Optimizing Runtime XML Processing in Relational Databases

Eugene Kogan, Gideon Schaller, Michael Rys,
Hanh Huynh Huu, and Babu Krishnaswamy

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
{ekogan,gideons,mrys,hanh,babuk}@microsoft.com

Abstract. XML processing performance in database systems depends on static optimizations such as XML query rewrites, cost-based optimizations such as choosing appropriate XML indices, and the efficiency of runtime tasks like XML parsing and serialization. This paper discusses some of the runtime performance aspects of XML processing in relational database systems using Microsoft® SQL Server™ 2005's approach as an example. It also motivates a non-textual storage as the preferred choice for storing XML natively. A performance evaluation of these techniques shows XML query performance improvements of up to 6 times.

1 Introduction

Most relational database management systems are in the process of adding native XML support [13], largely motivated by the necessity of querying and modifying parts of XML documents that cannot easily be mapped to the relational model. The native XML support consists mainly of a native XML data type – based on the SQL-2003 and upcoming SQL-200n standards, XML Schema and XQuery [3] support. Interestingly, all systems take a similar functional approach [13], and, even though some of their underlying physical approaches differ with respect to the various XML storage and indexing techniques, they have similar trade-offs to make in order to efficiently store, retrieve, validate, query and modify their XML.

While XML is a scalar data type for the relational type system, it has to be transformed into the XQuery data model [4] for XQuery evaluation, and the results of XQuery have to be serialized back into scalar form. This involves XML parsing, possible XML validation, and XML generation. XML parsing and validation are considered expensive in database environments [9].

Based on Microsoft SQL Server 2005's implementation, we will present some current state of the art runtime optimization approaches that are used by relational database management systems to address these issues.

Microsoft SQL Server 2005 provides the native XML data type which optionally can be constrained according to a collection of XML schemas and queried using XQuery. Additionally, a modification language based on XQuery can be used to update the XML in place. For more information, see [1] and [7].

S. Bressan et al. (Eds.): XSym 2005, LNCS 3671, pp. 222–236, 2005.

SQL Server internally represents the XQuery data model as a row set of XML "information items". During XQuery evaluation this row set is processed by a set of extended relational operators. Operators that transform an XML scalar into the XML info item row set and aggregate the row set into an XML scalar are two major XML-specific extended operators. The former is the more expensive operator since it has to perform XML parsing as well as typing XML info items when processing typed XML.

SQL Server 2005 provides the option to store the XML info item row set pre-shredded to speed-up XQuery processing. This pre-shredded form is called the primary XML index [2]. The XML index allows the relational optimizer to exploit the full power of cost based optimizations when building XML query execution plans.

However, there are multiple important scenarios when an XML instance can't be indexed or the XML indexing cost is prohibitive. In such cases the cost of the XML query evaluation is dominated by the cost of XML parsing and producing the type information for the XML info item row set.

This paper will provide details on the performance optimizations of these runtime operations. The optimizations are in the areas of tuning the serialization format used for the scalar XML storage and integrating XML processing APIs into the database infrastructure. In Section 2 we describe the XML support in Microsoft SQL Server 2005 from the runtime point of view. The information in Section 2 is presented indifferent to the employed XML storage format. Section 3 analyzes the inefficiencies of XML processing with traditional APIs and text XML as the storage format. Section 4 describes binary XML storage format properties as well as some of the solutions and techniques we employed in order to maximize the performance of runtime XML processing. It also shows how they address the performance bottlenecks we identified in Section 3. Section 5 provides the results of some performance evaluations based on XMark [8] and XMach-1 [6] workloads. Related work is discussed in Section 6. Section 7 contains concluding remarks and potential future work.

2 XML Support in Microsoft SQL Server 2005

In this section we briefly describe the main XML features in Microsoft SQL Server 2005 and highlight their execution phase. The XML support in SQL Server 2005 is described in more detail in [1], [2], [7], and [14].

XML is a new data type natively supported in Microsoft SQL Server 2005. It is supported as a variable type, a column type, and a parameter type for functions and stored procedures:

```
DECLARE @xvar XML
CREATE TABLE t(pk INT PRIMARY KEY, xcol XML(xsc))
CREATE PROCEDURE xproc @xparam XML(xsc)
AS SELECT xcol FROM t
```

The *XML(xsc)* syntax indicates that an XML schema collection *xsc* is associated with the type which validates and types the XML data. We refer to schema constrained XML data as "*typed XML*".

