Querying XML

An Oracle White Paper
December 2004

ORACLE

Querying XML

INTRODUCTION

As more and more data is stored and/or represented as XML, XQuery[1] is under the spotlight as a standard way to query XML. XQuery is a W3C Working Draft, expected to reach Recommendation status some time in 2005. XQuery extends the functionality of XPath, a language for expressing paths into XML structures, with predicates. XPath 1.0[2] has been a W3C Recommendation since November 1999. The next version of XPath – XPath 2.0 – is being defined as part of the XQuery 1.0 effort.

The most recent version of the popular SQL standard, SQL:2003, introduced a built-in XML data type plus a set of SQL/XML publishing functions. Oracle9*i* further extended SQL/XML with SQL functions such as *extract* , *extractValue,* and *existsNode*, which query XML using XPath 1.0. The latest SQL/XML proposal – expected to be part of SQL:2005 – introduces two new functions (*XMLQuery* and *XMLTable*) that support XQuery 1.0 within SQL.

As a strongly typed, highly expressive, compositional query language, XQuery is of interest to developers seeking to access data stored in multiple applications and data sources. Even where the storage is more homogeneous – say, entirely in a database – if the data is XML-oriented, XQuery preserves the XML abstraction, whereas SQL/XML also allows relational access.

Many see XQuery and SQL/XML as ***competing*** technologies for querying XML, but they are actually ***complementary***.

This paper provides an overview of XPath and XQuery support in Oracle database 10*g* Release 2, discusses where XQuery is most useful, and describes how the SQL/XML and Oracle extensions to SQL provide a complete solution for querying XML in the database. Mid-tier querying of XML – *i.e.*, using a query engine outside of any database to query XML stored in heterogeneous data sources – is

[1] See http://www.w3.org/TR/xquery/ for the XQuery Language, and http://www.w3.org/XML/Query for the full suite of XQuery specs (and other useful information on the XQuery effort).
[2] See http://w3.org/TR/xpath

similarly out of scope for this paper. If you are interested in mid-tier querying of XML, see the technology preview of Oracle XML Data Synthesis (XDS)[3].

Much of the material for this paper is taken from a paper of the same title presented at XML 2004 [4].

Readers who are already familiar with SQL, SQL/XML, and XQuery, and who want to learn about what's new in Oracle database 10*g* Release 2, should skip to "XQuery in SQL/XML".

QUERYING XML

Data is "factual information ... used as a basis for reasoning, discussion, or calculation"[5]. In this paper we focus on *persistent* data – data that needs to be stored in a reliable way so that it can be retrieved and manipulated. *Non*-persistent data is data that is produced, used, and discarded, such as the data packets exchanged in SOAP messages. While non-persistent data is important, especially in any discussion of XML, we will leave that discussion for another paper.

Persistent data has two important attributes - *storage* and *representation*. We describe storage and representation with reference to some sample data.

Data Sample 1 - House Resolution 558

Our first data sample is taken from a resolution presented to the U.S. House of Representatives[6]. The resolution is entitled "Welcoming the accession of Bulgaria, Estonia, Latvia, Lithuania, Romania, Slovakia, and Slovenia to the North Atlantic Treaty Organization (NATO)". It was created 11th March 2004, sponsored by Bereuter, and it "welcomes with enthusiasm the accession of Bulgaria, Estonia, Latvia, Lithuania, Romania, Slovakia, and Slovenia to the North Atlantic Treaty Organization (NATO)".

Data Representation

XML Representation

The previous paragraph is a natural language representation of (some of) the data in House Resolution 558. The first thing to notice is that the representation is *not* the data, it is merely an abstract view of (some of) the data. Similarly, the data is not the real-world object - no matter how well my data described House Resolution 558, the data would not *be* the resolution. This separation of representation, data and real-world object may seem obvious, but bear them in mind as we step through some variations in representation and storage.

---

[3] See the Oracle Technology Network, http://www.oracle.com/technology/tech/xml/xds/index.html
[4] See http://www.xmlconference.org/xmlusa/
[5] From the Merriam-Webster Online Dictionary (http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=data)
[6] See http://thomas.loc.gov, especially http://thomas.loc.gov/home/gpoxmlc108/

Example 1: Sample resolution - XML shows another possible representation for House Resolution 558, an XML representation.

Example 1: Sample resolution - XML

```
<resolution dms-id="42" public-private="public">
  <congress>108</congress>
  <session>2</session>
  <legis-num>558</legis-num>
  <action>
    <action-date>20040311</action-date>
    <action-desc>
      <sponsor>Bereuter</sponsor>
      <cosponsor>Wexler</cosponsor>
      <cosponsor>Gillmor</cosponsor>
      <cosponsor>Shimkus</cosponsor>
      <committee-name>Committee on International Relations
      </committee-name>
    </action-desc>
  </action>
  <official-title>Welcoming the accession of Bulgaria, Estonia, Latvia,
Lithuania, Romania, Slovakia, and Slovenia to the North Atlantic Treaty
Organization (NATO), and for other purposes.
  </official-title>
  <resolution-body>
    <paragraph>welcomes with enthusiasm the accession of Bulgaria,
Estonia, Latvia, Lithuania, Romania, Slovakia, and Slovenia to the
North Atlantic Treaty Organization (NATO);</paragraph>
    <paragraph>reaffirms that the process of NATO enlargement enhances
the security of the United States and the entire North Atlantic
area;</paragraph>
    <paragraph>agrees that the process of NATO enlargement should be
open to potential membership by any interested European democracy that
meets the criteria for NATO membership as set forth in the 1995 Study
on NATO Enlargement and whose admission would further the principles of
the Washington Treaty of 1949 and would enhance security in the North
Atlantic area; and</paragraph>
    <paragraph>recommends that NATO heads of state and government
should review the enlargement process, including the applications of
Albania, Croatia, and Macedonia, at a summit meeting to be held no
later than 2007.</paragraph>
  </resolution-body>
</resolution>
```

What does the XML representation give us that the informal, natural language description does not? XML gives us:

- Distinct, named data fields (elements)
  - benefit: makes it easier to express and query the data concisely and precisely
- Hierarchical, named structure (sub-elements)
  - benefit: when used to logically group elements, makes it easier to express and query the data. Think of an address element, with sub-elements street, city, state.
- Document order (except attributes)
  - benefit: breaking the information down into logical chunks (elements) is useful, but in some cases - and especially with passages of text - those logical chunks are only meaningful when you know their order.

