Hands-on Lab: An Introduction to Oracle XML DB in Oracle Database 12c

Mark Drake
Manager, Product Management
Oracle XML DB

Contents

Using Oracle XML DB to Optimize Performance and Manage Structured XML Data

Purpose

This tutorial shows you how to store, index and query XML data in Oracle XML Database.

Time to Complete

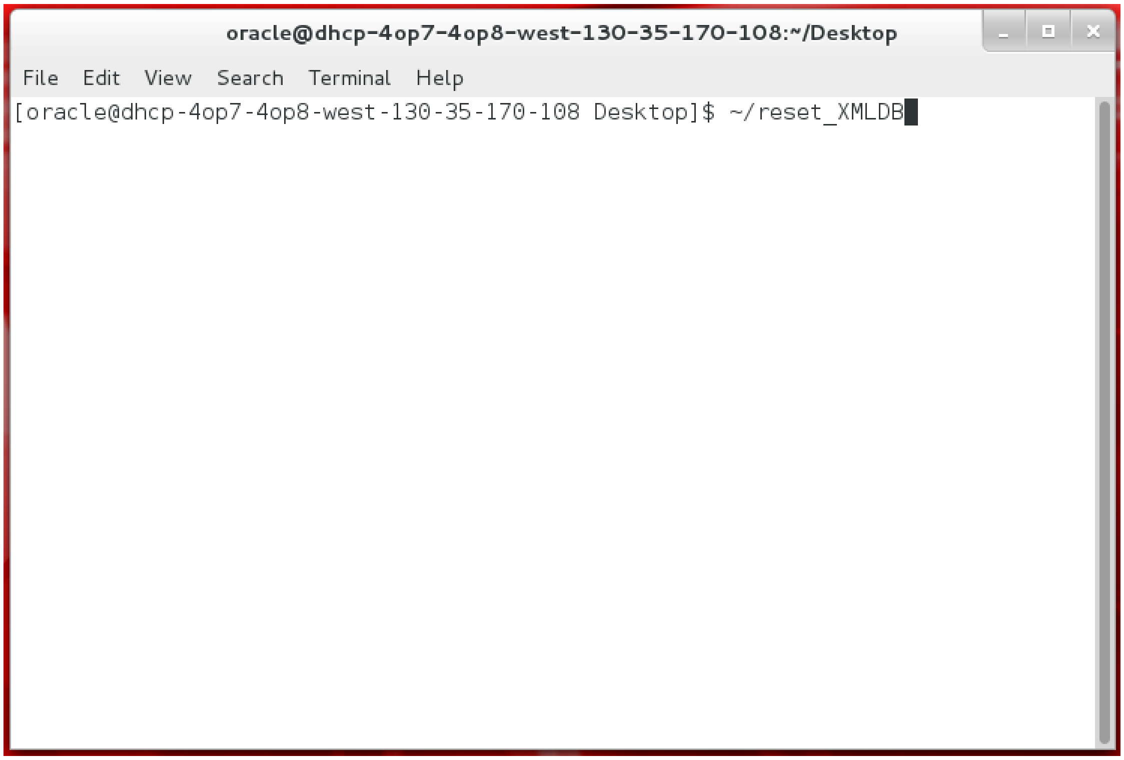
Approximately 120 minutes.

Initialize the Hands on Lab

Run the reset script

1. Open a command prompt by clicking on the terminal icon found on the desktop .

2. When the terminal window opens, run the reset script by typing ~/**reset_xmldb** at the command prompt.



```
[oracle@dhcp-4op7-4op8-west-130-35-170-108 Desktop]$ ~/reset_XMLDB
```

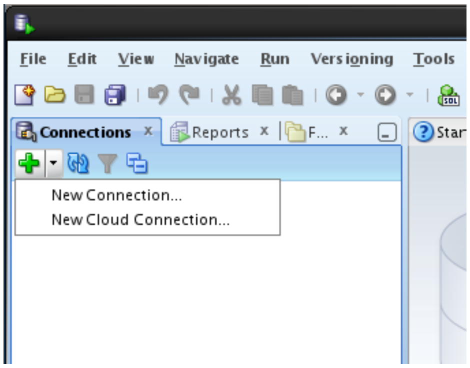
3. Once the script has completed type **CTRL+D** to close the terminal window.

Use SQL Developer to establish a connection to the OE schema.

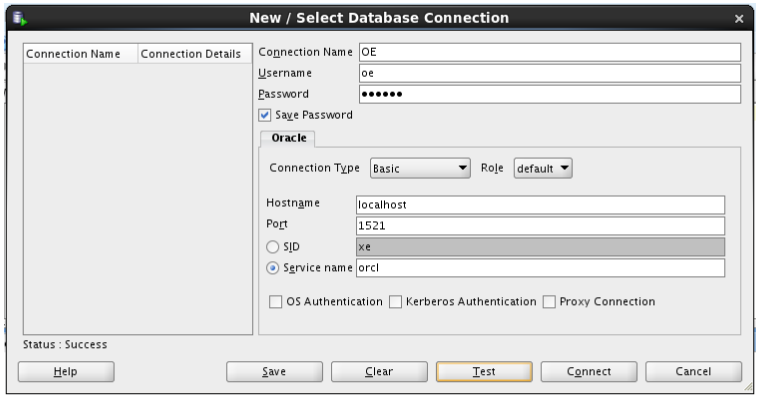
1. Click the SQL Developer icon on the desktop to start the application.

2. Create a database connection as user **oe** by performing the following steps.

3. In the Connections tab, right-click **Connections** and select **New Connection**.



The **New / Select Database Connection** window opens.



Connection Name: OE
Username: oe
Password: ••••••
☑ Save Password
Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
○ SID: xe
● Service name: orcl
□ OS Authentication □ Kerberos Authentication □ Proxy Connection
Status : Success

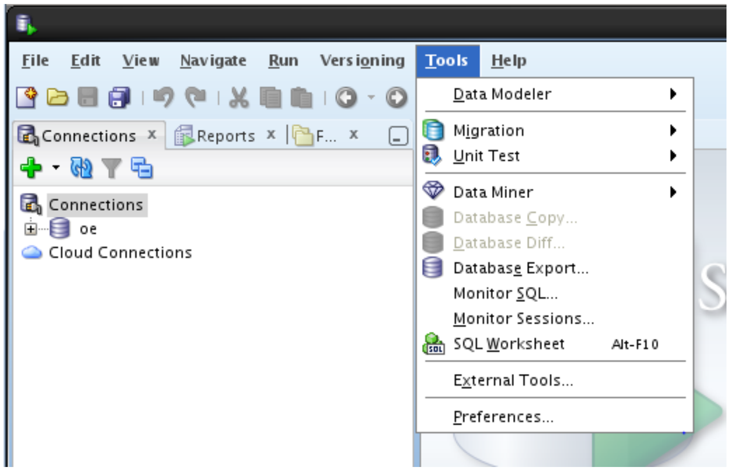
4. Enter the following details

   Connection Name: **oe**
   UserName: **oe**
   Password:**oracle**
   Hostname: **localhost**
   Port: **1521**
   Service name: **orcl**

5. Click **Test** to make sure that the connection has been set correctly. Select the **Save Password** check box, to ensure that the password is saved. Click **Connect** after the test status shows success.

Set the Autotrace parameters.

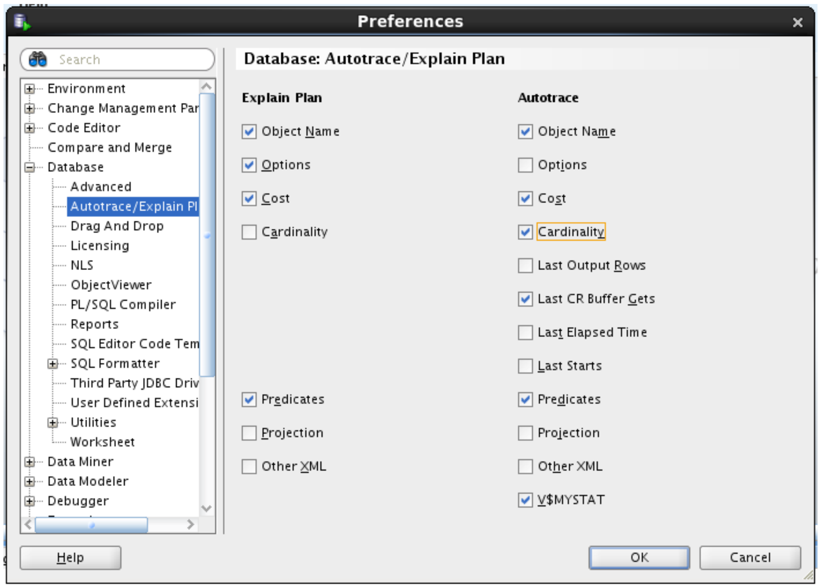
1. Go to Tools > Preferences.