SQL Server guarantees that data stored in XML type is a well formed XML fragment or document and provides *XML fidelity* [13]. Additionally, a typed XML instance is guaranteed to validate against its XML schema collection. To support this,

the server parses the XML data when it arrives in the database and during data conversions, and checks that it is well formed; typed XML is also validated at that point. Once parsed and validated, the server can write the XML into the target storage unit in a form optimized for future parsing. This format is the subject of discussion in Section 3 and 4. Physically, XML is treated as a large scalar data type and – just like any other large data types (LOB) in SQL Server, – has a 2GB storage limit.

2.1 Querying XML

SQL Server 2005 adds XQuery through methods on the XML type and applies it to a single XML type instance (in this release). There are 4 methods on the XML type that accept XQuery string and map the resulting sequence into different SQL types:

- the *query* method returns XML;
- the *value* method returns SQL typed values other than XML;
- the *exist* method is an empty sequence test;
- the *nodes* method returns a table of references to XML nodes;

Additionally, the *modify* method uses the Microsoft data modification extensions to XQuery in order to modify an XML instance. The *modify* method is not described is this paper. However, it is worth mentioning that its implementation in SQL Server 2005 provides the ability to perform partial updates of the LOB containing the scalar XML data, i.e. a minimally necessary part of XML LOB is modified and logged during updates with *modify()* method instead of changing the whole document.

The following example extracts the ISBN and constructs a list of AuthorName XML elements for every book in the XML variable @x:

```
SELECT
  xml_ref.value('@ISBN','NVARCHAR(20)') isbn,
  xml_ref.query('for $a in Authors/Author return
                   <AuthorName>
                     {data(Name/First), data(Name/Last)}
                   </AuthorName>') author_name_list
FROM @x.nodes('/Book') AS tbl(xml_ref)
```

Each individual XQuery expression is compiled into an XML algebra operator tree. At this stage SQL Server performs annotations of the XML operator tree with type information, XQuery static typing, and various rule-based optimizations like XML tree simplifications and XPath collapsing. Next, the XML operator tree is mapped into an extended relational operator tree and grafted into the containing SQL relational operator tree. Finally, the relational optimizer performs cost-based optimizations on the resulting single extended relational operator tree. For more details on mapping XQuery to relational operators see [14].

The extended relational operators, that evaluate XQuery, process row sets where each row represents an XML node with type information, i.e. a typed XML information item or *XML info item*. The XML hierarchy and document order are captured in this row set with the use of a special key called ORDPATH [2], [5]. The ORDPATH allows the efficient calculation of the relationship between two XML nodes in the XML tree. Some of the important XML-specific extensions of to the relational operators are those that shred XML scalars into XML info item row sets and aggregate the row sets into XML scalars: the XML Reader and the XML Serializer.

The XML Serializer operator serializes the info item row set into an XML scalar. XML well-formedness is checked by the XML Serializer and it throws dynamic errors in cases such as serialization of a top level attribute or occurrences of duplicate attributes in an element.

The XML Reader operator is a table-valued function that parses an XML scalar and produces an XML info item row set generating the ORDPATH keys in the process. On a typed XML instance, the XML Reader also has to generate typed values and provide the XML info item type information. Note that while parsing XML from XML Reader operator it is not necessary to check if the input XML is well formed or valid to its schema collection since well-formedness and validity are enforced when instantiating and modifying the XML scalar.

Except for trivial XQuery expressions, an XQuery compilation normally results in multiple XML Reader operators in the query plan. Thus, the XML instance can be shredded multiple times during the XQuery evaluation. This makes the XML Reader performance critical for overall XQuery performance when the XML is not indexed.

2.2 Indexing XML

Indexing XML [2] is the main option to increase XQuery performance in SQL Server 2005. Users can create a primary XML index and optional secondary XML indexes. A primary XML index is a materialized row set of XML info items with XML names tokenized. It contains the primary key of the base table to allow back joins. This allows query optimizer to use the primary XML Index instead of the XML Reader in the query plan.

One of the main performance benefits from the XML index comes from the ability to create secondary XML indexes to speed up XPath and value look-ups during query execution. Secondary XML indexes allow the query optimizer to make cost-based decisions about creating *bottom-up* plans where a secondary index seek or scan is joined back to the primary XML index which is joined back to the base table containing XML columns.

The XML Reader operator populates the primary XML index in XML insertion and modification query plans. Thus, performance of the XML Reader is important for the indexed XML column modification performance. It is especially visible since the XML index modification is done on top of the XML Reader as a partial update.

