XML and Relational Database Management Systems: Inside Microsoft® SQL Server™ 2005

Michael Rys
Microsoft
mrys@microsoft.com

ABSTRACT

Microsoft® SQL Server™ has a long history of XML support dating back to 1999. While first concentrating on enabling the transport of relational data via XML with the SQL Server 2000 release, SQL Server 2005 now additionally provides native XML storage and query support. This part of the tutorial will provide an insight into how SQL Server 2005 fits XML functionality into its core relational database management framework.

1. INTRODUCTION

Soon after the XML recommendation was finalized, XML was most commonly used as the common transport syntax for data exchange. Since large amounts of the interesting data that needed to be exchanged are stored inside relational database systems, it quickly became important to provide mechanisms to facilitate this exchange by generating and consuming XML in the context of relational database management systems. The most important aspect of this first generation XML support is the ability to publish existing relational data in XML form (*XML Publishing*) and then to decompose such published data back into the existing relational structures (*Shredding*).

Microsoft SQL Server was the leader in this technology. Early solutions focused on mid-tier approaches, where relational data is converted into XML outside the database engine. One example is the ADO recordset XML format. Starting with SQL Server 2000, SQL Server offers both a mid-tier and server-side approach for publishing and shredding XML [30].

The mid-tier approach provides a bi-directional XML view that is defined by mapping an XML Schema to the relational schema. The view then can be used to query the data using an XPath subset that is translated internally into SQL queries (so-called FOR XML EXPLICIT queries). The XML view is also used to shred XML data into relational form using a schema-driven approach with XML bulkload, and to support data modification using updategrams [31].

The server-side functionality provides a rowset-to-XML aggregator, called FOR XML which provides different modes for XML Publishing. The complementary functionality of generating a rowset from XML is called OpenXML that provides a query driven shredding mechanism. [30] gives an overview on these approaches and we will look more closely at FOR XML in the context of SQL Server 2005's XML Publishing functions.

While the publishing and shredding requirement is still one of the most important use cases for integrating XML and relational database systems, it is not the only one. With the advent of XML object serializations and the increased use of XML for representing documents and forms, an increasing number of customers want to store their XML documents inside a database management system without having to "shred" them into a relational structure or lose the ability to query into the structure of the XML.

To address this need, SQL Server 2005 adds native XML management capabilities along the functionality outlined in the introduction of this tutorial. While some of the functionality – such as the XML data type – follows the SQL-2003 standard, others – such as the publishing functionality – do not, since they were designed before the standardization effort and provide easier-to-use approaches. Instead of building a new XML-only database system, SQL Server 2005 deeply integrates the XML capabilities into the existing database management framework that provides general services such as backup, restore, replication, concurrency control etc., while both extending and leveraging the relational storage and query infrastructure.

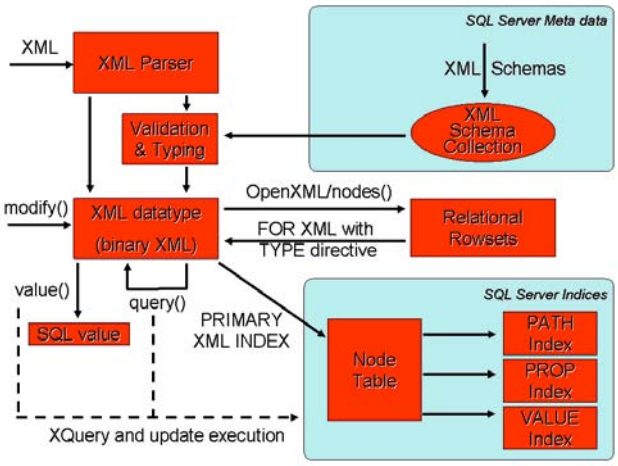


Figure 1: SQL Server 2005 XML Architecture overview

Figure 1 provides a high-level architectural diagram of the native XML support in SQL Server 2005 which will be used as the basis for this tutorial. The remainder of the SQL Server part of the tutorial will present both an overview of the functional extensions as well as lower-level insights into how they integrate into the existing relational framework. First, it will cover the logical and physical organization and storage of native XML data and how data is typed and validated using collections of XML Schemas. Then, the tutorial will look at how the data is queried and updated using XQuery and update extensions and how such expressions are mapped into internal operator trees. The tutorial will discuss the XML indexing framework to provide efficient query execution and will conclude with some examples on the extended XML Publishing functionality.