2. Expand Database, and select Autotrace Parameters.

3. Make sure to select the following check boxes

Object_Name
Cost
Cardinality
Predicates



Database: Autotrace/Explain Plan

Explain Plan	Autotrace
☑ Object Name	☑ Object Name
☑ Options	□ Options
☑ Cost	☑ Cost
□ Cardinality	☑ Cardinality
	□ Last Output Rows
	☑ Last CR Buffer Gets
	□ Last Elapsed Time
	□ Last Starts
☑ Predicates	☑ Predicates
□ Projection	□ Projection
□ Other XML	□ Other XML
	☑ V$MYSTAT

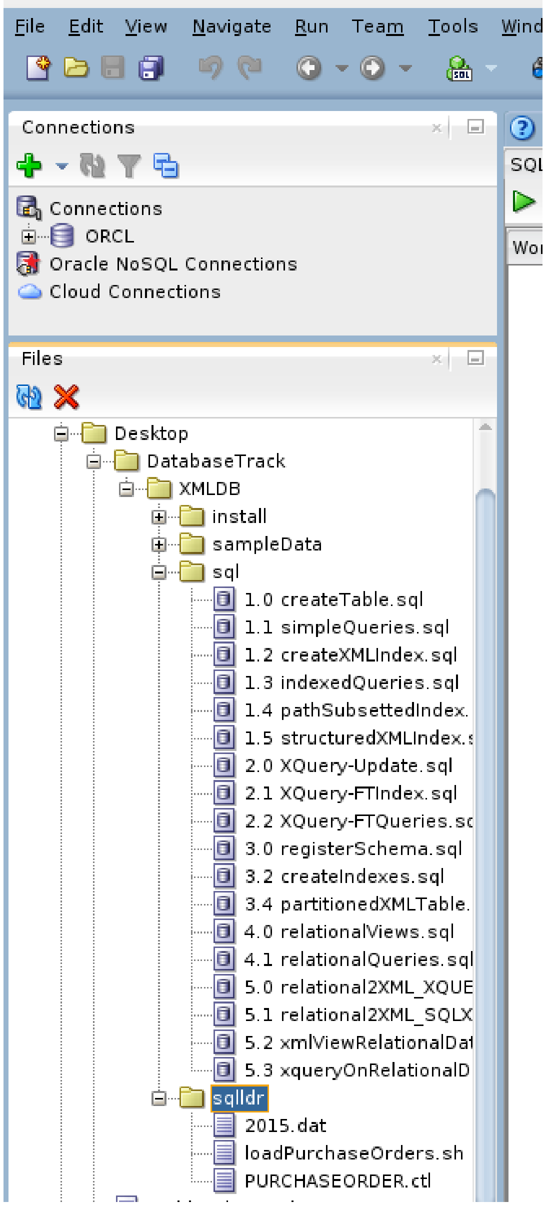
Click **OK**.

Using XQuery in Oracle Database 11g Release 2

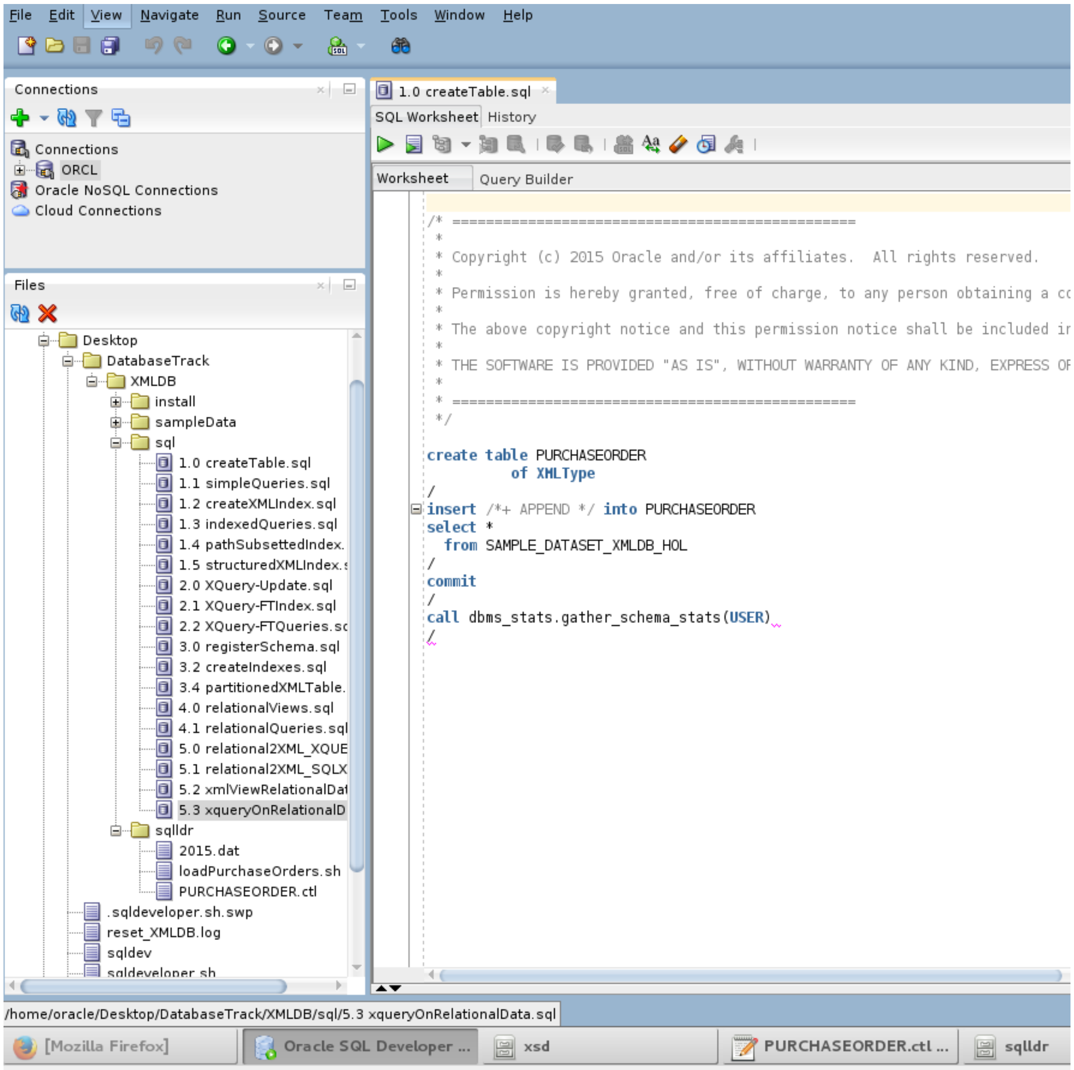
In this section, you will use XQuery and SQL/XML to query XML documents stored in the `PURCHASEORDER` table in the OE Schema.

Create a simple XMLType table.

1. Using the left window pane, click the **Files** tab and select the **createTable** script. If the **Files** tab is not visible, select it from the **View** menu. Expand the folder tree and locate the folder containing the SQL script files by selecting **MyComputer** -> **Desktop** -> **Database_Track** -> **XMLDB** -> **sql**



Right click on the file **1.0 createTable.sql** and select **open.** This will open the file as a SQL Worksheet in SQL Developer.



```
/* ==============================================
 *
 * Copyright (c) 2015 Oracle and/or its affiliates.  All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a co
 *
 * The above copyright notice and this permission notice shall be included ir
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OF
 *
 * ==============================================
 */

create table PURCHASEORDER
         of XMLType
/
insert /*+ APPEND */ into PURCHASEORDER
select *
  from SAMPLE_DATASET_XMLDB_HOL
/
commit
/
call dbms_stats.gather_schema_stats(USER)
/
```

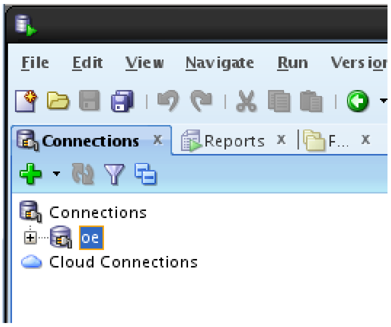
2. Click the RunScript icon to execute the script. If prompted for a connection select the OE connection that was created earlier.

   This script will create a very simple XMLType table, called **PURCHASEORDER**, using default settings. In database release 12.1.0.1.0 this will be XMLType stored as Binary XML in a Secure File LOB. Binary XML stores the XML in a "post-parsed" native XML format that allows for efficient storage, indexing and processing of XML. Since the format is "post-parsed", the database does not need to perform any additional parsing operations when performing XQuery operations on the XML content.

