# XML enabled database

**support for XML in Microsoft SQL Server 2000 & 2005**

Martin Malý
martin@php-compiler.net

# Agenda

- Three ways of approach
- Microsoft SQL 2000
  - Storing XML as text
  - SQLXML
  - FOR XML command
- Microsoft SQL 2005
  - XML datatype
  - new methods + examples

# Three ways of approach

Store and manage XML

1. use XML-based database engine (such as dbXML or Tamino)
   - good for "XML only" data
2. use relational database back-end and translate to XML (SQLXML 3.0)
   - suitable if relation database already exists
   - Microsoft SQL Server 2000
3. use native XML datatype
   - introduced in Microsoft SQL Server 2005

# Storing XML as text

- database used just as a storage of XML snippets
- possible types:
  - varchar – limitation of length (8000 chars)
  - text, ntext – non-Unicode and Unicode length up to $2^{31}-1$
  - image – binary data, length up to $2^{31}-1$

# SQLXML

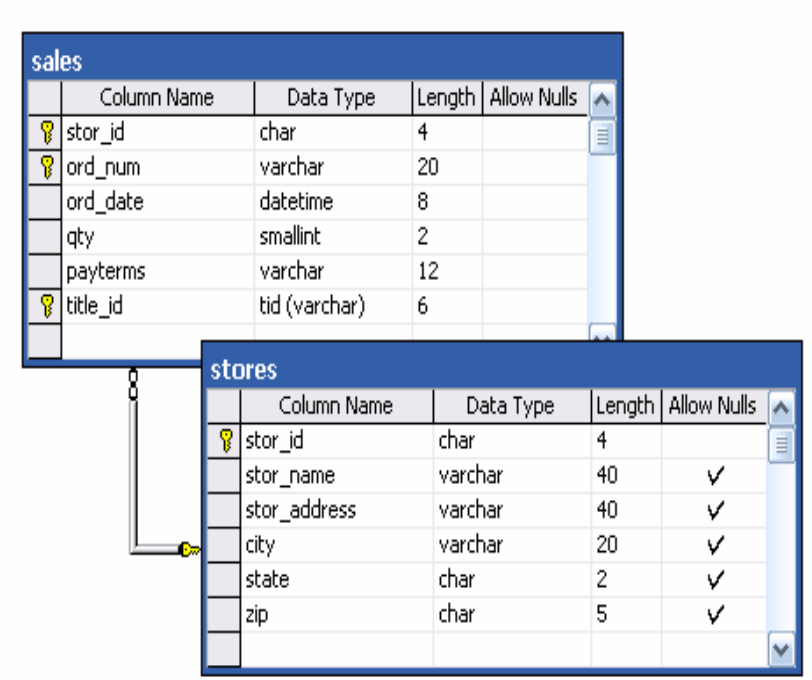
- history – module within Microsoft SQL 7
- Microsoft SQL 2000
  - data within relation tables
  - XML used as query output
  - FOR XML command added to T-SQL

# FOR XML command

- **FOR XML RAW**: Returns XML elements with the "row" prefix (ex: "<row tProduct …>"). Each column in a table is represented as an attribute and null column values aren't included.
- **FOR XML AUTO**: Returns nested XML elements, based on which tables are listed in the "FROM" part of the query, and which fields are listed in the "SELECT" part.
- **FOR XML EXPLICIT**: Explicit mode is the most complex shaping method used in SQL Server 2000. It allows users to query a data source in such a way that the names and values of the returned XML are specified before the query batch is executed.

# Example 1

```
SELECT
    store.stor_id as Id,
    stor_name as Name,
    sale.ord_num as
    OrderNo,
    sale.qty as Qty
FROM
    stores store
    INNER JOIN sales sale ON
    store.stor_id =
    sale.stor_id
ORDER BY stor_name
FOR XML <MODE>
```



# Example 1 – "classic result"

| Id | Name | OrderNo | Qty |
|---|---|---|---|
| 7066 | Barnum's | A2976 | 50 |
| 7066 | Barnum's | QA7442.3 | 75 |
| 8042 | Bookbeat | 423LL922 | 15 |
| 8042 | Bookbeat | 423LL930 | 10 |
| 8042 | Bookbeat | P723 | 25 |
| 8042 | Bookbeat | QA879.1 | 30 |
| 7131 | Doc-U-Mat: Quality Laundry and Books | N914008 | 20 |
| 7131 | Doc-U-Mat: Quality Laundry and Books | N914014 | 25 |
| 7131 | Doc-U-Mat: Quality Laundry and Books | P3087a | 20 |
| 7131 | Doc-U-Mat: Quality Laundry and Books | P3087a | 25 |
| 7131 | Doc-U-Mat: Quality Laundry and Books | P3087a | 15 |
| 7131 | Doc-U-Mat: Quality Laundry and Books | P3087a | 25 |
| 6380 | Eric the Read Books | 6871 | 5 |
| 6380 | Eric the Read Books | 722a | 3 |
| 7896 | Fricative Bookshop | QQ2299 | 15 |
| 7896 | Fricative Bookshop | TQ456 | 10 |
| 7896 | Fricative Bookshop | X999 | 35 |
| 7067 | News & Brews | D4482 | 10 |
| 7067 | News & Brews | P2121 | 40 |
| 7067 | News & Brews | P2121 | 20 |
| 7067 | News & Brews | P2121 | 20 |