Even though the XML index is the best tool for boosting XQuery performance, it is an overhead on storage space and on data modification performance. There are XQuery scenarios where creating the XML index may not be practical or beneficial:

- In scenarios where XML is used as the transport format for complex data. XML is often used as the variable or parameter type. XQuery is then applied in order to extract the values from the XML instance. XML indexes cannot be created on variables and parameters. The goal is to have a big range of similar queries performing well enough so that no index needs to be created.
- When XML instances in a table are predominantly located using a table scan or a non-XML index, such as an index on a relational column or full text index, the benefits of XML indexes are limited. Additionally, if XML instances are small or the XQuery expressions are simple and access a large number of nodes in the XML instances, XQuery on the XML blob should yield the result equally fast.

If XML instances are queried in such scenarios, the cost of XML index creation and maintenance could be avoided, making performance of parsing the XML scalar into the XML info item row set important.

3 Performance Challenges of Textual XML Storage Format

Here and in Section 4 we evaluate textual and binary XML storage format options based on the following goals for our runtime performance evaluation:

- to make XML parsing as fast as possible since many XML processing scenarios depend on that;
- to make XML parsing scalable with respect to the size of XML, different ratios of XML mark-up vs. text data, the number of attributes for an element, etc.
- to make the performance of XQuery on typed XML column without XML index comparable with XQuery on untyped XML column without XML index;
- to make XML serialization/generation scale and perform well.

Between the performance and scalability of XML parsing and serialization, XML parsing clearly commands higher priority because of its importance for a bigger number of XML processing scenarios. Note that XML scalability in a database server also means that XML processing should require only limited amount of memory in order to not hurt overall server performance and throughput.

The naïve choice for native XML storage format is to store its original textual form. Advantages are the ability to use a standard parser and a standard generator for all XML parsing and generation needs, and the ability to send XML content with no conversion to any client.

SQL Server 2005 leverages the fastest native-code text XML parser available from Microsoft. It is a pull model XML parser that provides a light weight COM-like interface. SQL Server uses it to build an XML Validating Reader that can produce typed XML info set items if the XML data is associated with a schema collection. It is optimized for performance and scalability and has deterministic finite state automata objects for typed XML elements cached in a global server cache.

In the remainder of this section we will discuss performance challenges and bottlenecks that result from the textual XML storage choice. Section 4, will present our solutions to the outlined problems. We will also use the textual XML storage as the base line for our performance evaluation in Section 5.

3.1 XML Parsing and Generation Performance Bottlenecks

Here we enumerate most of the CPU-intensive operations during XML parsing using traditional text XML parsers as well as operations that may require large amounts of memory:

- XML character validity checks and attribute uniqueness checks that are done as part of the XML well-formedness check may require large amounts of memory to store unbounded number of attribute names as well as add to the CPU load. Since XML well-formedness is guaranteed for the XML type, such checks should only

be necessary when converting other types to XML or during operations that generate an XML type and not during the conversion to the XML info item row set.

- The XML parser has internally to keep all in-scope XML prefix mappings at any point during the document processing. In the general case, the XML parser either needs to buffer in memory all attributes and XML namespace declarations for an element or, if the input stream is seekable (as it mostly is in server scenarios), it has to do two passes through the attributes in order to resolve all namespace prefixes to namespace URIs before returning the element node.
- Entity resolution, attribute whitespace normalization, and new line normalization in text nodes may require substantial CPU and memory resources depending on the implementation approach. What's important for our investigation is that if an attribute value or text node needs to be accessed by the caller of the XML parser as a single value then such a value needs to be copied and buffered either in memory or on disk if it is large.
- Many of the above listed operations involve memory allocations and de-allocations. Memory management during XML parsing is one of the most expensive operations in terms of CPU utilization. Besides, larger memory consumption decreases query throughput.
- Textual XML verboseness adds to CPU utilization when checking well-formedness and I/O overhead when reading and writing the XML to and from disk.
- XML can be provided in many different language encodings that in the XQuery data model are all mapped to Unicode. While encoding translation is one of most expensive operations during XML parsing, it can be dealt with by storing the XML in the target Unicode encoding (UTF-16 for SQL Server), so that the translation has to occur only once.
- Document Type Definition (DTD) processing can be very expensive both in terms of CPU utilization and memory consumption. We are not going into details of DTD processing here since it has very limited support in SQL Server 2005 – only internal entities and default attributes are supported and only with an explicit conversion flag. DTD is always stripped during population of XML type instances.