- flexibility
    - many people quote "flexibility" as one of their chief reasons for preferring an XML representation of data. We would argue that flexibility can be A Bad Thing. After all, the informal natural language representation of data is the most flexible one – and that's exactly why we are looking at alternatives.

SQL Representation

It is possible to represent this same data in many other useful, structured ways - as SGML, as HTML, as a comma-separated list of fields suitable for input to, say, a spreadsheet, or as a stream of codes suitable for punching on cards. For this paper we consider just one other representation, an SQL representation. This is a representation of the data as you might expect to see it produced from an Object-Relational database.

One possible SQL representation of House Resolution 558 is shown in Example 2: Resolutions - SQL.

Example 2: Resolutions - SQL

Table 1: resolutions

id	congress	session num	legis num	action date	committee name	official title
42	108	2	558	20040311	Committee on International Relations	Welcoming the ....
etc.						

Table 2: resolutions_sponsors

resolution id	sponsor	sponsor type
42	Bereuter	primary
42	Wexler	co-sponsor
42	Gillmor	co-sponsor
42	Shimkus	co-sponsor
etc.		

Table 3: resolutions_paragraphs

resolution id	p order	paragraph
42	1	welcomes with enthusiasm the accession of...
42	2	reaffirms that the process of NATO enlargement ...
42	3	agrees that the process of NATO enlargement ...
42	4	recommends that NATO heads of state and government ...
etc.		

This is a fairly naïve representation. We have not tried to either fully normalize the data for efficient disk storage, nor denormalize for performance. Perhaps more importantly, this representation does not take advantage of any of the relatively recent advances in Object-Relational modelling of hierarchical structures, such as variable-length arrays (for modelling sequences of data elements) or nested tables (for modelling unordered sets of data elements). It is sufficient to say that the same data we saw represented in natural language and in XML can also be represented in SQL (that is, Relationally or Object-Relationally).

What does the SQL representation give us that the informal, natural language description does not? SQL gives us:

- Distinct, named data fields (elements)
  - an element (or cell) is represented by a row/column intersection, named by a schema.table.column-name
  - benefit: makes it easier to express and query the data concisely and precisely.
- Relations between data fields
  - defined by the table structure and referential integrity.
  - benefit: logically groups elements, allows arbitrarily complex queries over realted data.
- A set-based algebra over the data + relations
  - benefit: allows complex queries with well-defined, standard characteristics.
- Constraints
  - with SQL, it's easy to code business rules and data integrity rules into the data - something XML does not (yet) handle well.

There are two things that were on the XML list that are missing from the SQL list. First, data represented in SQL does not have a default ordering (except in a few types such as varrays). SQL was designed to handle structured data, where order is rarely important. As you can see from the resolutions_paragraphs table, it is easy to introduce an ordering when it is necessary. Second, people tend to think the SQL representation is less flexible than XML. We would argue that this kind of flexibility is A Bad Thing - it allows anyone to add data in any form, and reduces the value of the data.

Reasons for choosing some representation

You might choose an SQL presentation for your data so that you can manage and query it with tools that are robust, mature, and readily available. You might already own a suite of tools that work well with an SQL representation, and you probably already have the skills in-house to work with SQL. Many applications need to publish data (make data available) to a 3rd-party tool or application. SQL has been

around for several decades, and there is a wealth of available tools, applications and skills.

On the other hand, you might choose an XML representation because you need to publish data to (or consume data from) a web service or a browser or an integration application.

See Table 5: Choosing Some Storage, Representation

Data Storage

Data storage is *related to* data representation, but it's not the same thing. Before describing how you might store your data, we must point out as an aside that, like representation, data storage is an abstraction - we rarely talk about where the bits actually sit on the disk (or how the bits are represented in the magnetic layer). For the purposes of this paper, files and tables is as concrete as we'll get.

SQL Storage

When people think of SQL (Object-Relational) storage, they generally think of tables, with rows and columns. The table *is* the basic unit of SQL storage, but there are a number of ways to model data that is not the same "shape" as a table.

- master-detail tables - create a table that stores "details" of the master table. In our example, the "resolutions_paragraphs" table stores detail records about the paragraphs in the resolution. This models the case where you want more than one detail (paragraph) in a single cell.
- variable-length arrays (varrays) - we could have achieved the same effect by adding a column to the "resolutions" table with a type defined as a variable-length array, or varray. A varray allows you to define an ordered array of elements, and store an instance of that array in a single cell in a table.
- nested tables - we could also store the paragraphs of resolutions in a nested table. Nested tables, as the name implies, allow you to define a table of elements and store that table "nested" in a single column of another table. Nested tables differ from varrays in that nested tables are naturally unordered.
- objects - the object types that appeared in SQL:1999 allow you to model arbitrarily complex objects - datatypes with any shape, plus associated methods - in an object-relational database.

In the examples in the rest of this paper, where we refer to "SQL storage" we mean the storage implied by the naïve representation in Example 2: Resolutions - SQL. In a real-world application you have many more SQL storage options to choose from.

XML Storage

In Oracle database 10*g* there are at least four ways to store an XML document - in a LOB, shredded, as a native XMLType (which follows the SQL:2003 XML data type), or as a file in the Oracle XML Database Repository.

- in a LOB (Large OBject) – you can store an XML document in a single cell in a table. The obvious LOB type is a CLOB (Character Large OBject), though you might also choose a VARCHAR2 (character string). When you store XML in a LOB, SQL queries are not aware of the structure of the XML data. (An exception to this rule is Full-Text queries using the Oracle Text "contains" function).

- shredded – you can break down the XML structure into SQL structures and store the data in tables. For example, you could pre-process Example 1: Sample resolution - XML and store it in the tables shown in Example 2: Resolutions - SQL. This is known as "shredding". Shredding can be done manually, or automatically using packages such as DBMS_XMLSAVE.

- natively, as XMLType – Oracle, beginning at Oracle9*i*, supports native storage of XML as data type XMLType (which follows the SQL:2003 XML type). XMLType storage may be schema-typed or untyped - that is, the XMLType may be associated with an XML Schema that gives type information about its structure and contents. XMLType storage offers excellent query performance while preserving DOM fidelity. You can create a table with one or more columns of type XMLType, or you can create an XMLType table.
  In an Oracle database, you can think of the XMLType as *storage* – in the sense that you can create a table with one or more columns of type XMLType, for example – or you can think of it as *representation* – it is an abstraction over several *lower-level storage* strategies, including CLOB and shredded storage.
  If you have XML data coming into the database we recommend that you store it as XMLType (that is, either an XMLType table or a table with a column of type XMLType) and choose the *lower-level storage* appropriately. The XMLType abstraction offers tremendous flexibility in data and performance management, while preserving your investment in applications.