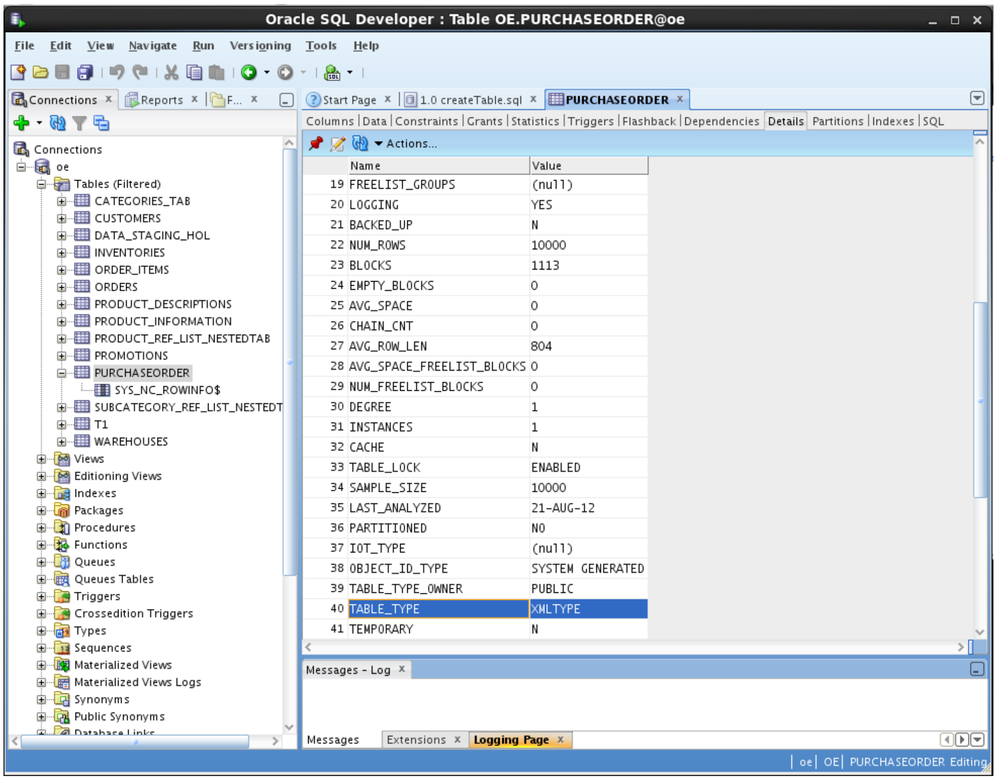
   The script then loads 10,000 XML documents into the **PURCHASEORDER** table.

Review table PURCHASEORDER in SQL Developer

1. Select the Connections table and click the **+** icon to the left of the oe connection to expand it.



2. Expand Tables, and select PURCHASEORDER. On the right window pane, select the Details tab to examine the values of the NUM_ROWS and the TABLE_TYPE parameters.



	Name	Value
19	FREELIST_GROUPS	(null)
20	LOGGING	YES
21	BACKED_UP	N
22	NUM_ROWS	10000
23	BLOCKS	1113
24	EMPTY_BLOCKS	0
25	AVG_SPACE	0
26	CHAIN_CNT	0
27	AVG_ROW_LEN	804
28	AVG_SPACE_FREELIST_BLOCKS	0
29	NUM_FREELIST_BLOCKS	0
30	DEGREE	1
31	INSTANCES	1
32	CACHE	N
33	TABLE_LOCK	ENABLED
34	SAMPLE_SIZE	10000
35	LAST_ANALYZED	21-AUG-12
36	PARTITIONED	NO
37	IOT_TYPE	(null)
38	OBJECT_ID_TYPE	SYSTEM GENERATED
39	TABLE_TYPE_OWNER	PUBLIC
40	TABLE_TYPE	XMLTYPE
41	TEMPORARY	N

Notice the highlighted TABLE_TYPE parameter with value XMLTYPE.

Accessing XML content using XQuery.

XQuery is to XML content what SQL is to relational data. The XQuery standard was developed by the W3C and as such, it is the natural language for querying, manipulating, and updating XML content. The following section will show how to use XQuery to work with XML content stored in an Oracle database**.**

1. Using the File tab, right click on the file **1.1 simpleQueries.sql** and select **open**

2. Click the RunScript icon to execute the script.

   This script demonstrates how to use the XMLTable() operator to perform XQuery operations on XML content stored in the Oracle database. The result of an XQuery operation is a sequence of zero or more nodes. The XMLTable() operator takes each node returned by the XQuery operation and converts it into a row consisting of a single column of XMLType, allowing the result of the XQuery to be understood by tools that expect a SQL result set.

   The first query counts the number of documents in the PURCHASEORDER table using the standard XQuery function fn:collection() to access the contents of the table. fn:collection() expects a path that identifies the set of XML documents to be processed. In this case, the path is prefixed with the ‘protocol’ oradb, indicating that the components of the path should be interpreted as DATABASE_SCHEMA and TABLE.

   The second query shows how we can use predicates to restrict which documents are returned. In this case the predicate on the Reference element uniquely identifies a single document. The document is returned as an XMLType object.

   The third example shows how to pass multiple predicate values into the XQuery. In a real-world example these values would be supplied as bind variables rather than hard coded literals. Also note that the XQuery expression in this example terminates in the Reference element. Consequently the result consists of just the Reference element from the documents that match the supplied predicates.

   The fifth query shows how to use XQuery to synthesize a new document from the documents that match the supplied predicates. Note that the generated XMLType is not ‘pretty printed’. Pretty Printing is a function of serialization.

   The sixth query shows how to use XMLSerialize to format or “pretty print” the XMLType returned in the previous example. Note that the output of serialization is not an XMLType. It is can be a VARCHAR, CLOB or BLOB. This is because the serialization process has generated a textual representation of the XMLType object.

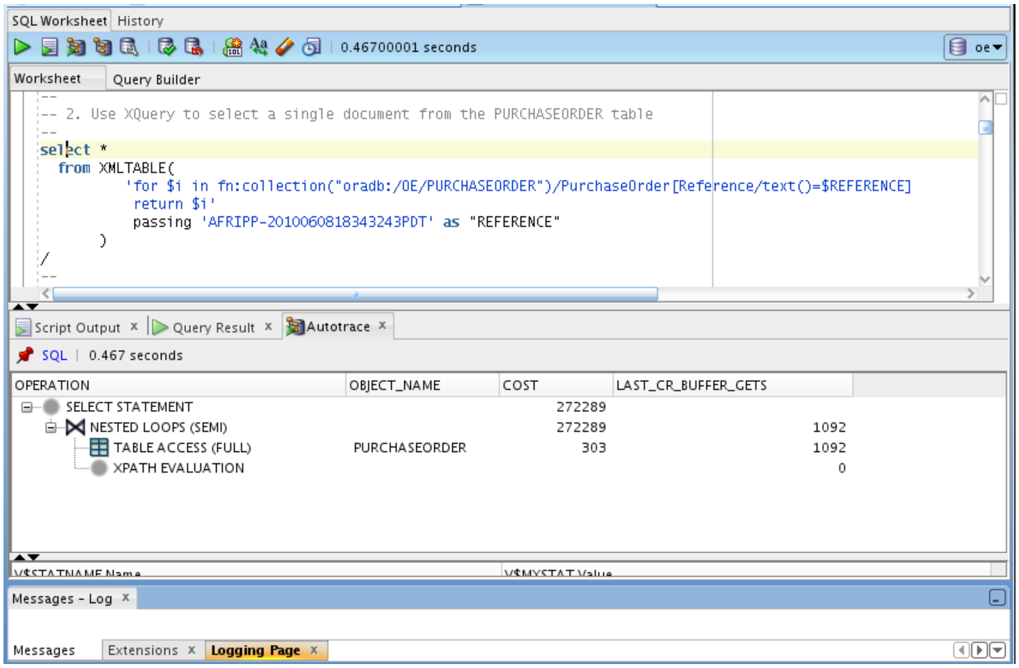
The seventh query shows how to use the columns clause of the XMLTable operator to create an inline relational view from the documents that match the supplied predicates. In this case the XQuery expression generates a result document from each of the PurchaseOrder documents that match the supplied predicates, and then the Columns clause maps elements and attributes in the result document into the columns of the in-line view. This allows a conventional relational result to be created by executing an XQuery operation on XML content.

The final query shows how XQuery can be used with relational data. In this example a join is taking place between the XML content of the PURCHASEORDER table and the content of the relational tables HR.EMPLOYEES and HR.DEPARTMENTS.

Monitoring XQuery execution using “Explain Plan”.

SQL Developer can be used to examine the execution plans for XQuery expressions, in the same way it can be used when working with SQL queries.

1. Position the cursor at the start of query #2, and click on the autotrace icon. This will show the execution plan for the query.



```
--
-- 2. Use XQuery to select a single document from the PURCHASEORDER table
--
select *
  from XMLTABLE(
          'for $i in fn:collection("oradb:/OE/PURCHASEORDER")/PurchaseOrder[Reference/text()=$REFERENCE]
           return $i'
           passing 'AFRIPP-20100608183432 43PDT' as "REFERENCE"
       )
/
--
```

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		272289	
NESTED LOOPS (SEMI)		272289	1092
TABLE ACCESS (FULL)	PURCHASEORDER	303	1092
XPATH EVALUATION			0

Since no indexes have been created on the PURCHASEORDER table the XQuery expression is executed using streaming XPath evaluation. This means that each document is examined to see if it contains a node which matches the supplied predicate. Streaming XPath is a very efficient technique for performing the equivalent of a relational full-table scan on binary XML content.

Optimizing XQuery performance with XML Indexing

Just like with SQL, XQuery operations can be improved by creating appropriate indexes. Again just like with SQL, XML indexing leads to trade-offs between DML and Query operations. This section will introduce techniques for indexing binary XML content stored in the Oracle database.