The tutorial will not cover the mid-tier XML support such as the SQLXML component [31] or go into details of how to use the XML functionality of the .Net Framework in the context of an in-process CLR user-defined function or the full-text search capa-

SIGMOD 2005, June 14–16, 2005, Baltimore, Maryland, USA.

bilities. For this and more detailed aspects, we would like to refer to the collection of whitepapers at [32].

2. STORING XML IN SQL Server 2005

SQL Server, like any other relational database system, allows the user to store markup data in a CLOB or BLOB type (called nvarchar(max) and varbinary(max), respectively) to preserve *textual fidelity*. However, these types provide no semantic guarantee about the XML well-formedness or provide any direct queryability. Basically, the XML data either needs to be processed on the mid-tier or via a CLR user-defined function, which necessitates parsing and main-memory processing outside of the query processor and thus the optimizer, or a runtime conversion to the built-in XML data type. Full-text search can be applied using the general full-text search capabilities, assuming an XML aware full-text filter (called IFilter) is available for building the full-text index.

2.1 Logical Model of the XML data type

In order to provide structurally-aware storage of arbitrary XML documents, SQL Server 2005 adds native XML support based on the XML data type provided by the ISO/ANSI SQL-2003 standard [13]. As in the standard, the XML data type can consist not only of well-formed XML documents but also allows XML fragments that can have an arbitrary number of top-level elements or text nodes underneath the document node. As explained in the tutorial introduction, the data type provides *XML fidelity*.

However, since SQL Server 2005 provides XQuery support and allows the use of XML Schemas to not only validate but also type the data (see section 3), the actual logical model of the XML data type is not based on the XML Information Set [14] but on the XQuery 1.0 Data Model [15] (thus anticipating the new version of the SQL standard). The XML data type can be used anywhere SQL scalar types can be used, including columns, variables and parameters. For example, the following statement defines a column `doc` of type XML in table `XDoc`:

```
create table XDoc (doc XML)
```

SQL Server 2005 provides parsing from a SQL string type (taking both the codepage and the XML encoding into account) or a SQL binary type (only looking at the XML encoding and treating the XML text as a byte sequence) to the XML type as part of the common implicit and explicit CAST/CONVERT semantics. Different conversion options are available to influence whitespace preservation and limited DTD processing. For example, the following insert statement converts the string constant in UTF-16 encoding into XML while inserting the value into the table `XDoc`:

```
insert into XDoc Values (N'<document/>')
```

2.2 Physical Model of the XML data type

Parsing the XML not only checks for the extended well-formedness constraint, but also generates an internal representation of the XML at the physical level. Similar to how relational databases map the logical concepts of rows and columns to a byte-layout on disk, the XML data type's physical model is a byte level representation of the logical concepts such as element and attribute nodes This internal format can be considered a binary encoding of the XML that provides a closer representation of the XQuery data model, which is easier to process in our infrastructure (such as defining indices or performing partial updates). It also provides an average size reduction of 20% to 30% over the fully expanded XML in UTF-16 encoding. The binary XML representation is faster to parse than its textual counterpart, especially when its content is typed. The binary XML can also be shipped directly to database clients such as ADO.Net if they indicate that they understand the binary XML format, thus avoiding the serialization of the XML into a textual representation at the server and the costlier text parsing on the client. If the API provides an XML-specific API such as SAX, DOM or XmlReader, then the client component can operate directly on the binary representation and achieve better performance.

SQL Server 2005 uses the existing BLOB infrastructure of the storage engine to persist the binary XML data. This means, for example, that a single XML instance can be up to 2GBytes in its binary encoding and benefits from the existing storage optimizations such as when the value is stored in-row and when out-of-row. In order to support XQuery expressions on the XML, SQL Server provides additional access paths to the data (see section 4).

3. XML TYPING AND VALIDATION

The basic, unadorned XML data type contains unvalidated XML data that only needs to satisfy the well-formedness constraints imposed by the type itself. In some scenarios, additional constraints on the XML structure and markup content need to be enforced when storing the XML data. For example, a purchase order contract column should only contain XML documents that satisfy the content model of purchase order contracts. Therefore, SQL Server provides a new metadata object called an *XML Schema collection* as a way to store W3C XML Schemas [7].