- as a file in the Oracle XML Database Repository – the Oracle XML Database Repository gives you a file/folder paradigm for storing XML (and non-XML) files. With the Oracle XML Database Repository you get all the benefits of an unbreakable database plus the convenience of a file system. You can access the Oracle XML Database Repository via http, ftp, or SMB (allowing drag-and-drop from your desktop to a database folder). See Example 17: Simple XQuery - vote count (and others) for an example of XQuery against a file in the Oracle XML Database Repository.

Remember we are talking about choosing a *storage* mechanism – you will see later (Table 5: Choosing Some Storage, Representation) that you can choose some

storage with a very different representation. One example is to store data relationally, and represent it as XML (via an XML View, as in Example 5: resolutions_xml_view).

Reasons for choosing some storage

You might choose to store data as SQL (in Object-Relational tables) for greater efficiency – efficiency of space (no need to store all those tags), and efficiency of time (SQL indexing techniques are mature and robust, so it may be faster to query data stored as SQL).

On the other hand, you might choose to store your data as XML if your application requires that the stored data exactly matches the incoming representation (including order, whitespace, etc.) for legal reasons. Or you might have XML data that rarely changes, but has to be input and/or output frequently. In such cases it may be more efficient to keep the data all in one place as a single XML "thing".

See Table 5: Choosing Some Storage, Representation.

SQL XML Duality - Storage and Representation

In this section we show that storage and representation are indeed separate. You can store your data in SQL and produce a representation either in some other form of SQL or in XML. You can store your data in XML and produce a representation either in some other form of XML or in SQL.

SQL Storage, SQL Representation

We start with the simplest transformation, SQL storage to a (different) SQL representation. The classic way to achieve this is with views. Example 3: Simple SQL View creates a simple view that joins (pulls data from) some columns in two tables, renames the columns, and restricts the data to include only some rows.

Example 3: Simple SQL View

```
create view body_text as
  select
    r.congress AS congress ,
    r.session_num AS congress_session ,
    r.legis_num AS legislation ,
    p.paragraph AS beginning
  from
    resolutions r ,
    resolutions_paragraphs p
  where
    r.id = p.resolution_id and
      p.p_order = 1
```

The content of a view can be defined using any query. Once created, a view can be described and queried as if it were a table.

Example 4: Use a View Like a Table

```
SQL> describe body_text
 Name                           Null?    Type
 ------------------------------ -------- --------
 CONGRESS                                NUMBER
```

```
 CONGRESS_SESSION                          NUMBER
 LEGISLATION                               NUMBER
 BEGINNING                                 VARCHAR2(4000)

SQL> select beginning from body_text
/

BEGINNING
--------------------------------------------------------
welcomes with enthusiasm the accession of Bulgaria,
Estonia, Latvia, Lithuania, Romania, Slovakia, and
Slovenia to the North Atlantic Treaty Organization (NATO);

SQL> create view beginning as
  2    select beginning from body_text
  3  /

View created.

SQL> select * from beginning
/

BEGINNING
--------------------------------------------------------
welcomes with enthusiasm the accession of Bulgaria,
Estonia, Latvia, Lithuania, Romania, Slovakia, and
Slovenia to the North Atlantic Treaty Organization (NATO);
```

Example 4**:** Use a View Like a Table describes the view body_text, queries from the view, then creates and queries another view based on body_text. In all these operations, the view body_text (the representation of some SQL data) is indistinguishable from a table (the storage of SQL data).

SQL Storage, XML Representation

OK, now that we've got the general idea that data can be physically stored on the disk in one way, and represented (used, manipulated, queried) as if it were stored in quite a different way, let's apply that to SQL and XML. There are a number of ways to generate an XML representation from data stored in relational or object-relational tables. Example 5: resolutions_xml_view uses some of the SQL/XML publishing functions introduced in SQL:2003 Part 14 to produce the XML representation illustrated in Example 1: Sample resolution - XML from data stored in the tables described in Example 2: Resolutions - SQL.

Example 5: resolutions_xml_view

```
create or replace view resolutions_xml_view as
(select
  r.id AS id ,
  XMLElement ("resolution" ,
    XMLElement ("congress" , congress) ,
    XMLElement ("session" , session_num) ,
    XMLElement ("legis-num" , legis_num) ,
    XMLElement ("action" ,
      XMLElement ("action-date", to_char(action_date,
                  'YYYYMMDD')),
    XMLElement ("action-desc" ,
      (select (XMLElement("sponsor", sponsor))
          from resolutions_sponsors s
          where s.resolution_id = r.id
            and s.sponsor_type = 'primary'
      ) ,
      (select (XMLAGG(XMLElement("cosponsor", sponsor)))
          from resolutions_sponsors s
```

```
          where s.resolution_id = r.id
            and s.sponsor_type = 'cosponsor'
       ) ,
       XMLElement ("committee-name" , committee_name)
    )
  ) ,
  XMLElement ("official-title" , official_title) ,
  XMLElement ("resolution-body" ,
    (select (XMLAGG(XMLElement("paragraph", paragraph)))
          from resolutions_paragraphs p
          where p.resolution_id = r.id
    )
  )
) AS resolution
  from resolutions r
)
```

With this view in place, a query such as Example 6**: Query resolutions_xml_view** gives the same results as a query against a table with two columns, id (number) and resolution (CLOB, VARCHAR2 or XMLType).

Example 6: Query resolutions_xml_view

```
select id ,
  resolution
  from resolutions_xml_view
```

Results:

```
  ID RESOLUTION
---- ----------------------------------------
  42 <resolution>
       <congress>108</congress>
       <session>2</session>
       <legis-num>558</legis-num>
....
```

XML Storage, SQL Representation

We said in "XML Storage" that XML can be stored in at least 3 ways in an object-relational database – in a LOB (Large OBject), shredded into object-relational tables, or natively as XMLType. In this section we assume that you have chosen to store XML in an object-relational database as XMLType. In the Oracle database, XMLType is itself an abstraction which may represent several storage strategies, but we will leave that level of detail for another paper. Consider a table that has three columns, id (number), legis_num (number), and resolution (XMLType), as in Example 7**: Table resolutions_xml**.