1. Using the File tab, right click on the file **1.2 createXMLIndex.sql** and select **open.**
2. Click the RunScript icon to execute the script.

This script will create a full, unstructured XML Index on the PURCHASEORDER table. The index will contain an entry for every node in every document. The full index requires no up-front knowledge about the structure of the XML being indexed. The index can be used to optimize both path (does the node exist ?), and path-value (does the node exist and does it have a particular value ?) searches.

3. Using the File tab, right click on the file **1.3 indexedQueries.sql** and select **open.**

4. Position the cursor at the start of the first query and click on the autotrace icon .

   The execution plan for the query is displayed in the autotrace window. The autotrace output shows that the XML Index is used to determine which documents match the specified predicates.

5. Repeat step 4 for each of the remaining queries.

   In each case the autotrace output shows that the XML Index is used to optimize the execution of the XQuery expressions.

Optimizing XQuery performance using a Path-Subsetted XML Index

A full unstructured index can be an expensive proposition in both disk space usage and index maintenance overhead. To address these issues XML Index provides two options.

The first option is the ability to maintain the index asynchronously. With an asynchronous index, the DML operation does not wait for indexing to complete before returning control to the application that invoked it. Instead, the indexing takes place in near real-time, ensure that index maintenance operations do not interfere with DML throughput.

The second option is the ability to use path-subsetting to explicitly include or exclude certain sections of an XML document from the index. This reduces both the size of the index and the overhead associated with index maintenance, by indexing only those nodes that will be referenced in predicates. While a path-subsetted index does not require upfront knowledge of the structure of the XML being indexed, it does require some knowledge of the kinds of searches that will be performed on the XML in question.

The next section shows how to create a path-subsetted XML index.

1. Using the File tab, right click on the file **1.4 pathSubsettedIndex.sql** and select **open.**
2. Click the RunScript icon to execute the script.

   This will re-create the index as a path-subsetted index. The index will include just the Reference and Part nodes.

3. Return to the SQL Worksheet containing the file **1.3 indexedQueries.sql**.
4. Use autotrace to generate the execution plan for each of the queries.

   The autotrace output for XQueries 1, 3 and 4 shows that the XML Index is used when evaluating the XQuery. This is because these XQuery expressions contain predicates that reference nodes that were included in the index. However, the XML Index is not used when evaluating XQuery 2, since in this case the predicate references a node that was not included in the index. In this case the explain plans show that the optimizer reverts to using streaming XPath evaluation.

Optimizing XQuery performance using a Structured XML Index

The unstructured index is best suited for working with highly unstructured XML. In situations where the XML is highly structured, but no XML Schema is available, or the XML is semi-structured, containing islands of structure floating in a sea of unstructured content, the Structured XML Index can be used to optimize operations on the XML.

With a structured XML Index, one or more XMLTable operators are used to define which nodes in the document will be indexed. The selected nodes are projected into a series of relational tables that form the basis of the index. Once the index has been created, the system will identify the XQuery expressions that can be evaluated using the index, and use the index appropriately. Secondary Indexes can be created on the tables that make up the index, enabling further performance optimizations.

Structured indexes are very useful for optimizing XPath expressions that return a set of sibling nodes, or when the predicates in an XQuery expressions reference sibling nodes. With an unstructured index, an independent index operation is required to find each of the nodes in question and then further processing is required to ensure that the selected nodes are indeed siblings. With a structured index all of the nodes for a given XPath expression are stored in the same row and can be accessed in a single index operation, and no additional processing is required for sibling verification.

The next section shows how to create a structured XML index.

1. Using the File tab, right click on the file **1.5 structuredXMLIndex.sql** and select **open.**
2. Click the RunScript icon to execute the script

   This script willl create a structured XML Index. The definition of the XML Index is provided using DBMS_XMLINDEX.createParameters. Using a parameters clause simplifies the actual DDL needed tocreate the index. This is very useful when indexing complex XML structures, with multiple levels of nesting. Note how each XMLTable operator has a table name associated with it. The script also creates B-TREE indexes on the tables underlying the index to further optimize performance.

3. Return to the SQL Worksheet containing the file **1.3 indexedQueries.sql**.
4. Use autotrace to generate the execution plan for each of the queries.

   The autotrace output shows that the Structured XML Index and associated secondary B-TREE indexes are picked-up when the XQuery expression contains nodes that are included in the index.

Updating XML Content using XQuery-Update

XQuery-Update is an extension to W3C XQuery standard that makes it possible to update the content of XML documents. XQuery update operations can modify the values of existing nodes, replace a fragment of XML with another fragment of XML and insert and remove nodes from the document. Support for XQuery-Update is available in Oracle Database 11.2.0.3.0 and later. In Oracle XML DB XQuery operations are executed using the XMLQuery Operator.

This section provides a brief introduction to the XQuery-Update standard and shows how to use the XMLQuery() operator to perform XQuery-Update operations on XML content stored in the Oracle database. While it appears that the result of an XQuery-Update operation is always a new document, underneath the covers the XQuery-Update is re-written into a series of modifications to the data stored on disc. This enables partial update of the XML document, leading to significant savings in re-indexing and undo and redo generation.

The general form of an XQuery update is

```
update tablename
     set XML = XMLQuery(XQuery-Update operation)
 where XMLExists().
```

The XMLExists operator is used to determine which documents the XQUERY-Update operation is applied to. Predicates supplied to the XQuery-Update operation will determine which nodes within the documents selected by the XMLExists operation are actually updated.

The next section shows how to use XQuery-Update

1. Using the File tab, right click on the file **2.0 XQuery-Update.sql** and select **open.**
3. Click the RunScript icon to execute the script.

   The first XQuery operation (INITIAL_STATE) generates a summary of the existing state of the document that is the target of the XQuery-Update operations.

   The first XQuery-Update operation shows how to replace the values of existing scalar nodes. Note how a predicate is used to determine which of the three description nodes in the document will be updated.

   The second XQuery operation (UPDATED_NODES) generates a summary showing the state of the document after the XQuery operation has been completed.

   The second XQuery-Update operation shows how to delete a node from the document. Again, note how a predicate is used to determine which of the LineItem nodes in the document will be deleted.

The third XQuery operation (DELETED_NODE) shows that document has been updated. Since the LineItem with an ItemNumber attribute of “2” has been removed, the LineItem whose ItemNumber attribute has the value “3” is now the 2nd LineItem element.

The third XQuery-Update operation shows how to insert a new node into the document. Again, note how a predicate is used to determine where the new LineItem node will be placed in the document.

The fourth XQuery operation (INSERTED_NODE) shows that document has been updated. Since the the new LineItem element (which has an ItemNumber of attribute of “4”) was placed after the LineItem element with an ItemNumber attribute of “3” the new LineItem is the 3rd LineItem element.

The final XQuery-Update operation shows how to update a fragment. In this case the entire LineItems element is replaced with a new LineItems element, and the changes made to the UserID and Requestor elements are reverted. Of course, since the Oracle Database provides complete transaction control as part of the SQL language the entire set up of updates could also have been undone by issuing a rollback command.

The final XQuery operation (FINAL_STATE) shows that document is back in its original state.

Searching XML Content using XQuery Full-Text

XQuery Full Text is an extension to W3C XQuery standard that makes it perform complex full-text style search operations on the content of XML documents. Support for XQuery Full-Text is available in Oracle Database 12.1.0.1.0 and later.

The original XQuery language included an operator called contains that allow pattern matching based searches on XML content. The contains operator provides a case sensitive substring style search capability, eg it returns true if the source string contains exactly the set of characters contained in the target string. The XQuery full-text specification adds the ability to perform word based searches of XML content, with all of the common features of a text retrieval system. XQuery Full-Text includes support for word match, windowing (word must appear within n words of word) and stemming (automatically recognize related words).

In order to provide an efficient and performant implementation of XQuery Full-Text, Oracle Database 12c makes use of Oracle's Text indexing technology. In order to make use of XQuery Full-Text it is necessary to create a XML Aware Full-Text index on the documents that are to be searched.

The next section shows how to create an XML Aware Full-Text index.

1. Using the File tab, right click on the file **2.1 XQuery-FTIndex.sql** and select **open.**
2. Click the RunScript icon to execute the script.

   The first step is to define the section group and storage preferences that will be used by the index. These items are managed using methods provided by the package CTX_DLL. In order to use this package the user must have been granted the role CTXAPP.

   The second step is to create a CTXSYS.CONTEXT index based on the section group and storage preferences.

Once the index has been created it can be used to optimize XQuery Full-Text operations. Currently XQuery Full-Text searches must always be performed via the XMLExists operator, to ensure that the optimizer uses the Full Text index to locate which documents match the specified search conditions. As per the XQuery standard, by default all XQuery Full-Text operations in Oracle Database 12c are case-insensitive.

The next section shows how to perform XQuery Full-Text searches on XML Content stored in Oracle XML DB.

1. Using the File tab, right click on the file **2.2 XQuery-FTQueries.sql** and select **open.**
2. Click the RunScript icon to execute the script.