In the case of XML serialization/generation the most expensive operations are well-formedness check and entitization. Also, having an API with XML attributes as second class objects require buffering attributes during element construction which may not scale well in database server scenarios.

3.2 Typed XML Processing Overhead

When using text XML as the XML type storage format the XML Reader operator uses the Validating Reader for producing the typed XML info item row set. We expect the XML instance to be valid according to its schema collection, but we still have to go trough the validation in order to annotate the info items with type information as well as provide typed values.

The Validating Reader introduces performance overhead on top of the basic XML parser when evaluating XQuery on a typed XML. This overhead is normally greater than the XML parsing overhead. For example, converting a 4.5MB Microsoft Word

2003 document in XML format to XML typed with a fairly simple WordML schema (available for download through http://www.microsoft.com/downloads/details.aspx?FamilyId=FE118952-3547-420A-A412-00A2662442D9&displaylang=en) takes 2.2 times longer than converting the same document to untyped XML. More complex schema validation that also involves more type conversions can introduce bigger overhead on shredding typed XML. We didn't invest into performance evaluation of various XML schemas concentrating instead on a solution that would minimize the number of XML validations required.

The XQuery evaluation in SQL Server 2005 requires that the XML Readers flow rows representing the simple type or the simple content element with their typed value "pivoted" in the same row. Since the XML parser and the Validating Reader return XML nodes in XML document order such "pivoting" in turn requires the buffering of attributes, comments, and processing instructions preceding the simple type/content element value text node. The buffering is done in an in-memory buffer that is spilled to disk if the amount of data buffered goes above a threshold. Note that the "pivoting" is required for shredding typed XML for both XQuery evaluation on the fly and for the XML index.

4 Using Optimized Representations to Store XML

In the previous section, we assumed that XML type instances are stored in their textual format. The idea of all of the runtime performance optimization we describe below is to do all the processing-expensive steps of XML parsing and validation only once – when populating the XML scalar. The XML scalar representation should then use an optimized representation that – together with a custom parser of that representation – will allow us to provide a more efficient XML Reader operator that can avoid many of the steps that traditional XML parsers go through.

After analyzing the textual XML runtime performance bottlenecks we also want to avoid buffering large or unbounded amounts of data in memory and have minimal data copying by implementing XML shredding and generation in a streaming way. Based on that, we perform the following three runtime optimizations:

- choosing a binary XML format as the storage format for XML type,
- revising the XML parsing and generation interfaces,
- introducing techniques to avoid copying larger values returned from the XML parser.

Below we describe each of the solutions and how they addressed the performance bottlenecks we described in Section 3. These solutions take advantage of the fact that the LOB storage in SQL Server 2005 is optimized for random access so the XML parser and generator operate on seekable streams with efficient seeks when working with both on-disk and in-memory XML LOBs.

4.1 Binary XML as XML Type Serialization Format

For the XML type representation SQL Server uses a binary XML format that preserves all the XQuery data model [4] properties. The binary XML format has to be self-contained so it can be sent to the clients that support the format without any pre-

processing, i.e. the format should not rely on the server metadata like the XML schema or the XML name token dictionaries. Also, the requirement to have efficient partial XML BLOB updates limits XML structure annotations that could be exploited in the format. Finding an optimized layout of the XML BLOB on-disk data pages (like it is done in Natix [12] and System RX [17]) that would require additional processing upon retrieval is not a goal - XML index is an option for optimal XQuery execution performance, and support for XML navigation APIs like Document Object Model (DOM) is also not required.

Describing the Binary XML storage format of SQL Server 2005 is beyond the scope of this paper. Instead, we will focus on the properties of the format that allow building a fully streamable binary XML parser and how they address the performance bottlenecks.

- XML attribute values and text nodes can be stored with typed values in the SQL Server 2005 binary XML representation. Primitive XML schema types are supported as well as most of SQL types. This allows avoiding performing type conversions when serializing the typed XML info item row set into a XML BLOB, and also allows skipping data conversions when shredding XML BLOB into a XML info item row set as well as when validating XML instances after modification.