Example 7: Table resolutions_xml

```
create table resolutions_xml (
  id number primary key ,
  legis_num number ,
  resolution XMLType
  )
```

We create an SQL representation of the XML in our example by using two functions - *extract* and *extractValue*. These functions take as arguments an XML document and an XPath, and return the result of evaluating the XPath against the XML document. *extract* returns a sequence of XML nodes, *extractValue* returns a single value. Note that you can apply these functions to any XML document stored

either in a table in the database, or in a file on the file-system (if you have the correct access rights), or in the Oracle XML Database Repository. Our example happens to use an XML document stored in a table in the database.

Example 8: resolutions_view

```
create or replace view resolutions_view as (
select
  id AS id ,
  extractValue( resolution, '/resolution/congress' )
      AS congress ,
  to_number( extractValue( resolution, '/resolution/session' ) )
    AS session_num ,
  extractValue( resolution, '/resolution/legis-num') AS legis_num ,
  to_date( extractValue( resolution,
    '/resolution/action/action-date'), 'YYYYMMDD' ) AS action_date,
  extractValue( resolution,
    '/resolution/action/action-desc/committee-name')
    AS committee_name ,
  extractValue( resolution, '/resolution/official-title')
    AS official_title
  from resolutions_xml
)
```

A query against the view resolutions_view (Example 8: resolutions_view) returns the same results as a query against the table resolutions - see Example 9**:** Query resolutions, resolutions_view for an example.

Example 9: **Query resolutions, resolutions_view**

```
select id ,
  action_date
  from resolutions

select id ,
  action_date
  from resolutions_view
```

So far we have avoided any discussion of data types. A full discussion of datatypes is outside the scope of this paper. For now, notice that *extractValue* in our examples returns a string (VARCHAR2(4000)) which you may coerce into some other SQL datatype using to_number, to_date, *etc.* In some cases it is possible to deduce the datatype to be returned – when the XML you are extracting from has an associated XML Schema, or is based on a view over SQL data. In such cases, *extractValue* will return data of the deduced datatype (mapped to an SQL data type).

XML Storage, XML Representation

The next cell in our Storage-Representation matrix has data stored as XML, with a representation that is XML in some other form. To achieve this we use the *extract* function, which takes as arguments an XML document and an XPath expression. *extract* returns the XML document (or fragment) obtained by evaluating the XPath expression over the XML document (or fragment). Again, we use an XML document stored in a table (Example 7**: Table resolutions_xml**), though we could have chosen a file in the file-system or an Oracle XML Database repository entry. And of course we could have applied a stylesheet using an XSLT engine, such as

the one built into the Oracle database. But for brevity, we stick with the SQL functions.

Example 10: View body_text_xml

```
create or replace view body_text_xml as (
  select
    id AS id ,
    XMLElement (
      "resolution" ,
      extract( resolution, '/resolution/congress') ,
      extract( resolution, '/resolution/session' ) ,
      extract( resolution, '/resolution/legis-num' ) ,
      extract( resolution,
        '/resolution/resolution-body/paragraph[1]' )
    ) AS resolution_brief
    from resolutions_xml
)
```

Example 10: View body_text_xml is a view that represents just a few of the elements of a resolution XML document, including the first paragraph. With this view in place, the query in Example 11**:** Query body_text_xml returns an id and a part of the XML document in Example 7**: Table resolutions_xml**. It queries an XML view over XML data, returning XML data.

Example 11: **Query body_text_xml**

```
select id, resolution_brief
  from body_text_xml
```

Results:

```
 ID RESOLUTION_BRIEF
---- ----------------------------------------
  42 <resolution>
       <congress>108</congress>
       <session>2</session>
       <legis-num>558</legis-num>
       <paragraph>welcomes with enthusiasm
       the accession of Bulgaria,
       Estonia, Latvia, Lithuania, Romania,
       Slovakia, and Slovenia to the
       North Atlantic Treaty Organization (NATO);
       </paragraph>
     </resolution>
```

Summary - SQL and XML Storage/Representation Duality

Table 4: SQL and XML Storage/Representation Duality

	REPRESENTATION			
STORAGE		**SQL**	**XML**	**Mixed**
	Relational	Relational	XML View	Relational + XML View
	XML	SQL View	XML Type	SQL View + XML Type
	Mixed	Relational + SQL View	XML View +XML Type	SQL View + XML View + Relational + XML Type

We have established that you can store data in tables and yet manipulate it (query, publish it) as if it were XML, and you can store data as XML and yet manipulate it (query, publish it) as if it were stored in tables. And of course you can mix'n'match both kinds of storage and representation.

There are a number of reasons for storing and representing data in either tables or XML — some are represented in Table 5: Choosing Some Storage, Representation. For many, the right answer will be Oracle's XMLType datatype which allows you to have the best of both worlds - it is an abstraction with several storage choices. A customer recently asked me "what's the best way to take all my relational data and convert it into XML?". Of course I asked, "Why ? What are you trying to achieve?" And the answer was, "We have a corporate directive that says we must be all-XML within 12 months". I recommended a good deal of thought and discussion about their storage and representation needs, rather than making a rash decision to physically convert data from one storage to another. In the short term, you can create one or more XML Views across all your relational data, and voilà - you can start using your data as if it were XML almost immediately. This approach is quick, easy and flexible - if you decide you want the XML to look different, you need only recreate the XML View.

Table 5: Choosing Some Storage, Representation

	STORAGE	REPRESENTATION
SQL	• efficient disk usage - no need to store bulky tags • efficient query - use mature indexing techniques (B-Tree, bitmap, ...) • fine-grained security • data integrity constraints • manage and query data using existing tools and skills	• manage and query data using existing tools and skills • publish data (make data available) to a 3rd-party tool or program
XML	stored data needs to exactly match incoming representation, e.g. for legal reasons • if the incoming and outgoing representation are both XML and the data rarely changes, store the data as XML for efficient input/output	manage and query data using XML tools and skills • publish data (make data available) to a 3rd-party tool or program, such as a web service or a browser

Data Query - SQL, SQL/XML, XQuery

SQL

SQL is the standard language for querying Object-Relational data (which we have referred to in this paper as "SQL data"). Let's take another look at Example 7**: Table resolutions_xml** . This table has three columns. The first two - id and legis_num - are numbers extracted from the XML document that is represented in full in the third column, resolutions. Those first two columns could be dates or character strings, and they could be any relational data, not necessarily taken from resolutions. You can query the table resolutions_xml using SQL.