The first query is a simple exact match query. The index cannot be used since the comparison is case sensitive. No results are returned since the source in mixed case and the target is in uppercase.

The second query is a simple exact match query. The index cannot be used since the comparison is case sensitive. Results are returned since the source and target are an exact match.

The third query shows the use of the XQuery contains() operator to search for an exact match on a phrase. The index cannot be used since the comparison is case sensitive.

There are a number of limitations to his kind of search. The contains() operator performs a substring Style of match, searching for the target string anywhere in the specified source. This kind of operation cannot easily be optimized using any an index. When thinking in text retrieval terms this kind of search often leads to false positives when a word in the source contains the target string. The case sensitive nature of the contains() operator also leads to false negatives unless the programmer takes care to ensure that comparison of source and target is performed in case-insensitive manner.

As can be seen in the results a search for the word ‘sport’ has returned documents that contain the term ‘transportation’ and omitted documents that contain the word ‘Sporting’.

The fourth query introduces the XQuery Full Text “contains text” operation in place of the contains() operator. The “contains text” operation searches for the target as a word in the source. Since the comparison is now based on words, rather than strings, a text index can be used to optimize the search. In this case there are no results since neither ‘transportation’ or ‘Sporting’ are an exact match for the phrase ‘sport’.

The fifth query introduces the XQuery Full Text “using stemming” operation. The “using stemming” operation searches for any word related to the target as a word in the source. Since the comparison is now based on related words, rather than exact words the query returns the documents that contain the word ‘Sporting’ since ‘sport’ is treated as a stem of ‘Sporting’.

The sixth query show that XQuery Full-Text can be used to search fragments, as well as leaf-level nodes. In this query the complex element Address is being searched for the word Oxford. The query returns documents where any child of address contains the target word. In this example the elements street and city contain the word ‘Oxford’.

The seventh query shows the use of the ftand operator to search for the presence of two words. The words do not need to be adjacent and do not need to be in any particular order.

The eighth query shows how to add a Window to the ftand operator. A window makes it possible to control how many words are allowed to exist between the two terms that are being searched on. In this case the window is restricted to 2 words and consequently no documents are returned.

The ninth query shows how expanding the size of the Window allows the search to return documents that were eliminated from the result set by the narrow window used in the previous query.

Optimizing XML Storage and processing with XML Schema.

XML schemas are used by many industries to define the XML structures used for information exchange. XML documents conforming to these XML schemas are highly structured. This tutorial will show you how an XML Schema can be used to optimize storage, indexing, query and management of structured XML data.

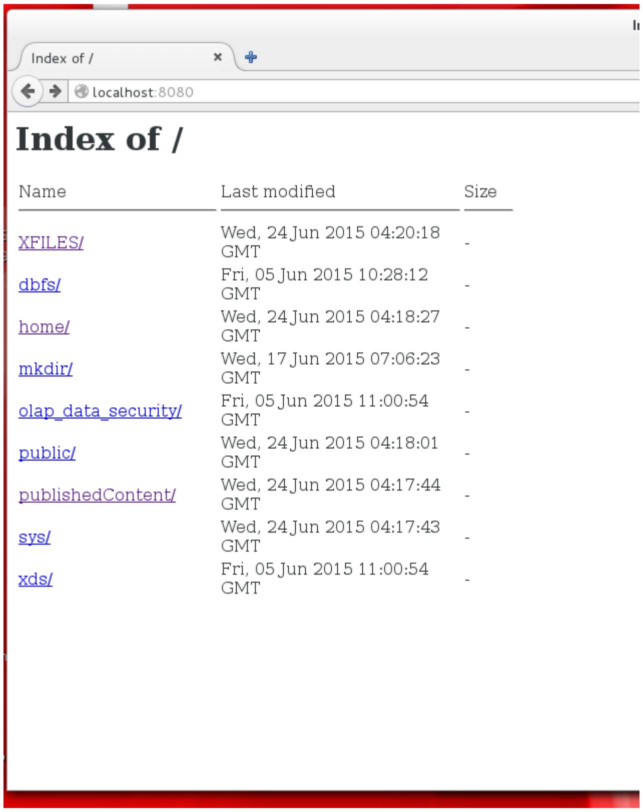
The PurchaseOrder XML Schema.

The XML Schema that is required for the next section of this Hands-on-Lab has been stored in the database as a document in the XML DB repository. Content stored in the XML DB repository can be accessed directly from a Web Browser.

1. Launch the Firefox web brower by clicking on the browser icon .

2. Open an new tab and enter the following URL : http://localhost:8080

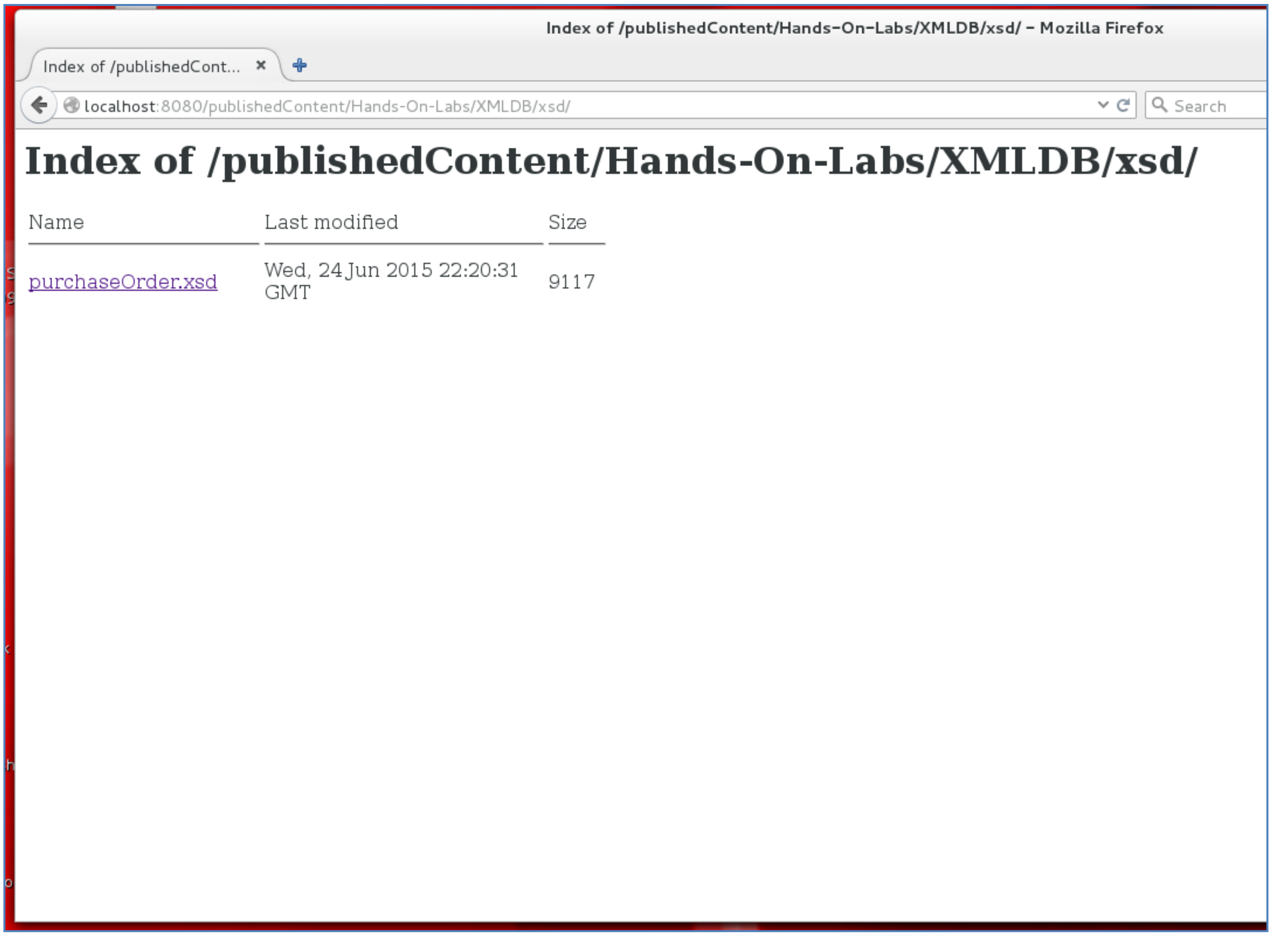
A web page similar to the one show below should be displayed.