# Example 1 – RAW mode

- fixed element <row>
- no control over element naming

```
<?xml version="1.0" ?>
- <Stores xmlns:sql="urn:schemas-microsoft-com:xml-sql">
    <row Id="7066" Name="Barnum's" OrderNo="A2976" Qty="50" />
    <row Id="7066" Name="Barnum's" OrderNo="QA7442.3" Qty="75" />
    <row Id="8042" Name="Bookbeat" OrderNo="423LL922" Qty="15" />
    <row Id="8042" Name="Bookbeat" OrderNo="423LL930" Qty="10" />
    <row Id="8042" Name="Bookbeat" OrderNo="P723" Qty="25" />
    <row Id="8042" Name="Bookbeat" OrderNo="QA879.1" Qty="30" />
```

# Example 1 – AUTO mode

- node relationship – order as declared in query
- aliases effect generated names

```
<?xml version="1.0" ?>
- <Stores xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  - <Store Id="7066" Name="Barnum's">
      <Sale OrderNo="A2976" Qty="50" />
      <Sale OrderNo="QA7442.3" Qty="75" />
    </Store>
  - <Store Id="8042" Name="Bookbeat">
      <Sale OrderNo="423LL922" Qty="15" />
      <Sale OrderNo="423LL930" Qty="10" />
      <Sale OrderNo="P723" Qty="25" />
      <Sale OrderNo="QA879.1" Qty="30" />
    </Store>
  - <Store Id="7131" Name="Doc-U-Mat: Quality Laundry and Books">
      <Sale OrderNo="N914008" Qty="20" />
      <Sale OrderNo="N914014" Qty="25" />
```

# AUTO mode disadvantages

- hierarchical structures possible
  - only in a linear fashion
  - parent node can only have one child and vice versa.
- using aliases to create element and attribute names sometimes influence readability of the query.
- cannot have both attributes and elements within the document
  - either all elements (ELEMENTS keyword)
  - or attributes (default)

# Example 1 – EXPLICIT mode

```
-The Store Data
SELECT 1 as Tag,
NULL as Parent,
        s.stor_id as [store!1!Id],
        s.stor_name     as [store!1!Name],
        NULL            as[sale!2!OrderNo],
        NULL            as [sale!2!Qty]
 FROM stores s

UNION ALL

-- The Sale Data
SELECT 2, 1,
        s.stor_id,
        s.stor_name,
        sa.ord_num,
        sa.qty
FROM stores s, sales sa

WHERE s.stor_id = sa.stor_id
ORDER BY [store!1!name]

FOR XML EXPLICIT
```

## Universal Table

- required for sqlxml.dll to generate XML
- contains data and metadata

| Tag | Parent | store!1!id | store!1!name | sale!2!orderno | sale!2!qty |
|---|---|---|---|---|---|
| 1 | NULL | 7066 | Barnum's | NULL | NU LL |
| 2 | 1 | 7066 | Barnum's | A297650 | 50 |
| 2 | 1 | 7066 | Barnum's | QA7442 | 375 |
| 1 | NULL | 8042 | Bookbeat | NULL | NU LL |
| 2 | 1 | 8042 | Bookbeat | 423LL9 | 2215 |

# Example 1 – EXPLICIT mode

```
-The Store Data
SELECT 1 as Tag,
NULL as Parent,
        s.stor_id as [store!1!Id],
        s.stor_name     as [store!1!Name],
        NULL            as[sale!2!OrderNo],
        NULL            as [sale!2!Qty]
 FROM stores s

UNION ALL

-- The Sale Data
SELECT 2, 1,
        s.stor_id,
        s.stor_name,
        sa.ord_num,
        sa.qty
FROM stores s, sales sa

WHERE s.stor_id = sa.stor_id
ORDER BY [store!1!name]

FOR XML EXPLICIT
```

- metadata
  - Tag + Parent > hierarchy
  - first SELECT has NULL parent
- data [1!2!3!4]
  - 1 = element name
  - 2 = tag number
  - 3 = attribute / element
  - 4 = optional **element** keyword (attribute by default)

