XML Parsing: A Threat to Database Performance

Matthias Nicola
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA 95123, USA
mnicola@us.ibm.com

Jasmi John
IBM Toronto Lab
8200 Warden Ave
Markham, ON L6G 1C7, Canada
jasmij@ca.ibm.com

ABSTRACT

XML parsing is generally known to have poor performance characteristics relative to transactional database processing. Yet, its potentially fatal impact on overall database performance is being underestimated. We report real-word database applications where XML parsing performance is a key obstacle to a successful XML deployment. There is a considerable share of XML database applications which are prone to fail at an early and simple road block: XML parsing. We analyze XML parsing performance and quantify the extra overhead of DTD and schema validation. Comparison with relational database performance shows that the desired response times and transaction rates over XML data can not be achieved without major improvements in XML parsing technology. Thus, we identify research topics which are most promising for XML parser performance in database systems.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems–*transaction processing.*

General Terms

Algorithms, Measurement, Performance, Design.

Keywords

XML, Parser, Database, Performance, SAX, DOM, Validation.

1. INTRODUCTION

XML has become much more than just a data format for information exchange. Enterprises are keeping large amounts of business critical data permanently in XML format. Data centric as well as document and content centric businesses in virtually every industry are embracing XML for their data management and B2B needs [8]. E.g. the world's leading financial companies have been working on over a dozen major XML vocabularies to standardize their industry's data processing [9].

All major relational database vendors offer XML capabilities in their products and numerous "native" XML database systems have emerged [2]. However, neither the XML-enabled relational systems nor the native XML databases provide the same high performance characteristics as relational data processing. This is partially because processing of XML requires *parsing* of XML documents which is very CPU intensive.

The performance of many XML operations is often determined by the performance of the XML parser. Examples are converting XML into a relational format, evaluating XPath expressions, or XSLT processing. Our experiences from working with companies, which have introduced or are prototyping XML database applications, show that XML parsing recurs as a major bottleneck and is often the single biggest performance concern seriously threatening the overall success of the project. This observation is general to using XML with databases, not particular to any one system.

2. XML PARSING IN DATABASES

There are two models of XML parsing, DOM and SAX. DOM parsers construct the "Document Object Model" in main memory which requires a considerable amount of CPU time and memory (2 to 5 times the size of the XML document, hence unsuitable for large documents). Lazy DOM parsers materialize only those parts of the document tree which are actually accessed, but if most the document is accessed lazy DOM is slower than regular DOM. SAX parsers report parsing events (e.g. start and end of elements) to the application through callbacks. They deliver an event stream which the application processes in event handlers. The memory consumption does not grow with the size of the document. In general, applications requiring random access to the document nodes use a DOM parser while for serial access a SAX parser is better.

XML parsing allows for optional validation of an XML document against a DTD or XML schema. Schema validation not only checks a document's compliance with the schema but also determines type information for every node of the document (aka *type annotation*). This is a critical observation because database systems and the Xquery language are sensitive to data types. Hence most processing of documents in a data management context not only requires parsing but also "validation".

Depending on an XML database system's implementation, there are various operations which require XML parsing and possibly validation. The first time a document gets parsed is usually upon insert into the database. At this point, parsing can only be avoided if the XML document is blindly dumped into the database storage (e.g. varchar or CLOB), without indexing or extracting information about it, severely restricting search capabilities. The initial parse of an XML document may also include validation, subject to the application's requirements.

Updates to an XML document in a database may require reparsing the entire document with or without validation, revalidation of the entire document without parsing, partial (incremental) validation, or none of the above. For example, if XML is mapped to a relational schema ("shredded") then an update to the XML view is

CIKM'03, November 3–8, 2003, New Orleans, Louisiana, USA.
Copyright 2003 ACM 1-58113-723-0/03/0011…$5.00

translated into SQL that are applied to the relational system without need for XML parsing. The problem then is validation.