Index of /

Name	Last modified	Size
XFILES/	Wed, 24 Jun 2015 04:20:18 GMT	-
dbfs/	Fri, 05 Jun 2015 10:28:12 GMT	-
home/	Wed, 24 Jun 2015 04:18:27 GMT	-
mkdir/	Wed, 17 Jun 2015 07:06:23 GMT	-
olap_data_security/	Fri, 05 Jun 2015 11:00:54 GMT	-
public/	Wed, 24 Jun 2015 04:18:01 GMT	-
publishedContent/	Wed, 24 Jun 2015 04:17:44 GMT	-
sys/	Wed, 24 Jun 2015 04:17:43 GMT	-
xds/	Fri, 05 Jun 2015 11:00:54 GMT	-

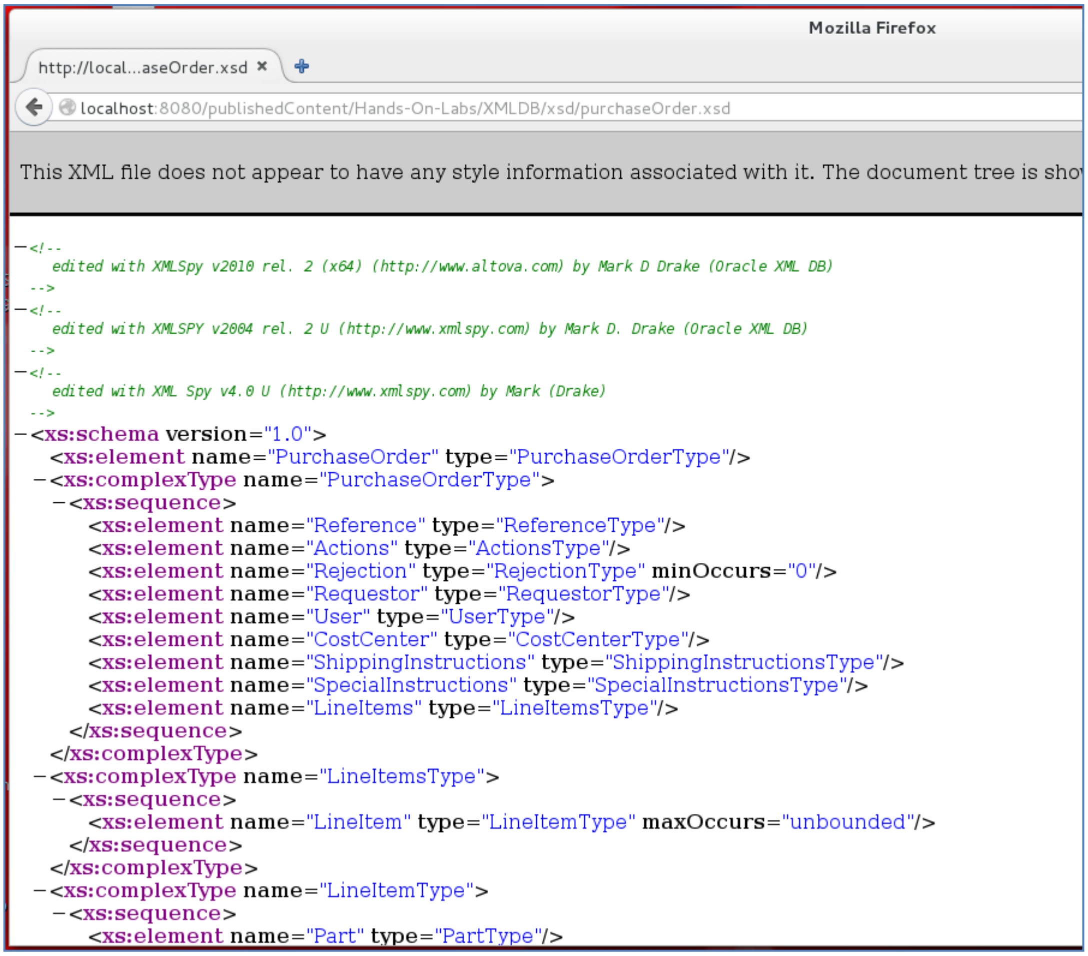
This shows the default browser view of the root folder of the oracle XML DB repository. Click the following links “publishedContent” ➔ “Hands-On-Labs” ➔ “XMLDB” ➔ “xsd”. This will show the contents of the folder /publishedContent/Hands-On-Labs/XMLDB/xsd which should be as follows:


Index of /publishedContent/Hands-On-Labs/XMLDB/xsd/ - Mozilla Firefox
Index of /publishedCont...
localhost:8080/publishedContent/Hands-On-Labs/XMLDB/xsd/
Search

Index of /publishedContent/Hands-On-Labs/XMLDB/xsd/

Name	Last modified	Size
purchaseOrder.xsd	Wed, 24 Jun 2015 22:20:31 GMT	9117

3. Click on the PurchaseOrder.xsd document to display the PurchaseOrder XML Schema.


Mozilla Firefox
http://local...aseOrder.xsd
localhost:8080/publishedContent/Hands-On-Labs/XMLDB/xsd/purchaseOrder.xsd

This XML file does not appear to have any style information associated with it. The document tree is sho

```
-<!--
   edited with XMLSpy v2010 rel. 2 (x64) (http://www.altova.com) by Mark D Drake (Oracle XML DB)
 -->
-<!--
   edited with XMLSPY v2004 rel. 2 U (http://www.xmlspy.com) by Mark D. Drake (Oracle XML DB)
 -->
-<!--
   edited with XML Spy v4.0 U (http://www.xmlspy.com) by Mark (Drake)
 -->
-<xs:schema version="1.0">
   <xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
 -<xs:complexType name="PurchaseOrderType">
   -<xs:sequence>
      <xs:element name="Reference" type="ReferenceType"/>
      <xs:element name="Actions" type="ActionsType"/>
      <xs:element name="Rejection" type="RejectionType" minOccurs="0"/>
      <xs:element name="Requestor" type="RequestorType"/>
      <xs:element name="User" type="UserType"/>
      <xs:element name="CostCenter" type="CostCenterType"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"/>
      <xs:element name="LineItems" type="LineItemsType"/>
    </xs:sequence>
   </xs:complexType>
 -<xs:complexType name="LineItemsType">
   -<xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
 -<xs:complexType name="LineItemType">
   -<xs:sequence>
      <xs:element name="Part" type="PartType"/>
```

You can see that the XML Schema is an XML document, compliant with an XML Schema called the “Schema for Schemas”. This Schema defines the structure of the PurchaseOrder documents.

XML Schema registration and Object-Relational storage

Before an XML Schema can be used with Oracle XML DB it must be registered with the database. XML Schemas can be registered for use with either Binary XML storage or Object-Relational XML Storage. The decision on whether to use Binary XML or Object-Relational storage will be driven by a number of factors, including :

- The complexity of the object-model defined by the XML Schema
- The access patterns for instance documents
- The kinds of update operations that will take place
- The nature and frequency of changes to the XML Schema

In general, if the XML Schema describes a hierarchical object model with limited amounts of recursion and variability, then the XML Schema is probably a candidate for object-relational storage. On the other hand, if the XML schema defines a complex, recursive structure with large degrees of variability and significant numbers of 'any' elements then Binary XML storage is probably a better option.

Object-Relational storage provides highly optimized leaf-level access and update, while Binary XML storage is more efficient for document level operations. Consequently in ETL scenarios, where the objective is to get SQL based access to XML content or populate existing relational tables with values extracted from the XML, Object-Relational storage is probably more effective. Content-centric scenarios, where document level operations are more common, are probably better suited for Binary XML, since it avoids the overhead associated with a complete decomposition and reconstruction of the XML.

Another factor to consider is XML Schema evolution. If the XML Schema changes in ways that do not invalidate the existing corpus of instance documents then object-relational storage's in-place schema evolution will probably be able to manage the schema changes, On the other hand, if the XML schema changes in ways that invalidate the existing corpus of documents on a regular basis, then schema-based Binary XML or even non-schema based Binary XML is probably more appropriate.

Object-Relational storage works by analyzing the object model defined by the XML Schema and deriving an equivalent SQL object model. It then creates the set of tables that allow the SQL object model to be persisted in the database. Collections in the XML model are mapped to nested tables in the SQL model.

A complete, lossless bi-directional mapping between the XML and SQL models ensures that XML documents are stored in the database with no loss of fidelity. The XML abstraction, based on XMLType, allows developers to manipulate XML using XQuery. XQuery operations on Object-Relational XMLType, are compiled into the same query algebra as SQL, allowing the Oracle Database to optimize XQuery in the same way that it optimizes SQL.

The next section shows how to register an XML Schema for use with Object-Relational storage and how to create indexes on the underlying storage model.

1. Using the File tab, right click on the file **3.0 registerSchema.sql** and select **open.**
2. Click the RunScript icon ▶ to execute the script.

   First the XML schema is loaded into memory from the XML DB repository using the XDBURITYPE operator.

   Next the package DBMS_XMLSCHEMA_ANNOTATE is used to annotate the XML Schema. XML Schema annotation provides a mechanism that allows a DBA some control over the object model and tables created by schema registration. This example disables DOM Fidelity, sets the name of the table that will be used to store instance documents in the database, and defines explicit names for some of the SQL objects that will be created by the schema registration process.

   Next package DBMS_XMLSCHEMA is used to register the XML Schema with the database. The SchemaURL is nothing more than a unique identifier for the XML Schema. LOCAL is set TRUE, meaning that the XML Schema is registered as a local XML Schema. A local XML Schema can only be used from the database schema that was used to register the XML Schema. Registering an XML Schema as a global schema allows the XML Schema to be shared across multiple database schemas. GENTypes is set TRUE, forcing the creation of the SQL objects that will be used to manage the XML. A SQL object will be created for each complexType defined by the XML Schema. GENTables is also set TRUE, forcing the creation the tables required to persist the SQL objects in the database.