Example 12: Query resolutions - SQL

```
select id, legis_num, resolution
  from resolutions_xml
  where legis_num > 100
  order by legis_num asc
```

Example 12: Query resolutions - SQL selects the id, legis_num and the full resolution, where the legis_num is greater than 100, ordered by legis_num in

ascending order. But if you only have SQL-92 you cannot look inside the resolution column - SQL-92 knows nothing about the XML structure or content. If you want to use SQL to query inside the XML data in the resolutions column, you need the SQL/XML extensions, which first start to appear in SQL:2003.

SQL/XML

SQL:2003 is the first version of SQL that includes the built-in datatype XMLType, plus publishing functions (*XMLElement*, *XMLForest*, *XMLAgg*, and others) to create XML directly from relational data in SQL queries. The XML-related functionality in SQL, starting at SQL:2003, is known collectively as SQL/XML.

Oracle, starting with Oracle9*i*, supports SQL/XML plus some additional extensions – the functions *extract*, *extractValue*, and *existsNode*, to query and extract XML data. The corresponding functions in SQL:2005 are expected to be *XMLQuery*, *XMLExists* and *XMLCast*.

extract and *extractValue* take as arguments an XML document and an XPath expression, and return the result of evaluating the XPath expression against the XML document. *extract* returns a sequence of XML nodes, *extractValue* returns a single value. *existsNode* is another function that accepts the same arguments as *extract* – it returns 1 or 0 depending on whether the XPath matches any node in the document. It is typically used in the WHERE clause of the SQL query .

The SQL standard continues to evolve in this area – for details, see the home page of the SQLX group[7]. In this paper, we use the term "SQL/XML" to cover both the current and proposed extensions to the SQL standard, and the Oracle extensions to SQL, all of which enable XML generation, query, and extraction from within SQL. You have already seen some SQL/XML queries - for example the view creation at Example 8: resolutions_view and Example 10: View body_text_xml.

Note: SQL/XML, part of the SQL standard, should not be confused with SQLXML (without the slash), which is a Microsoft term for something different.

We said that with SQL-92, when you query XML you can only treat that XML as a single thing - it's not possible to "look inside" the XML and query the structure and content of the XML. With the SQL extension functions *extract*, *extractValue*, and *existsNode* you **can** look inside the XML and query on its structure and content. The second argument to *extract*, *extractValue*, and *existsNode* is an XPath expression. XPath 1.0 is a language defined by the W3C. When an XPath expression is applied to an XML document it returns either XML (a node or sequence of nodes) or a value (a number, date, string). It's this XPath expression that gives SQL the expressive power to "look inside" the XML, and find values at a particular place in the XML tree which satisfy particular conditions. For example, Example 13: SQL/XML query - extract, extractValue:

---

[7] http://sqlx.org

Example 13: SQL/XML query - extract, extractValue

```
select
  id AS id ,
  extract( resolution,
    '/resolution[@public-private="public"]/action' ) AS action ,
  extractValue( resolution,
    '/resolution[congress="108"]/official-title') AS title
  from resolutions_xml
```

This query returns:

- the id from the table column "id"

  42

- an XML fragment from the table column "resolution". The fragment returned is the sub-tree starting at the element "action" whose parent is the top-level element "resolution". Only return this sub-tree if the "public-private" attribute of "resolution" equals "public" (else return NULL).

  ```
  <action>
    <action-date>20040311</action-date>
    <action-desc>
      <sponsor>Bereuter</sponsor>
      <cosponsor>Wexler</cosponsor>
      <cosponsor>Gillmor</cosponsor>
      <cosponsor>Shimkus</cosponsor>
      <committee-name>Committee on International Relations
      </committee-name>
    </action-desc>
  </action>
  ```

- the value (content) of the "official-title" element of the top-level element "resolution", where that "resolution" has an attribute "congress" equal to "108".

```
Welcoming the accession of Bulgaria, Estonia,
Latvia, Lithuania, Romania, Slovakia, and Slovenia to the
North Atlantic Treaty Organization (NATO).
```

extract and *extractValue* can also be used in the WHERE clause of an SQL query. And there is another extension function – *existsNode* – to test for existence of an XPath.

Example 14: SQL/XML query - existsNode

```
select
  id AS id
  from resolutions_xml
  where existsNode(resolution,
    '/resolution[legis-num = 558]') = 1
```

Example 14: SQL/XML query - existsNode is an example of an SQL query using *existsNode*. The query selects the id from each row in the resolutions_xml table where the resolution matches the XPath '/resolution[legis-num = 558]' – *i.e.*, where the legis-num equals 558.

XPath is good at expressing positions in an XML document and conditions, but it has some limitations when querying XML. The W3C is currently working on an even more expressive language for querying XML - XQuery 1.0[8]. In the next

[8] http://www.w3.org/TR/xquery/

section we briefly describe XQuery, and show how it can be used on its own or as part of SQL/XML.

XQuery

While XPath describes path expressions, XQuery 1.0 is a complete language for querying XML which encompasses XPath 2.0. The core expression in XQuery is the FLWOR (pronounced "flower") expression. FLWOR stands for "for ... let ... where ... order by ... return", which is the general shape of a FLWOR expression. Example 15: Simple XQuery shows a simple XQuery FLWOR expression.

Example 15: Simple XQuery

```
for $r in doc("/public/oow04/resolution.xml")/resolution
  let $a := $r/action
  where $a/action-date="20040311"
  order by $r/legis-num ascending
  return
<all-sponsors>
  {$a/action-desc/sponsor}
  {$a/action-desc/cosponsor}
</all-sponsors>
```

Results:

```
<all-sponsors>
  <sponsor>Bereuter</sponsor>
  <cosponsor>Wexler</cosponsor>
  <cosponsor>Gillmor</cosponsor>
  <cosponsor>Shimkus</cosponsor>
</all-sponsors>
```

The XQuery in Example 15: Simple XQuery says:

- **for** – iterate over each resolution in the document "/public/oow04/resolution.xml". Note this may be in the file system, or on the internet, or in a database.
- **let** – bind $r/action to the variable $a. This gives XQuery a convenient way to "bookmark" a point in the XML structure, to be easily referred to later in the query.
- **where** – for each resolution specified by the "for" clause, select only those resolutions where resolution/action/action-date equals "20040311".
- **order by** – produce results ordered by resolution/legis-num
- **return** – return the result. Note that you can construct XML elements - such as the new "all-sponsors" element - explicitly in the return clause. If you do, you need to delineate XQuery expressions inside constructed elements with squiggly brackets {}.