  Note that typed values do not necessary require less space than their string serialization; for example, the UTF-16 string "1" takes less space than the 8-byte integer 1. Also note that the binary XML format does not generally preserve string value formatting – typed values are serialized in their XML Schema canonical form when binary XML is converted to text XML.
- XML name strings as well as values of string types are stored in the target API code page – UTF-16, – so no code page conversion is required when generating or parsing the binary XML representation.
- All variable length values are prefixed with their length so that the binary XML parser can seek over such value when parsing XML from a seekable stream allowing the caller to access such value only if needed. We'll explain the idea in more detail in Section 4.3.
- All XML qualified names are represented as sets of local name, namespace URI, and namespace prefix and stored tokenized. Tokens can be declared anywhere in binary XML BLOB before the token is first used. This allows streaming binary XML generation.

  Tokenization of the XML QNames allows the binary XML parser to avoid supporting a XML namespace prefix resolution interface since namespace URI is returned with every element or attribute info item without pre-scanning all attributes of an element and without keeping the list of visible XML namespaces in memory. Note that typed values derived from the QName type are stored tokenized as well. Supporting namespace prefix resolution is still required for cases where the XML QNames are not stored typed such as in the case of XPath expressions stored as strings in XSLT.

  QName tokenization can lead to a significant space compression for XML documents where the same QNames are used repeatedly.
- The binary XML format supports format-specific extensions that are not part of the XML content and only visible to binary XML parsers. Like XML processing

instructions such extensions can be ignored by a XML parser that does not process those. They are ignored when the XML type is converted to string.

These format-specific extensions are used in the server space for two reasons.

1. Add element/attribute integer type identifiers needed for info item type annotations in XML info item row set. Together with supporting typed values this allows shredding the typed XML into XML info item row set without the validation step.
2. Annotate simple type/content elements with offsets to their typed values in the binary XML BLOB so the expensive typed value "pivoting" described in Section 3.2 can be done with two seeks in binary XML stream instead of buffering the attributes, comments, and processing instructions preceding the element value. Note that for simple content elements the trick with the offset makes partial update of a typed XML BLOB more expensive if an attribute, comment, or processing instruction of such element is modified – the offset of the value may need to be adjusted.

When generating binary XML, character entitization (such as transforming < into <) and string conversions of typed XML values become unnecessary. However, QName tokenization may add to CPU load during binary XML generation to the degree that that binary XML generation may become more expensive than the equivalent text XML generation. This can be mitigated by caching QName tokens in cases where the set of QNames is known statically, like when formatting a row set as XML or serializing typed XML (the latter case is not currently optimized in SQL Server 2005).

The QName tokenization performance overhead of generating the binary XML representation is outweighed by performance benefits for binary XML parsing. The binary XML format with the above properties allows the parser to be fully streamable and free from all the XML parsing bottlenecks we listed in Section 3.1.

However, it introduces the need to maintain an in-memory QName token look-up array in order to resolve tokens to XML QNames during the parsing process. This array can require a large amount of memory. To work around this issue, the binary XML format requires XML generators to insert a special binary token into the binary XML stream that signals to the readers that the QName token look-up array should be freed and new tokens are declared for QNames used afterwards. This command for flushing the token array must be inserted when the total size of all QName string or total number of defined QNames reach some threshold. This way there's always a preset limit on the memory consumption of the QName look-up array.

Note that SQL Server 2005's binary XML format may be used in other areas than the SQL Server Engine. Even though we don't document the binary XML format in more detail in this paper, the properties of the format listed above are all the important binary XML features the server takes advantage of – XML QName tokenization, typed values, prefixing variable length types with data length, element/attribute type annotations.

4.2 XML Parsing and XML Generation API Improvements

To take advantage of the properties of the binary XML format we need to revise the pull model XML parser and generator APIs. The API improvements include:

- added typed value support instead of chunked character data; no value chunking is supported for typed values; it does not result in the intermediate value buffering – more details in section 4.3;
- added info item type identifier (specific to the server); the generator and the parser add/retrieve it through binary XML extension tokens;
- stopped supporting XML namespace prefix resolution interface unless specifically requested by the caller for custom QName resolution;
- attributes are returned/accepted as first class XML items, i.e. instead of returning list of attributes with the element event there's a new attribute info item event; there's no need in buffering all attributes for a given element in order to support multiple iterations through the attribute array;
- for all QNames, including values of types derived from QName type, the improved API supports passing those as a triplet of local name, namespace URI, and namespace prefix; the binary XML generator also allows callers to access tokens assigned to QNames and then pass only a QName token if the QName is written multiple times.