Kim et al. [6] distinguish *partial* and *full* validation as well as *deferred* and *immediate* validation. "*Deferred*" means that validation is performed after updating the document, and the update is rolled backed if validation fails. This is an *optimistic* approach. "*Immediate*" means that validation is performed before executing the update which is then processed only if valid. Kim et al. find the cost of full validation grows super-linearly with document size while partial validation is constant [6].

If XML is shredded into a relational schema, read operations, such as XQueries or XPath expressions, are translated into SQL and do not require XML parsing. Other implementations, e.g. main memory XPath processors, read and parse plain XML documents, often using DOM. For these, the parsing overhead is often an order of magnitude more expensive than XPath evaluation itself [7].

3. PARSER-BOUND XML APPLICATIONS

In this section we report real-word XML database usage situations where parsing performance is a key obstacle. These are experiences from dealing with companies who are currently using XML in their databases and applications, or are intending to do so in the near future. Some of them use database systems from multiple vendors, so the experiences described below apply to using XML and databases in general and not to any particular system.

3.1 A Life Science Company

This life science company receives and produces XML documents between 10MB and 500MB in size regularly. These are currently held in XML files in the file system. The company looks for an ad-hoc way to import these documents into a relational database but concludes that none of the major database systems is currently able to digest such large documents with acceptable elapsed time. The root cause is the CPU consumption of XML parsing.

This is a common case. Sometimes large XML documents can be split into smaller pieces so that a DOM-based solution can be used. However, splitting XML documents requires some sort of parsing in itself and can usually only be done if the XML consists of repeating blocks which are semantically independent. In simple cases, file system tools such as "csplit" are sufficient. We know of life science and other applications where this has been used. XML cutting tools for more complex and automated splitting of XML usually require SAX parsing or a cheaper version thereof.

3.2 XML Loader in IBM Red Brick®

Due to near 24x7 operations, high speed bulk loading is one of the key requirements that we find in data warehousing. Version 6.2 of IBM Red Brick Data Warehouse® XML supports XML load and export. For XML load, one or more typically very large files are parsed and mapped to relational rows. The XML4C SAX parser was chosen due to the large document size. The SAX parser streams the parsing events to the "rowmapper" component, which builds relational input rows based on the mapping specification provided in the load control file. These relational rows are then pipelined from the rowmapper into the regular load logic.

On a 16-CPU SMP machine we loaded a single large flat file in delimited field format (> 10 million rows) as well as the same data in XML format (~3GB in 8 files). The XML load was ~26 times slower than the equivalent flat file load (see Fig. 1). Over 99% of this overhead was XML parsing. This did not include validation. If indexes are built during the load, it slows down the flat file load but not the XML load. Since parsing is the bottleneck, index building can easily be done in the background.

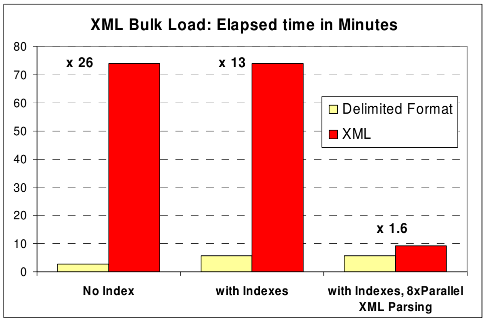

Fig. 1: Loading XML with serial vs. parallel XML parsing

Setting the `max_xml_tasks` parameter to 8 allows Red Brick to parse up to 8 XML input files in parallel. The parallel rowmapper processes write to a staging area, from which the relational part of the loader can consume its input decreasing the delay from XML parsing. This reduces the total XML bulk load time significantly. This emphasizes that XML parsing can be highly parallelized if the XML data comes in multiple documents. Still, this parallelism only helps the overall throughput of the XML load but not the parsing time of individual XML documents. Thus, database applications that require very short response times on a single document do not benefit from parallel parsing in this manner.