XQuery has some obvious advantages over XPath:

- the let clause makes XQuery queries simpler and easier to read by letting you identify a point in the XML structure and refer to it later in the query with a variable. There is no equivalent in XPath

- the return clause allows you to construct new elements that contain the results of XQuery expressions. This means you can return XML in any shape you want. There is no element construction in XPath.
- you can express a join – a query across more than one collection of XML documents – in XQuery, but not in XPath. We describe a simple join query in Example 18: Simple XQuery join
- order by – you can specify an ordering for the result of an XQuery, while XPath results are always in document order
- mixed data model – the result of an XQuery can contain not just XML nodes, but also scalar values
- strong typing – the XQuery data model allows type expressions and typing rules to be applied at compile time (statically), reducing run-time errors and the overhead of run-time error checking

Data Sample 2 - House Voting Record

To show the join capabilities of XQuery, we need to introduce a second piece of data. Example 16: Sample voting record is an XML document that describes the voting record for the resolution in Example 1: Sample resolution - XML.

Example 16: Sample voting record

```
<rollcall-vote>
  <vote-metadata>
    <congress>108</congress>
    <session>2</session>
    <chamber>U.S. House of Representatives</chamber>
    <rollcall-num>99</rollcall-num>
    <legis-num>558</legis-num>
    <vote-question>On Motion to Suspend the Rules and Agree,
    as Amended</vote-question>
    <vote-type>2/3 YEA-AND-NAY</vote-type>
    <vote-result>Passed</vote-result>
    <action-date>20040330</action-date>
    <action-time>16:18</action-time>
    <vote-desc>Welcoming accession of Bulgaria, Estonia, Latvia,
    Lithuania, Romania, Slovakia, and Slovenia to NATO</vote-desc>
  </vote-metadata>
  <vote-data>
    <recorded-vote>
      <legislator party="D" state="HI">Abercrombie</legislator>
      <vote>Yea</vote>
    </recorded-vote>
    <recorded-vote>
      <legislator party="D" state="NY">Ackerman</legislator>
      <vote>Yea</vote>
    </recorded-vote>
    <recorded-vote>
      <legislator party="R" state="AL">Aderholt</legislator>
      <vote>Yea</vote>
    </recorded-vote>
    <recorded-vote>
      <legislator party="R" state="MO">Akin</legislator>
      <vote>Yea</vote>
    </recorded-vote>
    <recorded-vote>
      <legislator party="D" state="LA">Alexander</legislator>
      <vote>Yea</vote>
    </recorded-vote>
    <recorded-vote>
      <legislator party="R" state="AK">Young (AK)</legislator>
      <vote>Yea</vote>
    </recorded-vote>
    <recorded-vote>
      <legislator party="R" state="FL">Young (FL)</legislator>
      <vote>Yea</vote>
    </recorded-vote>
  </vote-data>
</rollcall-vote>
```

Example 17: Simple XQuery - vote count is a simple XQuery that returns some information about the votes counted.

Example 17: Simple XQuery - vote count

```
for $v in doc("/public/oow04/vote.xml")/rollcall-vote
  let $d := $v/vote-data
  return
<vote-count>
  <total>
    {count($d/recorded-vote[vote="Yea"])}
  </total>
  <republican>
    {count($d/recorded-vote[legislator/@party="R" and vote="Yea"])}
  </republican>
  <democrat>
    {count($d/recorded-vote[legislator/@party="D" and vote="Yea"])}
  </democrat>
</vote-count>
```

Results:

```
<vote-count>
  <total>7</total>
  <republican>4</republican>
  <democrat>3</democrat>
</vote-count>
```

Now that we have the resolution and the voting record, we can correlate (join) the data in the two collections. For example, Example 18: Simple XQuery join produces the title, sponsor and Yea-vote-count for each resolution for which we have a voting record. We show the query and results for a single resolution and a single voting record, but of course these queries are only interesting when applied to a large collection of resolutions and voting records.

Example 18: Simple XQuery join

```
for $r in doc("/public/oow04/resolution.xml")/resolution ,
  $v in doc("/public/oow04/vote.xml")/rollcall-vote
  let $d := $v/vote-data/recorded-vote[vote="Yea"]
  where $r/legis-num = $v/vote-metadata/legis-num
  return
<resolution>
  <title>{$r/official-title/text()}</title>
  <sponsor name="{$r/action/action-desc/sponsor}"/>
  <yea-votes>{count($d)}</yea-votes>
</resolution>
```

Results:

```
<resolution>
  <title>Welcoming the accession of Bulgaria, Estonia, ... </title>
  <sponsor name="Bereuter"></sponsor>
  <yea-votes>7</yea-votes>
</resolution>
```

In the XQuery examples here, we have said nothing about how the data is stored or about how the data should be returned to the user. The XQuery specification says very little about input data, output data, or the host language – all these things are purposely left "implementation-defined" - you could say that XQuery is by definition "context-free". These examples could query data stored in a database, or in a file system. They could run in an SQL engine, in a Java program, or both (a Java engine that talks to a database via *e.g.*, JDBC). In "XQuery in SQL/XML" we describe how XQuery might be used in the context of an SQL (Object-Relational) database with SQL/XML.

XQuery in SQL/XML

We have already looked at some of the XML extensions to SQL, collectively called SQL/XML. We have also looked at some Oracle extension functions that make use of XPath to "look inside" an XML document. The next major release of SQL/XML is expected to include a couple more functions – *XMLQuery* and *XMLTable* – that make use of XQuery. These SQL/XML functions will be part of the Oracle database 10*g* Release 2.

XMLQuery is a function that naturally fits in the SQL select clause. It takes an XQuery string and a set of arguments for XQuery variables and the context item, and returns an XMLType instance. Example 19: Simple XMLQUERY in SQL/XML shows the XQuery in Example 17: Simple XQuery - vote count as an SQL/XML statement. Note the "select ... from dual", where dual is a "dummy table" with one row. This provides a convenient SQL harness for the XQuery. Any tool or application that uses XMLQuery should not expose this construct – *e.g.*, in Oracle's SQL*Plus command-line tool, you would run this XQuery by simply typing the keyword "XQUERY" followed by the XQuery – SQL*Plus does not require the user to type in "select … from dual".