   In this case the XMLType table PURCHASEORDER is created. PURCHASEORDER uses Object-Relational storage. Since the PurchaseOrder schema defines two repeating elements, two SQL collection types are generated, and two nested tables are created to manage the collections. The nested tables are given system generated names. These names can be difficult to work with, so DBMS_XMLSTORAGE_MANAGE is used to supply user friendly names.

Bulk Loading XML files using SQL Loader.

Oracle’s SQL Loader utility provides a convenient way of loading large volumes of XML documents into the database. The following example shows a SQL Loader control file that will load a set of documents into an XMLType table. SQL Loader can be used with Binary and Object-Relational storage models, and works with both XMLType tables and XMLType columns.

```
load data
infile '2015.dat'
append
into table PURCHASEORDER
xmltype(XMLDATA) (
 filename filler char(999),
 XMLDATA  lobfile(filename) terminated by eof)
```

The files will be loaded into the PURCHASEORDER table. The names of the files to be loaded are contained in the file 2015.dat. Each file name is a maximum of 999 characters long.

The file 2015.dat contains 10,000 entries in the following format.

```
sampleData/2015/Apr/ABANDA-20150407201438314PDT.xml
sampleData/2015/Apr/ABANDA-20150417140257543PDT.xml
sampleData/2015/Apr/ABANDA-20150427125050959PDT.xml
…
```

The next section shows how to use SQL Loader to load the sample documents into the database

1. Close SQL Developer.
2. Open a command prompt by clicking on the terminal icon found on the desktop .
3. Execute the command **cd Database_Track/XMLDB/sqlldr**
4. Execute the command **sqlldr -userid=OE/oracle -control=PURCHASEORDER.ctl rows=1000**

   SQL Loader will direct path load 10,000 documents into the PURCHASEORDER table.
5. Once sqlldr has completed, type **CTRL+D** to close the terminal window

Using XQuery with Object Relational storage

One of the key features of Oracle XML DB is that applications are developed independently of the way in which the XML is stored. This means that the choice of storage model does not affect the way in which applications are written. In the previous section the Binary XML version of the PURCHASEORDER table was replaced with one based on Object-Relational storage. This section will demonstrate that no changes are required to execute the XQuery expressions used with the Binary XML table on the Object-Relational table.

1. Using the File tab, right click on the file **1.1 simpleQueries.sql** and select **open**
2. Click the RunScript icon to execute the script.

   All of the queries execute successfully. No changes were required to the XQuery expressions. As expected, the results of running the queries on Object-Relational storage are identical to the results of running the queries on Binary XML storage.

3. Using the File tab, right click on the file **1.3 indexedQueries.sql** and select **open**
4. Use autotrace to generate the execution plan for each of the queries.

   The autotrace output shows that full table scans are being used to execute the queries. This is expected as currently there are no indexes that can be used to optimize the queries.

Indexing Object-Relational storage to optimize XQuery operations.

XQuery operations on Object-Relational storage are optimized by creating conventional B-Tree indexes on the tables that are used to persist the SQL objects. There are two ways of determining which tables and columns should be indexed.

- The first involves generating the execution plan for the query and then creating indexes based on the information in the plan. In this case the index will be created using the system generated table and column names shown in the execution plan.

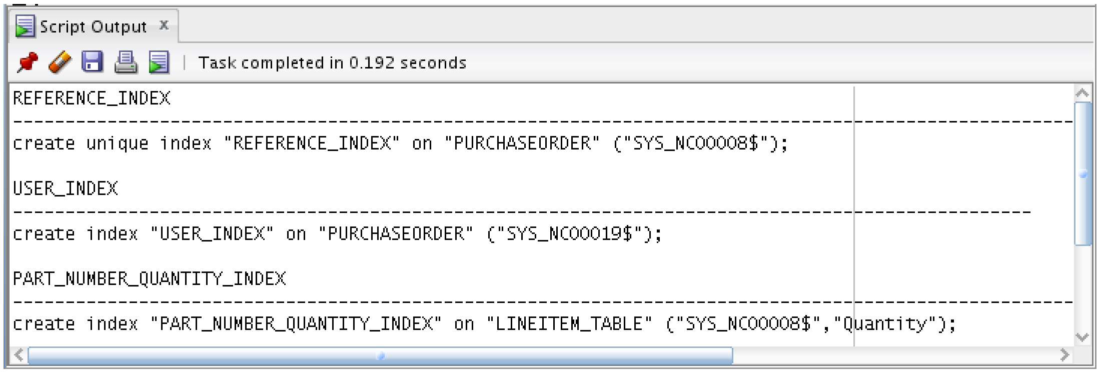
- The second involves using the method XPATH2TABCOLMAPPING in the package DBMS_XMLSTORAGE_MANAGE to determine which table and column corresponds to a particular XPath expression, and then using this information to generate a DDL statement that creates the appropriate index.

In this example, the second approach will be used to create indexes to optimize queries on the Reference and User nodes and a compound index on the repeating elements Part Number and Quantity.

1. Using the File tab, right click on the file **3.2 createIndex.sql** and select **open**

2. Click the RunScript icon to execute the script.

   The script uses package DBMS_XMLSTORAGE_MANAGE to determine which tables and columns need to be indexed. The output from running these queries is aa set of DDL statements that will create the required indexes.



```
REFERENCE_INDEX
------------------------------------------------------------------------------------------------------------
create unique index "REFERENCE_INDEX" on "PURCHASEORDER" ("SYS_NC00008$");

USER_INDEX
--------------------------------------------------------------------------------------------------------
create index "USER_INDEX" on "PURCHASEORDER" ("SYS_NC00019$");

PART_NUMBER_QUANTITY_INDEX
------------------------------------------------------------------------------------------------------------
create index "PART_NUMBER_QUANTITY_INDEX" on "LINEITEM_TABLE" ("SYS_NC00008$","Quantity");
```

3. Click the Run script output as script icon .

   This will open a new SQL worksheet which will contain the create index statements.

4. Position the cursor at the start of the first create index statement and click on the run statement icon .

   This will create the index needed to optimize queries on the Reference element.

3. Repeat step 4 for each of the remaining “create index” statements as well as the call to DBMS_STATS.GATHER_SCHEMA_STATS

4. Return to the SQL Worksheet containing the file **1.3 indexedQueries.sql**.

5. Use autotrace to generate the execution plan for each of the queries.

   The autotrace output shows that the indexes REFERENCE_INDEX, USER_INDEX and PART_NUMBER_QUANTITY_INDEX indexes are used to optimize the XQuery operations.

Equi-Partitioning of Object-Relational Storage

Partitioning helps simplify XML data life-cycle management and performance. Equi-partitioning for object-relational storage was introduced in Oracle Database 11g Release 2. Equi-partitioning ensures that all of the nested tables used to manage collections automatically follow the partitioning strategy defined for the parent XMLType table. This allows partitioning pruning and partition maintenance operations to operate on XMLType tables in the same way as they operate on simple relational tables.

In this section, you will learn how to create a partitioned object-relational XMLType table.

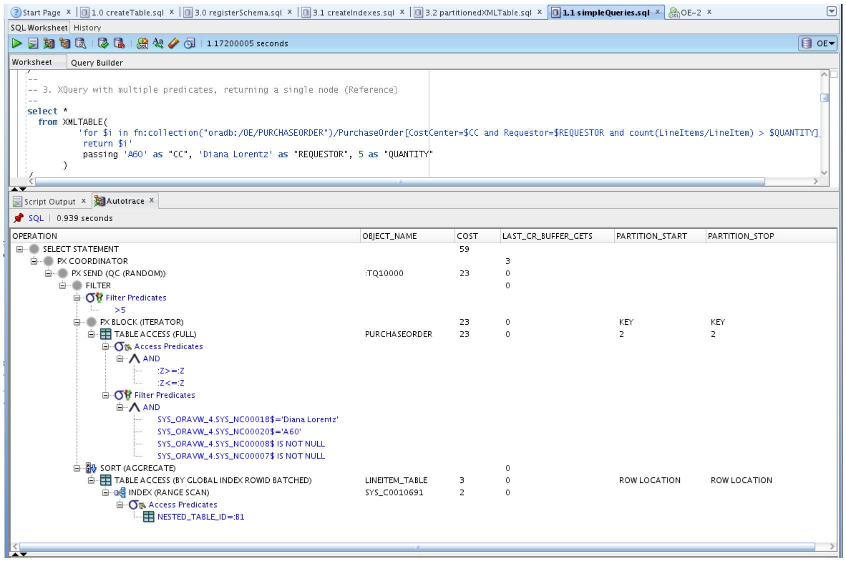
1. Using the File tab, right click on the file **3.4 partitionedXMLTable.sql** and select **open**
2. Click the RunScript icon to execute the script.