Taken together with the binary XML format properties, these API improvements allowed creating a fully streaming binary XML parser and generator. Note one exception to the streaming behavior when generating typed XML and when parsing typed XML for XQuery evaluation or for XML index: seeks are required in order to annotate simple type/content elements with offsets of their values, and when retrieving the value with the element info item.

4.3 Dealing with Large Values

With text XML parsing either the parser or the calling code have to aggregate chunks of single value into a single scalar value. This is required because of entity resolution and various value normalizations that the text XML parser has to do. Such value aggregation from multiple chunks can be expensive since in database server scenarios it has to be disk-backed in order to avoid allocating large amounts of memory.

The binary XML format contains all values in the form returned by the XML parsing API. Specifically, the binary XML format has value lengths implicitly (through value type) or explicitly present in the binary XML stream before the value. Since LOBs in SQL Server are passed as references, we can refer to a fragment of a binary XML BLOB without requiring a value copy.

An object called a Blob Handle is used when LOB is transported within the database server boundary. A Blob Handle is a universal reference to a large object within the server. It can contain a reference to on disk storage containing LOB data, or inlined LOB data for smaller data sizes, or a reference to LOB data dynamically constructed in memory. The usage of Blob Handles ensures that large amounts of data are not moved unnecessarily when LOBs are passed as parameters or flowed in the query processor during query execution.

A new class of Blob Handles is introduced that can refer to a fragment in another LOB and use such Fragment Blob Handles when the source LOB is not mutable. Fragment Blob Handles are used by the binary XML parser for returning string and binary values larger than a preset copy threshold. The implementation of the Frag-

ment Blob Handle allows retrieving the value from the parser input buffer if it is still there (when the value is retrieved from the same thread as the parser is on).

This technique allows the binary XML parser to skip over large string or binary values when working on a seekable stream. If further XQuery processing filters out XML nodes with large values then the disk pages containing these large values do not have to be fetched. This improves performance and scalability of XQuery on XML instances with large string/binary values as well as XML with high data to mark-up ratio.

The above described techniques lowered memory management expenses and allowed building binary XML parser and generator that do not allocate memory after construction except for adding entries to the QName token look-up table. We took advantage of the fact that the QName-token mappings are allocated one by one and freed as a whole and used an incremental memory allocator based on a buffer chain instead of using a more expensive heap allocator.

5 Performance Evaluation

This section reports the SQL Server 2005 runtime performance improvements measured on a set of XML query benchmarks and some additional tests. We evaluated the performance of XQuery execution in the database server environment and not the performance of XML parsing and generation in isolation. All the tests were run with a warm database cache in single user mode so the results reflect mostly CPU cost of the query where XML BLOB processing, in turn, takes the most part. For our experiments we instrumented the sever code so we are able to switch between text and binary XML as the XML type storage format and parser type. The text XML parser was wrapped into a class that implements the new binary XML parser API. Textual XML was stored after it went through text-to-text XML conversion while instantiated so it was in UTF-16 format with any DTD stripped, and all values normalized and re-serialized with minimally necessary entitization. So, during text XML parsing from XML Reader operator parts of the most expensive operations were not performed.

The tests were run on a 4-way Intel Xeon 1.5GHz machine with 2GB RAM. The performance differences are large enough to fall into the realm of statistical significance.

5.1 XMark Benchmark for Untyped and Typed XML

We tested our XML storage and runtime improvements on the XMark test suite. XMark [8] is an XQuery benchmark that models an online auction and represents data-centric XML scenario. We chose an XMark scaling factor 0.5 adopted for relational database use so that one large XML instance was shredded into smaller XML instances put into 5 tables representing the data model entities. [2] gives more information on the XMark modifications for SQL Server 2005. We did two runs – one for an XMark database where the XML columns are untyped and one for an XMark database containing XML columns typed according to the XMark schema. The results are presented in Table 1.

Table 1. Performance gain on XMark scale factor 0.5 comparing untyped and typed Binary XML storage format against text format

	Avg gain	Gain Range	Avg Binary XML compression rate
Untyped	2.0	1.53 – 2.48	1.10
Typed	4.7	2.65 – 6.40	1.02

We didn't provide performance gains for individual queries since the results were distributed rather evenly around the avarage.

The way one big XML instance is shredded into many small ones in our XMark database is less benefitial for binary XML compression rate since XML QNames are practically not repeated in XML instances. Typed XMark database got practically no compression since the binary XML included type annotations. Therefore, we can consider the XMark test suite as relatively less beneficial for our runtime XML improvements.

The larger performance gains when running the queries against typed binary XML format can be attributed to skipping XML validation during XML info item row set generation by XML Reader operator. Compared to the untyped XML the typed XML performed 13% slower on average. This can be attributed to the overhead of parsing the type annotations.

5.2 XMach-1 Benchmark and Additional Measurements

We also ran our performance comparison on a more document-centric workload like XMach-1 [6]. We ran 5 out of 8 data retrieval queries – queries 3, 4, 6, 7, and 8, – that were ported to the SQL Server 2005 XQuery implementation.

The queries were run on a set of XML instances totaling 30.8MB in UTF-16 text XML. In binary format the total size of the XML instances was 26.2MB – showing a compression rate of 1.18. The average performance gain was 3.05 with gains of individual queries ranging from 2.77 to 3.31.

We measured raw parsing performance on a basic 2.6GHz Intel Pentium 4 machine with 1GB RAM in a single user load. We formatted a 20000 row table containing customer data as XML in an element-centric manner. This 18MB UTF-16 XML instance with highly repeated XML tags yielded a 2.4 compression rate when stored as binary XML. We used XQuery *count(/*/*)* where practically all the time is spent in parsing the XML. The query on the binary XML performed 2.77 times faster than the same query on the text XML LOB.

The same table formatted in attribute-centric manner resulted in 12.8MB UTF-16 text XML with a 1.77 compression rate when converted to binary XML (binary XML representations of the element-centric and the attribute-centric formatting were nearly of the same size – 7.43MB and 7.23MB). Similar XQuery counting attributes in the document (*count(/*/@*)*) performed 4.17 times faster on the binary XML than on the text XML. The reason of the higher performance gain is that the query on the binary XML performed 1.62 times faster after switching from element-centric to attribute-centric formatting while the performance gain for the text XML was only 1.08. That shows that with all the optimization we described binary parsing cost in this case is largely dominated by the number of calls into the parser, i.e. by the number of info

items returned; note that one attribute event when parsing of the attribute-centric formatting corresponds to 3 parser events in case of parsing the element-centric formatting. The text parsing, while also returning fewer info items, had to perform all the expensive attribute list processing we mentioned in 3.1.

6 Related Work

In [9] Nicola et al. analyze cost of SAX parsing and XML schema validation in database scenarios. While their findings are generally in line with our analysis made in this paper we analyzed a pull model XML parser and went into greater detail about its cost. We took the analysis further as a motivation for changes needed in XML storage format, APIs, and integration with the rest of database system in order to decrease cost of XML parsing. We also reported results of a real world commercial implementation of the improvements.

There's a number of publications on binary XML format and compression techniques including [15], [16], plus materials of the W3C Workshop on Binary Interchange of XML Information Item Sets [11]. While being excellent source of ideas none of them enumerates binary XML features that are particularly important for serialization format in a database server environment where XML stream is seekable but has to allow efficient partial updates, where parsing performance and memory consumption are the priorities, and where I/O is easier to scale than CPU power. Bayardo et al. in [10] analyzed use of various binary XML features on XML parsing for stream based XML query processing. Our paper validates that only the most basic binary XML features are requested in database servers. We also listed the necessary XML parsing API performance improvements and the ways to integrate with the database server.

7 Conclusion and Future Work

We took a detailed look at the runtime optimization side of XML processing in relational database systems based on the XML processing framework in Microsoft SQL Server 2005 and showed that generally XML support as a native data type can be highly efficient and scalable.

We analyzed performance bottlenecks of XML parsing in database server environment and set the goal of implementing a streaming XML shredding and generation that do not require buffering large or unbounded amounts of data in memory, have minimal data copying, and avoid unnecessary well-formedness and validation checks. We identified that changing the XML scalar storage format from text XML to a binary XML format is necessary for building a fully streaming XML parser and identified the necessary properties of the binary format. We also listed improvements of the XML parsing and generation API required in order to take advantage of the binary XML format so there's no need for the parser to buffer any parts of XML in memory. Building a custom XML parser allowed avoiding well-formedness and validation checks when evaluating XQuery expressions – all checks are done when instantiating the XML instance. We measured XML query performance gain in the range of 1.5 to 4.17 for untyped XML when using the binary XML format and the new APIs.

We didn't specifically measured performance gains from using BLOB fragment references to avoid large value copies. While this optimization benefits all queries with large values, it can have a big impact for XML queries that filter out nodes with large values based on XPath or other predicates. In such cases the performance gain from skipping the retrieval of some of disk pages containing the LOB can be very large in corner cases with very small markup-to-values ratio and large values like in case of storing audio and video data in XML format.

Parsing typed XML for XQuery evaluation does not require full XML validation and only requires adding type annotations and "typing" values. Using the binary XML format allows to completely avoid XML validation during XQuery evaluation and consequently brings even bigger performance gains for XQuery on typed XML BLOBs – we measured a 5x average for the XMark schema.

As SQL Server always validates XML coming from the client side, a conversion from text to binary XML on the way into the server space is not really an overhead. However, when sending XML to clients that do not support the binary XML format the server has to perform binary-to-text XML conversion. Even though SQL Server performs such conversion by streaming text XML directly into network buffers, it is still a considerable overhead comparing to the simple retrieval of an XML BLOB.

For future releases we envision the binary XML features to remain basic so they do not require more CPU utilization. Typed XML processing may benefit from keeping QName token definitions for QNames present in XML schema as metadata and only adding them when sending binary XML to components that do not have access to SQL Server metadata. Another idea that can benefit typed XML parsing without adding considerable CPU utilization is to package all statically known singleton properties (attributes and elements) for a typed element into a record-like structure, thus avoiding storing mark-up for such properties.

The current LOB storage on the SQL Server Storage Engine level is B^+-trees optimized for random seeks on offset in BLOB. We think that changing the XML BLOB storage format to trees that are optimized for seeks on XML node ID (ORDPATH) can bring the biggest performance gain in XML runtime area. An ability to bind to such a tree as both scalar and a row set would make creating Primary XML index unnecessary – secondary XML indexes can be built on top of such an XML BLOB storage.

Acknowledgements

The authors would like to thank their colleagues Istvan Cseri, Goetz Graefe, Oliver Seeliger, Dragan Tomic, Shankar Pal, Jinghao Liu, Mike Rorke, Derek Denny-Brown for their support and discussions on XML runtime performance improvements.

References

1. M. Rys: XQuery in Relational Database Systems. XML 2004 Conference
2. S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, V. Zolotov: Indexing XML Data Stored in a Relational Database. VLDB Conference, 2004

3. Edited by S. Boag, D. Chamberlin et al.: XQuery 1.0: An XML Query Language. W3C Working Draft 04 April 2005, http://www.w3.org/TR/xquery/
4. Edited by M Fernández, A. Malhotra et al.: XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 4 April 2005, http://www.w3.org/TR/xpath-datamodel/
5. PE. O'Neil, EJ. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels, SIGMOD Conference, 2004
6. E. Rahm, T. Böhme: XMach-1: A Multi-User Benchmark for XML Data Management. Proc. VLDB workshop Efficiency and Effectiveness of XML Tools, and Techniques, 2002
7. S. Pal, M. Fussell, I. Dolobowsky: XML support in Microsoft SQL Server 2005. MSDN Online, http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp
8. AR Schmidt, F. Waas, ML Kersten, MJ Carey, I. Manolescu, and R. Busse. XMark: A benchmark for xml data management. In VLDB, pages 974-985, 2002
9. M. Nicola and J. John , XML Parsing: A Threat to Database Performance. CIKM 2003
10. R. J. Bayardo, V. Josifovski, D. Gruhl, J. Myllymaki: An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. WWW 2004 Conference
11. Report From the W3C Workshop on Binary Interchange of XML Information Item Sets. http://www.w3.org/2003/08/binary-interchange-workshop/Report
12. CC. Kanne, G. Moerkotte: Efficient Storage of XML Data, ICDE 2000
13. M. Rys, D. Chamberlin, D. Florescu et al.: Tutorial on XML and Relational Database Management Systems: The Inside Story, SIGMOD 2005
14. S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, P. Kukol, W. Yu, D. Tomic, A. Baras, C. Kowalczyk, B. Berg, D. Churin, E. Kogan: XQuery Implementation in a Relational Database System. In proceedings of VLDB 2005 Conference
15. WY. Lam, W. Ng, P. Wood, M. Levene: XCQ: Xml Compression and Querying System. Proc of WWW 2003 Conference
16. B. Martin, B. Jano: WAP Binary XML Content Format. W3C NOTE 24 June 1999, http://www.w3.org/TR/wbxml/
17. K. Beyer, R.J. Cochrane et al: System RX: One Part Relational, One Part XML. SIGMOD 2005