLOB value means that any operation on the collection requires parsing the contents of the CLOB and then using functional evaluation to perform the required operation.
- When a VARRAY is stored as a LOB, you cannot directly access members of the collection. Converting the collection into a set of database-native objects that are serialized into a LOB removes the need to parse the documents. However, any operations on the members of the collection still require that the collection be loaded from disk into memory before the necessary processing can take place.
- When a VARRAY is stored as a Nested Table, each member of the VARRAY becomes a row in a table, so you can directly access members of the collection.
- When a VARRAY is stored as an XMLType value, each member of the collection becomes a row in an XMLType table. Members of the collection can also be directly accessed. Depending on your requirements, the VARRAY of REF XMLType can be stored as LOB or Nested Table. If reversed tree

navigation is required, Nested Table should be used for storing the the  
VARRAY of REF XMLType .

Finally, with the in-depth knowledge you have learned from this paper about XML DB storage options in Oracle Database 10g, you can now masterfully develop, deploy, and manage web applications to provide optimal performance for storing and retrieving XML data.



Mastering XML DB Storage in Oracle Database 10g Release 2

March 2005

Author: Geoff Lee

Contributing Authors: Mark Drake

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

[www.oracle.com](http://www.oracle.com)

Oracle Corporation provides the software  
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various  
product and service names referenced herein may be trademarks  
of Oracle Corporation. All other product and service names  
mentioned may be trademarks of their respective owners.

Copyright © 2005 Oracle Corporation

All rights reserved.