Example 19: Simple XMLQUERY in SQL/XML

```
select
  XMLQUERY(
    'for $v in
      doc("/public/xml2004/vote.xml")/rollcall-vote
    let $d := $v/vote-data
    return
<vote-count>
  <total>
    {count($d/recorded-vote[vote="Yea"])}
  </total>
  <republican>
    {count($d/recorded-vote[legislator/@party="R" and vote="Yea"])}
  </republican>
  <democrat>
    {count($d/recorded-vote[legislator/@party="D" and vote="Yea"])}
  </democrat>
</vote-count>'
    returning content) AS result from dual
```

Results:

```
RESULT
------
<vote-count>
  <total>7</total>
  <republican>4</republican>
  <democrat>3</democrat>
</vote-count>
```

Similarly, Example 20: Simple XMLTABLE in SQL/XML is a simple SQL/XML query using the *XMLTable* function. *XMLTable* takes in an XPath or XQuery string, and returns the result as a table. This result can be included in the from clause of an SQL query, or it can be used to create an SQL view.

Example 20: Simple XMLTABLE in SQL/XML

```
select * from
  xmltable('for $r in
    doc("/public/oow04/resolution.xml")/resolution
    let $a := $r/action
    where $a/action-date="20040311"
    order by $r/legis-num ascending
    return
<all-sponsors>
  {$a/action-desc/sponsor}
  {$a/action-desc/cosponsor}
</all-sponsors>')
```

Results:

```
COLUMN_VALUE
------------
<all-sponsors>
  <sponsor>Bereuter</sponsor>
  <cosponsor>Wexler</cosponsor>
  <cosponsor>Gillmor</cosponsor>
  <cosponsor>Shimkus</cosponsor>
</all-sponsors>
```

In Example 20: Simple XMLTABLE in SQL/XML, each item of the XQuery result sequence is converted into a row with a single XMLType column. There is no further decomposion of the XMLType into relational columns. The next example shows further decomposion of the XMLType value into multiple relational columns via the COLUMNS clause in *XMLTable*. Example 21: XMLTABLE in SQL/XML - resolutions_sponsors uses *XMLTable* to model Table 2: resolutions_sponsors.

Example 21: XMLTABLE in SQL/XML - resolutions_sponsors

```
select
  resolution_id,
  sponsor,
  sponsor_type
from xmltable('for $r in
    doc("/public/xml2004/resolution.xml")/resolution
    let $a := $r/action/action-desc
    where $r/action/action-date="20040311"
    return
     (
       <sponsor>
         <resid>{$r/@dms-id}</resid>
         <type>primary</type>
         <name>{$a/sponsor/text()}</name>
       </sponsor>,
       for $j in $a/cosponsor
       return
         <sponsor>
           <resid>{$r/@dms-id}</resid>
           <type>cosponsor</type>
           <name>{$j/text()}</name>
         </sponsor>
     )'
   COLUMNS
       resolution_id  number PATH 'resid',
       sponsor        varchar2(4000) PATH 'name/text()',
       sponsor_type   varchar2(200) PATH 'type'
  ) resolutions_sponsors
```

Results:

```
RESOLUTION_ID SPONSOR      SPONSOR_TYPE
------------- ------------ ------------
           42 Bereuter     primary
           42 Wexler       cosponsor
           42 Gillmor      cosponsor
           42 Shimkus      cosponsor
```

Example 21: XMLTABLE in SQL/XML - resolutions_sponsors is a bit more complex than Example 20: Simple XMLTABLE in SQL/XML. First, the XQuery models a repeating field in the denormalized resolutions_sponsors table (co_sponsor), using a sub-query (starting at "for $j ..."). Second, *XMLTable* includes the "COLUMNS" keyword to describe the *shape* of the table. The table returned by *XMLTable* is, by default, a single-column table where the column is named "COLUMN_VALUE". Using the "COLUMNS" keyword we can name each column in the output table, assign the value of a PATH to each column, and cast it to an SQL type. The PATH is evaluated relative to the XML returned by the XQuery in the first argument.

Our previous XQuery examples (Example 17: Simple XQuery - vote count and Example 18: Simple XQuery join) showed how you can execute standalone XQuery expressions (*i.e.*, run XQueries against XML files stored in the database repository) within SQL. You can also query SQL expressions (tables, views, or the results of SQL queries) using XQuery. SQL/XML lets you pass SQL expressions to the XQuery as external variables and/or the context item. In Example 22:

Passing SQL values to XQuery, the XML document to be queried is stored as an XMLType column (*resolution*) of the *resolutions_xml* table. This column is passed in as external variable "res" to the XQuery expression.

Example 22: Passing SQL values to XQuery

```
select xmlquery('
 for $r in $res/resolution
    let $a := $r/action
    where $a/action-date="20040311"
    order by $r/legis-num ascending
    return
<all-sponsors>
  {$a/action-desc/sponsor}
  {$a/action-desc/cosponsor}
</all-sponsors>' passing resolution as "res" returning content)
from resolutions_xml
```

Results:

```
RESULT
------
<all-sponsors>
  <sponsor>Bereuter</sponsor>
  <cosponsor>Wexler</cosponsor>
  <cosponsor>Gillmor</cosponsor>
  <cosponsor>Shimkus</cosponsor>
</all-sponsors>
```

In this section you have seen how XQuery, and not just XPath, can be used as part of SQL/XML to query data in an XML representation and return results either in an SQL or an XML representation. The data that is queried by XMLQuery and XMLTABLE can be stored as XML Type objects in Object-Relational tables, or it can be stored as "files" in the Oracle XML DB repository. A third source of data for XMLQuery and XMLTable is relational data stored in tables. For that, Oracle has created an extension function - *ora:view* - that takes in a relational table (or a view) and outputs an XML representation of that table (or view).

Example 23: XMLQUERY and ora:view

```
select xmlquery('
 for $r in ora:view("resolutions_xml")/ROW/RESOLUTION
    let $a := $r/resolution/action
    where $a/action-date="20040311"
    order by $r/legis-num ascending
    return
<all-sponsors>
  {$a/action-desc/sponsor}
  {$a/action-desc/cosponsor}
</all-sponsors>' returning content) AS RESULT
from dual
```

Example 23: XMLQUERY and ora:view is the same query as Example 22: Passing SQL values to XQuery, but instead of passing in the column value by using the "passing …" keyword, we use the *ora:view* Oracle extension. *ora:view* takes in a table name and returns an XML view of that table, where each row is a ROW element and each column is a COLUMN-NAME element.

SQL and XML Duality – Query

Table 6: SQL and XML Storage/Representation Duality

	STORAGE/REPRESENTATION			
QUERY		**Relational Tables**	**XML Type**	**Mixed**
	SQL	SQL query	Use *extract*, *extractValue*, *existsNode* (with XPath argument), or XMLTABLE (with XPath or XQuery argument)	Use *extract*, *extractValue*, *existsNode*, or XMLTABLE function. Consider creating an SQL View.
	XQuery	Use XMLQUERY function over *ora:view*	Use XMLQUERY function over *ora:view* or Oracle XML DB Repository files	Use XMLQUERY over *ora:view*. Consider creating an XML View.
	Mixed	Use XMLQUERY function over *ora:view* in any SQL query	Use *extract*, *extractValue*. *existsNode*; or XMLTABLE + XMLQUERY	Use *extract*, *extractValue*, *existsNode*; or XMLTABLE + XMLQUERY

Try Doing This With XQuery ….

XQuery is a powerful, concise, strongly-typed language for querying XML., but it is still in its infancy. At the time of writing (December 2004) the XQuery Working Drafts (the suite of specifications) is about to go into Last Call[9] for the second time. That's still three steps away (Candidate Recommendation, Proposed Recommendation, W3C Recommendation) from version 1.0 of the "standard" (in W3C terms, a *Recommendation*). SQL, on the other hand, has had several decades of definition, many thousands of users, and many iterations of the standard. So SQL is a more complete, robust standard than XQuery, with many capabilities that have not yet made it into XQuery.

Many consider *update* and *full-text* to be the most important features that are lacking in the emerging XQuery standard. Fortunately, SQL/XML with Oracle extensions supports both.

Update

The definition of "query" offered by http://searchDatabase.com includes this:

[9] See the W3C Process document at http://www.w3.org/Consortium/Process/, especially http://www.w3.org/Consortium/Process-20010719/tr.html#Reports

> "*A database query can be either a select query or an action query. A select query is simply a data retrieval query. An action query can ask for additional operations on the data, such as insertion, updating, or deletion.*"

By this definition, the ability to update data is an essential part of query. At the time of writing (December 2004) the W3C has a Task Force looking into update capabilities for XQuery, but XQuery Update is likely to appear only *after* V1.0 of XQuery. But you can update XML data in-place today with SQL/XML and Oracle's extensions.

Example 24: Update XML

```
UPDATE votes_xml
  SET vote = UPDATEXML( vote,
    '/rollcall-vote/vote-data/recorded-vote[2]/vote/text()',
    'Nay' )
  WHERE id = 42
```

Example 24: Update XML updates voting record 42, changing the second recorded-vote (Abercrombie's vote) from 'Yea' to 'Nay'. Note the second argument to UPDATEXML (an Oracle extension) can be any XPath expression – we chose to use an XPath expression that picks the second recorded-vote.

Full-Text

Full-Text search is different from string-comparison, since it is based on searching for *words* or *tokens* in a piece of text, as opposed to searching for exact *substrings* within a string. In our example data, you could do a *string-based search* for "Bulgaria" within the official-title element of a resolution. XQuery and XPath have the "contains" function for doing this kind of search. But if you want to find the *word* "Bulgaria", without having to worry about uppercase *vs.* lowercase spelling, and without finding words that merely contain "Bulgaria" (such as "Bulgarianization"), then you need *full-text search*. Note that string-based searches for "man" (as in "Romania"), "love" (as in "Slovenia"), and "Organ" (as in "Organization") would all return our sample resolution as a hit – full-text searches would not.

Oracle's XPath/XQuery extension function – *ora:contains* – looks a lot like the standard XPath function *contains*, but instead of taking just a string as an argument it takes a full-text search expression. The full-text search expression can contain words and/or phrases, combined using the boolean operators *and*, *or* and *not*. It can also express full-text search concepts such as proximity searching (*near*), which finds two words that are near to each other, or within a specified number of words.

You can use *ora:contains* inside an XPath in *extract*, *extractValue*, or *existsNode*, or as part of one of the XQuery extensions XMLQUERY and XMLTABLE. Example 25: Full-Text returns the titles of resolutions that contain the word "Bulgaria" near the word "Slovenia".

Example 25: Full-Text

```
SELECT XMLQUERY(
  'for $r in
    $res/resolution/official-title[
      ora:contains(., "bulgaria NEAR slovenia")>0]
    return
      $r' passing resolution as "res" returning content)
AS result FROM resolutions_xml
```

CONCLUSION

We started this paper by describing the duality of SQL and XML storage and representation - you can store data either as SQL or as XML and then, whatever storage choice you made, represent the data as either SQL or XML. There is also a duality of SQL and XML query. It is impossible to query "inside" a piece of XML using unextended SQL. But we have shown that the recent SQL extensions in SQL/XML allow you to use SQL to query both SQL and XML data, including querying "inside" XML using XPath or XQuery. This strategy lets you choose the data storage and representation that makes most sense from a technical, business, and organizational perspective, while satisfying your customers' and suppliers' requirements for data in a particular format for interchange and publishing. And you can choose the query language that best suits your needs, tools and skills set.

Let's revisit the hypothesis of this paper – that SQL/XML and XQuery are not ***competing***, but rather ***complementary***, technologies. We have shown that if you need to use XQuery to query your data – perhaps queries are coming in from some application that only speaks XQuery – that data can be stored in a database either object-relationally in tables or ias an XML type. And we have shown that if you want to use SQL to query your data - perhaps you have tools and available skills for SQL but not for XQuery - then you can use SQL/XML (leveraging XPath and possibly XQuery) to do the bits that need to look "inside" an XML structure.

When we discussed storage and representation, we recommended that you think carefully about your requirements before deciding between the many storage and representation options (and combinations of options) available. In the same way we recommend that you consider the query options - SQL, SQL/XML, XPath, and XQuery - and the many combinations of query options available before making a decision.

Querying XML
December 2004
Author: Stephen Buxton
Contributing Authors: Muralidhar Krishnaprasad, Zhen Hua Liu, Jim Melton

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