   The script uses a Create Table as Select (CTAS) operation to make a copy of the content of the existing PURCHASEORDER table. It then drops the PURCHASEORDER table and re-creates it as a partitioned table. The table is partitioned based on the value of the User element. Note that the partition key has to be specified using Object-Relational syntax. The script then reloads the documents into the PURCHASEORDER table and re-creates the indexes.

   REFERENCE_INDEX is created as a global index, while indexes USER_INDEX and PART_NUMBER_QUANTIY_INDEX are created as local indexes.

3. Return to the SQL Worksheet containing the file **1.1 simpleQueries.sql**.
4. Use autotrace to generate the execution plan for each of the queries.

   The autotrace output shows that table is now partitioned, parallel execution occurs where appropriate and partition pruning takes place for XQuery operations that include a predicate on the partition key.



```
/
--
-- 3. XQuery with multiple predicates, returning a single node (Reference)
--
select *
  from XMLTABLE(
          'for $i in fn:collection("oradb:/OE/PURCHASEORDER")/PurchaseOrder[CostCenter=$CC and Requestor=$REQUESTOR and count(LineItems/LineItem) > $QUANTITY]
           return $i'
           passing 'A60' as "CC", 'Diana Lorentz' as "REQUESTOR", 5 as "QUANTITY"
       )
/
```

```
OPERATION                                          OBJECT_NAME     COST  LAST_CR_BUFFER_GETS  PARTITION_START  PARTITION_STOP
SELECT STATEMENT                                                   59
  PX COORDINATOR                                                         3
    PX SEND (QC (RANDOM))                          :TQ10000        23    0
      FILTER                                                             0
        Filter Predicates
            >5
        PX BLOCK (ITERATOR)                                        23    0                    KEY              KEY
          TABLE ACCESS (FULL)                      PURCHASEORDER   23    0                    2                2
            Access Predicates
              AND
                :Z>=:Z
                :Z<=:Z
            Filter Predicates
              AND
                SYS_ORAVW_4.SYS_NC00018$='Diana Lorentz'
                SYS_ORAVW_4.SYS_NC00020$='A60'
                SYS_ORAVW_4.SYS_NC00008$ IS NOT NULL
                SYS_ORAVW_4.SYS_NC00007$ IS NOT NULL
        SORT (AGGREGATE)                                                 0
          TABLE ACCESS (BY GLOBAL INDEX ROWID BATCHED)  LINEITEM_TABLE  3    0                    ROW LOCATION     ROW LOCATION
            INDEX (RANGE SCAN)                     SYS_C0010691    2     0
              Access Predicates
                NESTED_TABLE_ID=:B1
```

Relational access to XML content

Oracle's implementation of the SQL language provides many powerful features that are not yet supported by the XQuery standard. Also there are many tools (and developers) that are not yet able to work effectively with XML content and XQuery. One of the key benefits of XML DB is that it provides relational tools and programmers with the ability to use SQL to work directly with XML content, enabling all the features of SQL to be used with XML and preserving existing investments in relational technology and infrastructure.

Creating relational views of XML content

This next section will show how the combination of XQuery and XMLTable enables efficient SQL operations on XML content.

1. Using the File tab, right click on the file **4.0 relationalViews.sql** and select **open**
2. Click the RunScript icon to execute the script.

   This script uses XQuery and XMLTable to create two views that provide relational access to XML content. These views provide a typical MASTER-DETAIL relationship that can be used to access the contents of the XML using SQL. The views use XQuery to map scalar values from the XML into the columns defined by the view.

   The first view provides access to values from elements and attributes that occur at most once per document. It can be considered the master table.

   The second view provides access to values from the collection of LineItem elements that can occur multiple times per document. It can be considered the detail. This view is created using nested XMLTable operators.

   The first XMLTable operator projects out the values that form the 'primary key' for each document and the set of repeating elements. The set of repeating elements forms the input to the second XMLTable operator, which generates one row for each of the repeating elements.

   Since the second XMLTable processes the output of the first XMLTable, a correlated join takes place. This ensures that the rows produced by the secondary XMLTable are joined with the corresponding rows from the primary XMLTable.

Examine the views in SQL Developer

This section shows that the views created in the previous step can be used with any tool that supports accessing relational data via SQL views.

1. Select the Connections tab and click the **+** icon to the left of the oe connection to expand it.

2. Expand Views, and select PURCHASEORDER_MASTER_VIEW.

3. On the right window pane, select the Columns tab.

   Note that SQL Developer sees the view as a standard relational view.

4. Click the Data tab.

   Note that SQL Developer is able to retrieve the data, even though the content is stored in the database as an XMLType table.

5. Repeat steps 2, 3 and 4 for the PURCHASEORDER_DETAIL_VIEW.

This demonstrates that any ID, reporting tool or Business Intelligence tool that understands the relational paradigm and SQL can use this technique to access content stored in XMLType tables within Oracle Database.

Performing SQL operations on XML content

This section shows how relational views of XML enable simple and advanced SQL statements to operate on XML content stored in the Oracle database.

1. Using the File tab, right click on the file **4.1 relationalQueries.sql** and select **open**
2. Click the RunScript icon to execute the script.

   None of the queries contain a reference to XML constructs. Advanced SQL functionality like group by, group by (rollup()) and lag can be used to analyze XML content. Predicates can be supplied as part of the SQL queries.

3. Use autotrace to generate the execution plan for each of the queries

   The execution plan shows that all of the queries are executed as SQL queries over the tables that manage the XML content. The XML abstraction has been completely bypassed in these cases.

XML access to Relational Content

The previous section showed how to access XML content from SQL. Oracle XML DB also allows XQuery operate directly on relational data. Two distinct metaphors are can be used for this, one is XML-centric, and the other is SQL-centric. Both techniques are based on the concept of creating an XML representation of the data in relational tables. There is no difference in terms of efficiency, the decision on which approach to adopt is a matter of personal preference.

The first technique uses a canonical mapping to generate an XML representation of each row in a relational table. Non canonical XML representations of the relational data can then be created by using XQuery to transform from the canonical representation into the required format.

The second approach involves a SQL–centric approach, based on the SQL/XML publishing functions. The SQL/XML publishing functions allow a SQL statement to return one or more XML documents rather than a traditional tabular result set.

Both techniques can be used to create XML views that allow XQuery based operations to be performed on relational data.

Using XQuery to generate XML from relational tables

The first example shows how to use XQuery and fn:collection to generate XML documents from relational tables.

1. Using the File tab, right click on the file **5.0 relational2XML_XQUERY.sql** and select **open**
2. Click the RunScript icon to execute the script.

   The first query shows the output of the simple canonical mapping used to generate XML documents from rows in a relational table. Each document has a root node called ROW. The root node contains one element for each column in the table. The element names will be generated from the column names; character escaping will be used when the column name contains characters that are not permitted as an XML name. XQuery pragmas can be used to determine how null values are handled.

   The second query shows how XQuery can be used to transform the canonical mapping into a more useful XML format. This example also shows how this technique can be used to generate documents containing content that comes from multiple relational tables.

3. Use autotrace to generate the execution plan for the second query.

   The autotrace output shows that a purely relational execution plan is used to generate the XML, despite the fact that all of the joins operations needed to construct the XML were specified as part of the XQuery expression.

Using SQLXML to generate XML from relational tables

The second example shows how to use SQL/XML functions to generate XML documents from relational tables.

1. Using the File tab, right click on the file **5.1 relational2XML_SQLXML.sql** and select **open**
2. Click the RunScript icon to execute the script.

   This query shows the SQL/XML equivalent of the second XQuery in the previous example. SQL/XML uses a template based approach to generating XML from relational data. The nesting of the SQL/XML operators describes the shape of the document and how the column values are mapped into text nodes and attributes.

3. Use autotrace to generate the execution plan for the second query.

   The autotrace output shows that a purely relational execution plan is used to generate the XML. As expected the execution plan for the SQL/XML approach is just as efficient as the execution plan for the XQuery based option.

Create a persistent XML View of relational data.

The example shows how to create an XML view of relational data. It uses the same SQL/XML query as the previous example, however the XQuery expression from the first example in this section could also be used for this purpose. The view is a view of XMLType, which means that the view definition must include a mechanism for generating a unique ID for each row in the view. In this example the ID is supplied by the expression in the with object id clause of the create view statement. In this case, uniqueness is based on the contents of the Name element.

1. Using the File tab, right click on the file **5.2 xmlViewRelationalData.sql** and select **open**
2. Click the RunScript icon to execute the script.

   An XMLType view called DEPARTMENT_XML is created.

3. Using the File tab, right click on the file **5.3 xqueryOnRelationalData.sql** and select **open**
4. Click the RunScript icon to execute the script.

   The XMLType view allows XQuery operations to be performed against relational data. In the same way that relational views of XML enable SQL-centric development on top of XML content, XML views of relational data enable XML-centric developers to work with relational data.
5. Use autotrace to generate the execution plan for the second query.

   The autotrace output shows that the execution plan obtained when executing an XQuery on a

XMLType view of relational data is a purely relational plan, demonstrating that XML DB is able to optimize XQuery operations on relational data in the same way that it can optimize relational operations on XML content.

This concludes the Oracle XML DB Hands on Lab.