3.1 XML Schema Collections

An XML Schema collection is a collection of multiple XML Schema components (e.g., an element or type declaration) from potentially many different XML Schemas of different target namespaces, and can be used to constrain, validate and type XML data type instances. It is identified by a SQL identifier and its information is fully accessible through relational catalog tables. For example, the expression

```
create xml schema collection S1 as @s
```

creates an XML Schema collection with name `S1` that consists of the XML Schemas contained in the SQL variable `@s`. The query

```
select C.name as SchemaColl, N.name as NSUri
from sys.xml_schema_collections C
left outer join sys.xml_schema_namespaces N
on C.xml_collection_id = N.xml_collection_id
```

for example returns a list of all XML Schema collections together with their contained namespaces.

Since the XML Schemas are stored as meta-data, the original XML form is not preserved and non-validation relevant information such as comments is lost. If preserving such information is important, the schemas can additionally be stored in an XML data type column. If a schema should be returned in XML form, a system built-in function provides the reconstructed schema.

Using a schema collection has two advantages over just relying on a single namespace URI: A database schema can contain different versions of the same URI in different schema collections (for example the different XHTML schemas) and each collection yields a closed, consistent type world for open content sections that helps in statically typing XQuery expressions.

3.2 Typing and Validating XML Data

SQL Server 2005 allows you to associate an XML Schema collection with an XML data type and to differentiate between an

XML type that has to contain a well-formed document or may contain content fragments. The additional schema constraints guarantee that any XML instance inserted into the column will be valid according to the schemas and the instance will be typed accordingly. For example, a `price` element inside the XML will be of type `xs:decimal` as described by the associated schema component instead of being untyped. The following example shows a table definition that constrains the instances in the `POContract` XML column to a well-formed document (i.e., it can only have a single top-level element) that is valid according to a `ContractSchema` schema collection in the same database:

```
create table Orders(
   id int, oderdate datetime,
   POContract XML(document ContractSchema))
```

Such schema collections can also be used to validate previously untyped XML data during execution such as in

```
select CAST(@x as XML(S1))
```

where the CAST will fail if the document is not valid according to the schemas in S1.

In addition, SQL Server 2005 supports XML Schema evolution as long as it does not require revalidation of existing data, e.g., adding new top-level schema components. Scenarios where revalidation is required are supported by creating new schema collections and explicitly reassigning and revalidating the data.

Associating XML Schema collections also provide information that can be used to optimize the physical design of the stored XML data both in its binary format and when an index is created. For example, simple typed element content will not be represented as a text node anymore, but as the typed value of the element, thus making the storage more compact and access of the typed value more efficient. Furthermore, the type information can be used to statically type and optimize XQuery expressions (see section 4).

4. QUERYING AND MANIPULATING XML

SQL Server 2005 provides the following XQuery-based capabilities on the XML data type: query transformation, atomic value extraction, existence check, node-to-row mapping capabilities, and some node-level update functionality using a data modification language that is based on XQuery (XML-DML).

4.1 Calling XQuery and XML-DML

SQL Server 2005 provides the query and modification capabilities using the method invocation syntax used also for the newly added CLR user-defined types instead of a keyword-based syntax. Each method takes a string literal as the query or update expression and potentially other arguments. The XML data type instance, on which the method is applied provides the context item for the path expressions. The in-scope schema definition context of the query will be populated with the necessary type information provided by the XML Schema collection associated with that type, or if no collection is provided, will assume that the XML is untyped. This means that you do not have to use an explicit call to a document function or variable to bind to the XML to be queried and all schema information is implicitly provided, thus removing the need for explicit schema import.

Furthermore, the SQL Server 2005 XQuery implementation is statically typed, which will provide you with early detection of errors in path expressions, type and cardinality mismatch errors, and additional optimizations. The following gives an overview of the available methods for querying XML data.

The *query method* takes an XQuery expression and returns an always untyped XML data type instance that then can be cast to a target schema collection if the data needs to be retyped. In XQuery specification-speak, the construction mode is set to strip. The following example shows a simple XQuery expression that transforms a complex `Customer` element into a summary showing the name and sales leads for `Customer` elements that have sales leads (ignoring the remainder of the `Customer` content):

```
select doc.query('
  for $c in /doc/customer
  where $c//saleslead
  return
    <customer id="{$c/@id}">{
      $c/name, $c//saleslead
    }</customer>')
from TripReports
```

The query will be executed for each row in the table `TripReports` and be applied to every `doc` XML data type instance.

The *value method* takes an XQuery expression and a SQL type name, extracts a single atomic value from the result of the XQuery expression, and casts it into the specified SQL type. If the XQuery expression results in a node, the typed value of the node will implicitly be extracted. Note that the value method performs a static type check that at most one value is returned. Since the static type of a path expression often may infer a wider static type, even though the dynamic semantics will only return a single value, we recommend using the positional predicate to retrieve at most one value. The following example shows a simple XQuery expression that counts the `order` elements in each XML data type instance and returns it as a SQL integer value:

```
select doc.value('
  count(/doc/customer/order)', 'int')
from TripReports
```

The *exist method* takes an XQuery and returns 1 if the expression results in a non-empty sequence and 0 otherwise. The following expression retrieves every row of the `TripReports` table where the document contains at least one customer with a sales lead:

```
select doc
from TripReports
where 1 =
  doc.exist('/doc/customer//saleslead')
```

So far, the expressions always map from one XML data type instance to one result value per relational row. Sometimes however, you want to split one XML instance into multiple subtrees where each subtree is in a row of its own for further relational and XQuery processing. This functionality is provided by the *nodes method* which takes an XQuery expression and generates a single-column row per node that the expression returns. Each value in the row contains an internal reference to a different node. Since the resulting type is a reference type that does not exists in SQL Server outside the context of a single query, the query methods have to be applied for dereferencing and materializing the result. These methods will be applied like on any other XML data type with the difference that the context item for the path expressions is not the document root of the XML but at the referenced node. The following example will extract a row that contains the XML representation of its customer, the name of the customer, and the order id for every customer order in the XML column:

```
select N.o.query('..') as Customer,
  N.o.value('../name[1]',
            'nvarchar(20)') as CustomerName,
  N.o.value('@id', 'int') as OrderID
from TripReports cross apply
TripReports.doc.nodes('/doc/customer/order')
as N(o)
```

Note that the nodes method is similar to the OpenXML functionality in that it can be used to shred XML into relational form, but its expression is integrated into XQuery processing.

The value method above provides a way to promote an XML value into the SQL value space. Sometimes, one wants to access relational data in the context of XQuery instead. For that case, SQL Server 2005 has added two special XQuery functions called `sql:variable()` and `sql:column()` that take constant string literals to refer to a SQL variable or a correlated column.

Finally, the *modify method* provides a mechanism to change an XML value at the subtree level. SQL Server 2005 provides for inserting new subtrees at specific locations inside a tree, changing the value of an element or an attribute, and deleting subtrees. The following example deletes all customer `saleslead` elements of years previous to the year given by a SQL variable or parameter with the name `@yr`:

```
update TripReports
set doc.modify('delete  /doc/customer
   //saleslead[@year< sql:variable("@yr")]')
```

4.2 Execution of XQuery Expressions

How does the database system execute these XQuery expressions? As mentioned earlier, the XML data is stored in an internal binary representation. However, in order to execute the XQuery expressions, the XML data type will internally be transformed into a so-called node table format. The internal node table basically uses a row to represent a node. Each node receives an OrdPath as its node id, enough key information to point back to the original row in the base table, information about the node name and its type (in a tokenized form), node value, an inverted path (the Path_ID) from a node to the document root, and more. Figures 2 and 3 show an example of how to number an XML document using an OrdPath (based on [33]):

```
<BOOK ISBN="1-55860-438-3">
    <SECTION>
       <TITLE>Bad Bugs</TITLE>
          Nobody loves bad bugs.
    </SECTION>
    <SECTION>
       <TITLE> Tree Frogs </TITLE>
          All right-thinking people
          <BOLD>love</BOLD> tree frogs.
    </SECTION>
 </BOOK>
```

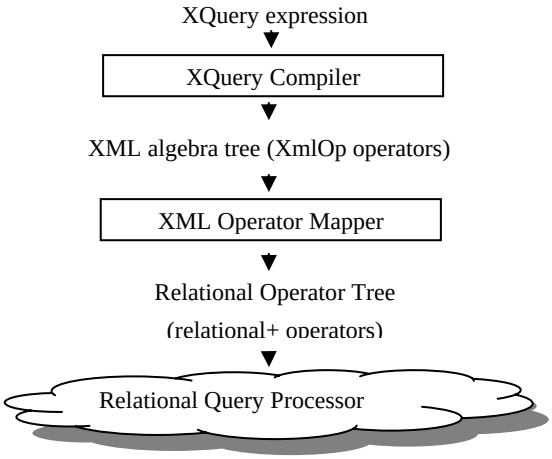
Figure 2: Sample XML data

The OrdPath encodes both the document order and the hierarchy information. By using an intermediate careting number between the siblings, insertions can occur without affecting the neighbor numbering. For example, inserting a new element before the `SECTION` element will create a new node with the OrdPath 1.2.1. Retrieval of a subtree or a parent node can easily be achieved by a single range scan using the OrdPath prefix on the node table, thus avoiding the self-joins normally associated with subtree retrieval. For more information about the numbering scheme see [33].

ORDPATH	TAG	NODE TYPE	VALUE
1.	1 (BOOK)	1 (Element)	null
1.1	2 (ISBN)	2 (Attribute)	'1-55860-438-3'
1.3	3 (SECTION)	1 (Element)	null
1.3.1	4 (TITLE)	1 (Element)	'Bad Bugs'
1.3.3	--	4 (Text)	'Nobody loves bad bugs.'
1.5	3 (SECTION)	1 (Element)	null
1.5.1	4 (TITLE)	1 (Element)	'Tree frogs'
1.5.3	--	4 (Text)	'All right-thinking people'
1.5.5	7 (BOLD)	1 (Element)	'love'
1.5.7	--	4 (Text)	'tree frogs'

Figure 3: XML data of Figure 2 in simplified node table

All XQuery and update expressions are then translated into an XML algebra tree which in turn is then translated into an extended relational operator tree against this internal node table that uses the common relational operators and some operators specifically designed for XQuery (see Figure 4). The resulting tree is then grafted into the operator tree of the relational expression so that in the end, the query execution engine will receive a single operator tree that it will optimize and execute (see



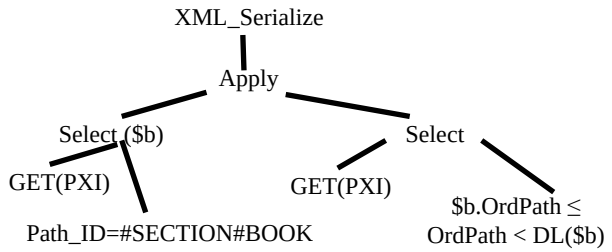
[34]).

Figure 4: Architecture for XQuery compilation.

For example, the XQuery algebra provides an `XmlOp_Path` operator that provides the algebra operation for path expressions. Depending on the path expressions and the presence or absence of indices (see section 4.3) it is mapped to different operator trees. Figure 5 shows a relational operator tree for the expression `doc.query('/BOOK/SECTION')`. Assuming that we have an exact path expression (no wildcards) and a primary XML index (see section 4.3) is present, then `XmlOp_Path` is mapped to a relational SELECT operator that filters primary XML index rows (GET(PXI)) by matching the supplied path `/BOOK/SECTION` with the value in the Path_ID column. The Path_ID column stores the reversed path in an encoded form. The XML Operator Mapper applies a Path_ID generation function over the path to generate the search value for the Path_ID column (depicted as #SECTION#BOOK). Since the result is returned as an XML data type instance, the XML_Serialize operator receives rows corresponding to the subtree of each `SECTION` and produces the XML result. The APPLY operator in the relational operator tree is a correlated join between the `SECTION` rows and the right child of the APPLY operator.

The retrieval of each of the `SECTION` subtrees (right-side SELECT) selects the nodes belonging to the subtree having the OrdPath value in between those of the `SECTION` node and its

descendant limit (DL). This is executed efficiently using a range



scan over the primary XML index.

Figure 5: Relational operator tree XML Indices

4.3 XML Indices

In order to avoid costly runtime transformations from the binary XML into the node table, a user can prematerialize the node table using the primary XML index:

```
create primary xml index xIdx on XDoc(doc)
```

Additionally, SQL Server 2005 provides three secondary XML indices of which the query execution can take further advantage. The PATH index supports simple types of path expressions (it indexes the Path_ID column), the PROPERTY index provides support for the common scenario of property-value comparisons and the VALUE index helps if the query uses wild-cards in comparisons. For more information about the XML Indices see [35].

4.4 Optimizations

The operator tree will be optimized using the general database optimizer that will make cost based decisions on how to execute the operator tree and how to best utilize the available indices. Additional optimizations can already be done when generating the XML algebra tree and when mapping the XML operators. For example, the static typing rules can be applied to avoid runtime type checks and type inferences can avoid runtime type casts. When multiple path expressions are being used, the mapping of the `XmlOp_Path` operators can collapse paths that are close for more efficient path retrievals. [34] provides more information about the query compilation and optimization.

5. XML Publishing

SQL Server 2000 introduced the `FOR XML` clause for transforming relational rowsets into a variety of XML shapes. SQL Server 2005 extends this functionality by taking advantage of the XML data type and provides efficient, simple and intuitive mechanisms to generate XML results [36]. The following query shows the new `FOR XML PATH` mode producing a `Doc` element that contains customers with their contact information and their orders and employees working on their orders (based on the Northwind example database):

```
select CustomerID as "@CustomerID",
  CompanyName,
  City       as "address/city",
  PostalCode as "address/zip",
  ContactName  as "contact/name",
  Phone        as "contact/phone",
  (select OrderID as "@OrderID"
   from Orders
   where Orders.CustomerID = Customers.CustomerID
   for xml path('Order'), type),
  (select distinct LastName as "@LastName"
   from Employees join Orders
   on Orders.EmployeeID = Employees.EmployeeID
   where Customers.CustomerID = Orders.CustomerID
   for xml path('Employee'), type) as Employees
from Customers
for xml path('Customer'), root('Doc')
```

Namespaces can be associated using the SQL-2003 standard `WITH XMLNAMESPACES` clause.

6. Conclusion and Outlook

This part of the tutorial has shown how SQL Server 2005 extends its relational database engine to support XML and XQuery in a seamlessly integrated way. First customer deployments and performance and scalability investigations show that the chosen functionality and architectures provide lots of benefits over manual shredding and main-memory programming approaches.

While the current implementation only supports a subset of XQuery (mainly due to the fact that the standard will be finalized after SQL Server 2005 releases) and only operates on a single XML data type instance per query, the presented architecture will have no problem supporting the full language over a set of XML documents or a variety of input documents. Future releases will extend the presented support based on customer feedback and by taking the next versions of the SQL/XML standard into account depending on customer benefits. Some interesting areas are node-level concurrency control [38], additional indexing options and top-level XQuery [37].

7. ACKNOWLEDGMENTS

The author would like to thank all his colleagues at Microsoft for their contributions to the XML support in SQL Server 2000, 2005 and beyond. Special thanks to Shankar Pal who took the time to review the presented material.

8. REFERENCES

References below [30] refer to references in the tutorial introduction.

[30] M. Rys: *Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems* in 7th ICDE, pages 465-472, Heidelberg 2001.

[31] http://msdn.microsoft.com/sqlxml

[32] http://msdn.microsoft.com/XML/BuildingXML/XMLandDatabase/default.aspx

[33] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury. *ORDPATHs: Insert-Friendly XML Node Labels.* SIGMOD 2004.

[34] S. Pal, I. Cseri, O. Seeliger, M. Rys et al.: *XQuery Implementation in a Relational Database System*. Submitted for publication.

[35] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, V. Zolotov. *Indexing XML Data Stored in a Relational Database*. In *Proceedings of VLDB Conference*, Toronto, 2004.

[36] M. Rys. What's New in FOR XML in Microsoft SQL Server 2005 at http://msdn.microsoft.com/library/en-us/dnsql90/html/forxml2k5.asp

[37] M. Rys. *XQuery and Relational Database Systems.* In *XQuery from the Experts*, Howard Katz (ed.), Addison-Wesley, 2003.

[38] M. Haustein and T. Haerder. Adjustable Transaction Isolation in XML Database Management Systems. In XSym 2004, Springer LNCS 3186, 2004.

Inside MicrosoftÂ® SQL Server(TM) 2005: Query Tuning and Optimization (Developer Reference) by Kalen Delaney Paperback $53.76. Only 1 left in stock - order soon. Ships from and sold by PHILLY - MART.Â The Guru's Guide to SQL Server Stored Procedures, XML, and HTML. Ken Henderson. 4.0 out of 5 stars 71. Paperback. $39.80. The Guru's Guide to SQL Server Architecture and Internals. Ken Henderson. 3.7 out of 5 stars 24.Â Kalen Delaney, a Microsoft MVP for SQL Server since 1993, provides advanced SQL Server training to clients worldwide. She is a contributing editor and columnist for SQL Server Magazine and the author of several highly regarded books, including Microsoft SQL Server 2008 Internals. Read more. Product details. MicrosoftÂ® SQL Serverâ„¢ 2005 Programming For DummiesÂ® Published by Wiley Publishing, Inc. 111 River Street Hoboken, NJ 07030-5774. www.wiley.com. Copyright Â© 2007 by Wiley Publishing, Inc., Indianapolis, Indiana.Â You discover how to create an assembly to run on the Common Language Runtime inside the SQL Server 2005 database engine. Introduction. 5.