3.3 A Large Bank

This bank's OLTP system serves ~20,000 users and executes on average 17 million transactions a day. They are investigating a novel application which involves the exchange of XML documents between the database and the application layer. Each document needs to be parsed before insertion. Their tests with SAX parsers left them far from the OLTP performance they require. Therefore, they implemented their own special purpose parser, optimized towards their particular type of XML. Still, XML parsing time is more than 1 second per transaction which is ~80% of the total response time. The requirement is 100ms or less.

The same bank runs, like most other banks, a variety of batch jobs every night. Their current estimates indicate that any involvement of XML data would make some (sequential) batch jobs exceed their allotted time slot. Other batch jobs which can run massively parallelized may benefit from parallel XML parsing similar to the Red Brick's XML bulk load described above.

3.4 A Different Banking System

In this system, the intended usage of XML includes several thousand distinct XML schemas with potentially millions of XML documents per schema. Most of the XML documents tend to be very small (10K or less) but a single transaction may touch 30 or more XML documents for read, insert and update. Thus, given the demand for extremely short response times, the overhead of XML parsing and schema validation is a major performance concern.

3.5 A Securities Transaction Processing Firm

This company is one of the world's largest providers of IT systems for processing brokerage transactions and securities data. Their system is used for handling transactions for various financial instruments, such as equities and funds, as well as for managing customer and firm accounts. One part of their system will receive multiple streams (10 or more) of securities data. These streams may or may not be in XML format but need to be converted into a common XML format and stored and queried in a database. The goal is to support up to 50,000 XML transactions per minute. Initial tests have shown that XML parsing is a main hurdle for achieving the desired performance on the target hardware.

4. XML PARSER PERFORMANCE

Given the experiences above, we analyze the performance of XML SAX parsing. Parser performance depends on document characteristics such as tag-to-data ratio, heavy vs. light use of attributes, etc., but we do not strive to quantify these dependencies here. Our goal is to relate the cost of IBM's XML4C SAX parser [3] to relational database performance. We use XML documents which are representative of several real-world XML applications, including financial data such as FPML [9].

4.1 Path Length Analysis

We count the number of CPU instructions required to parse different XML documents between 2K and 16K with the XML4C SAX parser (written in C++, AIX 4.3.3). The instruction count is a metric of the expected CPU time and allows us to assess the performance of individual parser components. We parsed each document twice with a single parser instantiation, and collected the instruction count of only the second parse. This excludes the considerable parser instantiation and start-up cost. E.g. parser instantiation can take 5 times as long as parsing a 100k document. A subset of the results is shown in Table 1.

Table 1. Instruction count of XML4C (SAX)

	Doc1: 2K	Doc2: 4K	Doc3: 10K	Doc4: 16K
XML4c5.0	515,705	778,738	1,269,157	3,816,107
XML4c5.1	462,566	660,927	1,108,980	3,512,110

Breaking down the instruction counts by subroutines revealed the top 3 most expensive parser components. (1) Memory management, (2) transcoding the input document encoding to UTF-16, (3) attribute handling, especially attribute normalization. The key issues with memory management are frequent allocation and deallocation of objects, frequent copies of input data in memory, and in-memory copies for transcoding. (see section 5). For 7 documents between 1k and 95k we also calculated the number of instructions per kilobyte of XML data parsed (Table 2).

Table 2. Instruction count per KB of XML

	Min	Max	Avg
XML4c5.1	110,898	231,283	174,364

So, SAX parsing ranges between 460,000 and 3.5 million instructions for documents of 16k and less, and the average parsing cost per KB for documents <100k is approximately 175,000 instructions. For comparison, inserting a row into a relational table requires about 30,000 to 200,000 instructions, depending on the row length, data types and other factors. Also, typical OLTP transactions in relational databases range from several hundred thousand to several million instructions, depending on their complexity.

Comparing these numbers shows that XML parsing can easily double or triple the instruction count of a database transaction. For businesses like the financial companies described in sections 3.3 through 3.5, this is very disconcerting. Imagine the application in section 3.4 parsing 30 documents of only 2k within one transaction. This increases the transaction cost by 13.8 million instructions, the equivalent of about 30 to 40 simple OLTP transactions. This cost is even higher with DTD or schema validation.

4.2 DTD and XML Schema Validation Cost

We quantify the validation cost with simple timing tests, parsing 3 XML documents of 10KB, 100KB, and 1MB (using XML4C SAX). Each document was parsed with and without DTD and schema validation 1,500 times, without grammar caching. Although validation cost depends on the schema complexity, we only exemplify the dramatic overhead of XML validation against moderately complex DTD and schema definitions (see Table 3).

Table 3. Parsing time with & without validation (1,500 times).

	10KB	**100KB**	**1MB**
Without validation	11.19 sec	41.91 sec	410.69 sec
With DTD validat.	22.64 sec	59.23 sec	494.97 sec
Validat. Overhead	102.32%	41.33%	20.52%
With XML schema	50.73 sec	107.6 sec	784.17 sec
Valid. Overhead	353.35%	156.84%	90.94%

The validation overhead relative to the total parsing time depends on the size of the document. It is relatively higher for small documents due to the fix cost to read and parse the DTD or schema itself. Some parsers, including XML4C, allow to pre-parse and cache DTD or schema grammars for subsequent validations. The performance gain is big if a large number of small documents is validated against the same schema or DTD. But, schema caching is less effective for applications that deal with a large number of different schemas, such as the banking system in section 3.4.

Figure 3 shows a breakdown of the total parsing time and emphasizes that schema validation can easily double, triple or quadruple the parsing cost. In section 4.1 we estimated that SAX parsing *without* validation increases the CPU cost of a database transaction by a factor of 2x to 3x. Adding schema validation raises this factor to a range of 4x to 12x. This is a serious threat to database performance since not only transaction response times but also throughput is heavily dependent on the CPU consumption.

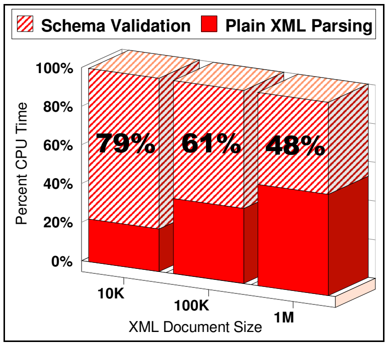

Fig. 2: Breakdown of total parsing time

5. FURTHER WORK

Given the severe conflict between XML parsing/validation cost and transaction processing performance requirements, significant progress in research and development is needed. In this section we identify current and further research topics for improving XML parsing in general and specifically in database systems.

Tighter integration of database system and XML parser. Frequently, SAX parsers make in-memory copies of data fragments and pass them to the event handlers. Often, the event handlers then copy the data yet another time, i.e. into the database's own data structures and memory management. If the database had direct access to the parser's buffers, or vice versa, a lot of memory copies would be avoided. Preliminary tests with Xerces and an XPath processor as "the database" [5] have shown that tighter memory integration may yield a performance gain of ~3x.

Separate 'validation' from 'type annotation'. Type annotation of XML document nodes happens during schema validation. We suggest to let applications choose whether validation, type annotation, or both should be performed. For example, if the database receives XML from a trusted source then it may not need to check compliance with a schema but may still want to obtain type information for optimal Xquery support.

Better support for incremental parsing/validation. Considering the cost of validation, reparsing or revalidating a full document as part of a small update is often unacceptable overhead. More work is needed to investigate efficient mechanisms for partial and incremental validation of updates [6].

Provide more flexible transcoding options. Related to [5] and [1], preliminary tests with an XPath processor over Xerces-SAX achieved a 2x speedup by feeding UTF-16 to the parser and bypassing the transcoding routines. We suggest research into optimization of transcoding algorithms and encoding specific parser libraries that avoid transcoding altogether. E.g. a parser that is entirely UTF-8 based can have significant performance benefits.

Research into parsing with intra-document parallelism. Life sciences and content management require database support for very large XML documents (MBs to GBs per doc). Work is required to learn how best to fork & merge concurrent parsing tasks.

Research and API development for pull parsers. The idea of XML pull parsing, as opposed to SAX, is to let the application request ("pull") the next event instead of being forced to consume a stream of events. E.g. at a specific node the application may decide to move to the next sibling, allowing the parser to use whatever highly-optimized code it has to quickly get there and "skip" the current node's sub tree. The potential performance gains are remarkable but a standard API is still in development.

Optimized parser design and deployment for repeated parsing. Databases often have to parse many XML documents at a time on an ongoing basis, e.g. bulk inserts, or financial OLTP workloads. Techniques like grammar caching and maintaining a pool of live parser instances improve the performance of repeated parsing. More work is required to further optimize for repeated parsing.

Optimized XML parsing primitives for specific CPU architectures. Fundamental but costly parsing routines could be optimized for each target CPU. For example, transcoding or searching for special characters could likely be sped up. Also, hardware assisted parsing engines should be an active research topic.

Optimize parsing across database and application server. Often the database receives XML documents from an application server.

If the application server parses and perhaps validates the XML documents, then these documents should be given to the database in a parsed format, including the Post-Schema-Validation Infoset (PSVI). Optimized parsing for the database and application server as a whole is required to ensure high end-to-end performance.

Binary XML. Binary XML formats encode parsed XML documents to reduce the transmission and storage size. Bin-XML encodes the PSVI including all type information [1]. The encoded documents can be accessed through decoders with DOM and SAX APIs up to 60 times faster than a SAX parser with non-encoded XML. There is great potential for binary XML to reduce storage and processing costs, but the integration with databases operations like updates or XQuery evaluation needs further work.

6. SUMMARY

We reported real-world experiences of using XML with databases where XML parsing was the main performance bottleneck. This motivated an analysis of the cost of SAX parsing and DTD & XML schema validation. We find that parsing even small XML documents without validation can increase the CPU cost of a relational database transaction by 2 to 3 times or more. Parsing with schema validation and without grammar caching can increase transaction cost by 10 times or more. This is a serious problem for high performance transaction oriented database applications which intend to use XML. Therefore we identified and encouraged specific research topics for XML parsing in databases.

7. REFERENCES

[1] Barton, C., Charles, P., Goyal, D., Raghavachari, M., Josifovski, V., and Fontoura, M.,: *Streaming XPath Processing with Forward and Backward Axes*. ICDE 2003

[2] Bourret, R.: *XML Database Products.* http://www.rpbourret.com/xml/XMLDatabaseProds.htm

[3] Expway*: Bin-XML^{TM} for encoding XML documents.* http://www.expway.com/graph/Bin-XMLTechnical%20White%20Paper-jan03.pdf , 2003

[4] IBM XML for C++, http://www.alphaworks.ibm.com/tech/xml4c

[5] Josifovski, V., Fontoura, M., and Barta, A.: *Enabling relational engines to query XML streams*. IBM Internal publ., 2002

[6] Kim, S., Lee, M., and Lee, K.: *Immediate and Partial Validation Mechanism for the Conflict Resolution of Update Operations in XML Databases*. Advances in Web-Age Information Management (WAIM), 2002: 387-396

[7] Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E. J., and Zhang C.: *Storing and querying ordered XML using a relational database system*. SIGMOD Conference 2002: 204-215

[8] XML Applications and Initiatives, http://xml.coverpages.org/xmlApplications.htm

[9] XML on Wall Street, http://lighthouse-partners.com/xml