# Example 1 – EXPLICIT mode

```
-The Store Data
SELECT 1 as Tag,
NULL as Parent,
        s.stor_id as [store!1!Id],
        s.stor_name     as [store!1!Name],
        NULL            as[sale!2!OrderNo],
        NULL            as [sale!2!Qty]
 FROM stores s

UNION ALL

-- The Sale Data
SELECT 2, 1,
        s.stor_id,
        s.stor_name,
        sa.ord_num,
        sa.qty
FROM stores s, sales sa

WHERE s.stor_id = sa.stor_id
ORDER BY [store!1!name]

FOR XML EXPLICIT
```

```
<?xml version="1.0" ?>
<stores xmlns:sql="urn:schemas-microsoft-com:xml-sql">
- <store Id="7066" Name="Barnum's">
    <sale OrderNo="A2976" Qty="50" />
    <sale OrderNo="QA7442.3" Qty="75" />
  </store>
- <store Id="8042" Name="Bookbeat">
    <sale OrderNo="423LL922" Qty="15" />
    <sale OrderNo="423LL930" Qty="10" />
    <sale OrderNo="P723" Qty="25" />
    <sale OrderNo="QA879.1" Qty="30" />
    <sale OrderNo="P3087a" Qty="20" />
    <sale OrderNo="P3087a" Qty="25" />
    <sale OrderNo="P3087a" Qty="15" />
    <sale OrderNo="P3087a" Qty="25" />
  </store>
- <store Id="7131" Name="Doc-U-Mat: Quality Laundry and Books">
    <sale OrderNo="N914008" Qty="20" />
    <sale OrderNo="N914014" Qty="25" />
  </store>
- <store Id="6380" Name="Eric the Read Books">
    <sale OrderNo="6871" Qty="5" />
    <sale OrderNo="722a" Qty="3" />
  </store>
```

# Microsoft SQL 2005 "Yukon"

- native XML data type
- new commands added to T-SQL (bound to XML columns)
  - query()
  - value()
  - exist()
  - nodes()
  - modify()
- XML columns
  - typed (using XSD)
    + validation, primary keys
  - untyped

# XML indexing

- provided to speed up queries over XML columns
- extensively use B+ trees
- **primary** XML index on an XML column creates a B+tree index on all tags, values, and paths of the XML instances in the column
- provides
  - efficient evaluation of queries on XML data
  - reassembly of the XML result from the B+tree
    (while preserving document order and document structure)
- **secondary** XML indexes can be created on an XML column to speed up different classes of commonly occurring queries
  - PATH index for path-based queries (path, value)
    (common case is the use of the exist() method on XML columns in the WHERE clause of a SELECT statement )
  - PROPERTY index for property bag scenarios (PK, path, value)
  - VALUE index for value-based queries (value, path)

# query()

- search through a larger XML structure to find a set of data based on an XML Query (XQuery) definition

```
SELECT
      pk,
      xCol.query('/doc[@id = 123]//section')
FROM docs
```

# exist()

- optimized method to screen XML data the same way as a relational WHERE clause
- instead of retrieving a value from your XML data, the condition is passed to the XML processor
- only records that match the condition are retrieved
- returns 1 if the XQuery expression evaluates to non-null node list, otherwise it returns 0.

```
SELECT ShippingDate, Details
FROM Order
WHERE
      Details.exist ('/order[@orderyear = "2004"]') = 1
```

# Example 2 - exist()

```
<order orderyear="2004" orderno="1234" loccode="1234567">
  <description>business</description>
  <item>
    <product>herring</product>
    <quantity>2</quantity>
  </item>
  <item>
    <product>haddock</product>
    <quantity>4</quantity>
  </item>
  <price>21.12</price>
</order>
```

- Table **Order** having
  - XML column 'Details'
  - "normal" columns 'ShippingDate' & 'ShippingCode'.
- index

  CREATE PRIMARY
  XML INDEX ixDetails
  on Order (Details)

```
SELECT ShippingDate, Details
FROM Order
WHERE Details.exist ('/order[@orderyear = "2004"]') = 1
```

# value()

- returns a specific value from within your XML structure
- limitation - value must be a single instance (string, number, etc.)
- cannot be a subset (i.e., node) of your XML structure

```
SELECT xCol.value(
    'data((/doc//section[@num = 3]/heading)[1])',
    'nvarchar(max)')
FROM   docs
```

# nodes()

- returns a subset of your XML structure in the form of a node
- result can then be used by other methods, such as exist() and value() to pull out repeating values that might be embedded within a single XML column
- the key is that by using the XQuery syntax, which is native to XML, you can embed your queries against the data in a single relational column

# modify()

- method lets you insert and update values and nodes that are contained within an XML column
- accepts both INSERT and UPDATE statements not only for scalar values but also for entire subtrees
- allow to add specific child elements to a collection of items
- by leveraging XQuery statements within each XML column command, one can manipulate the individual components contained within your custom XML structure

# Q&A

...

# Resources

- http://www.microsoft.com
- http://www.sitepoint.com/article/data-as-xml-sql-server
- http://www.developer.com/db/article.php/3294151
- http://www.15seconds.com/issue/001102.htm
- http://dnjonline.com/article.aspx?ID=sep04_yukon_xml
- http://www.yukonxml.com/
- http://www.msdn.net/SQL/sqlreldata/XML/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp