Index Challenges in Native XML Database Systems

Henrik Loeser[1], Matthias Nicola[2], Jana Fitzgerald[2],

1: IBM Deutschland Research & Development
Schönaicher Str. 220, D-71032 Böblingen

2: IBM Silicon Valley Lab.
555 Bailey Ave, San Jose, CA 95123, USA
Email: hloeser@de.ibm.com, {mnicola, jfitzge}@us.ibm.com

Abstract. Today, more and more enterprises process XML data, many of them already in XML database systems. Once systems start to grow in size, scalability becomes an issue. One of the core operations during insert processing is the index maintenance. Typical relational systems only have few indexes per table, however, XML database users already look at creating hundreds of XML indexes per XML column. A similar situation is present during query processing when many indexes are present which the system can choose from and has to combine them for optimal performance. This creates new challenges for the database system – unknown in the traditional relational world. In this paper, we discuss those index challenges and possible solutions.

Keywords. XML, Indexing, Database Systems, Scalability, Performance

1 Introduction

Today, more and more enterprises process XML data. That data comes from different sources and is often managed by separate applications and stored in separate databases. There are also scenarios where XML data with a very high degree of variability is stored in a single database or even a single XML column. This creates challenges for native XML database systems. In this paper, we focus on the challenges related to maintaining XML indexes in insert-intensive applications.

The majority of existing work on XML indexes focuses on query performance and not on insert performance. In fact, the majority of research papers that propose XML index structures do not evaluate the cost of building and maintaining those indexes. They typically use a query-only workload to evaluate the query performance only. However, real-world XML applications often consist of a mix of read and write operations, and even a moderate amount of insert, update and delete operations can adversely affect overall system performance if the index structures are heavy and require costly maintenance.

In section 2, we first describe typical XML usage patterns in real-world applications, characteristics of popular industry standard XML formats, and the resulting index requirements. Then we describe a specific application scenario – financial application logging – and the index challenges that it creates. In section 3 we explain how XML data can be indexed in DB2 and how optimized index processing can address the challenges posed by the application logging scenario. Section 4 presents a performance evaluation of different approaches for maintaining XML indexes. We conclude the paper with a summary in section 5.

2 XML Index Requirements in Real-World Application Scenarios

In this section we discuss common characteristics of XML applications and how these characteristics frequently require the use of a large number of indexes.

2.1 XML Representation of Business Objects

XML is commonly used as a format for business objects, or messages that represent or transport these business objects. Common examples include purchase orders, trades, customer records, sales records, patient records, claim forms, and other business artifacts. In many cases these business objects can be very diverse and variable and it requires a large number of attributes to model them. A classic example is customer demographics such as age, geographical location, marital status, ethnicity, income range, profession, favorite newspaper, favorite sport, holiday preferences, hobbies, current car, and many other attributes. Another example is product characteristics such as color, size, weight, length, volume, material, water resistance, power requirement, resolution, operating temperature, adapters, display type, battery capacity, and many more. A business with a diverse set of products can easily require hundreds or several thousands of attributes to model their product portfolio. However, any particular product typically only has a subset of these attributes, often between 10 and 100. Such data is also known as having *sparsely populated attributes*.

Examples of such diverse and feature-rich products include electronics, telecommunication services, and financial investment products. It's easy to see that a similar richness in attributes exists in many other industries as well, such as patient records in health care or tax forms in government.

Such rich application scenarios and business requirements have existed long before the emergence of XML and are not fundamentally new. In fact, they have frequently posed challenges in relational database modeling. One such problem is that the number of attributes might exceed the maximum row length or the maximum number of columns in a relational table. Sometimes this can be addressed by splitting the attributes over multiple tables and multiple rows. However, this often leads to a less intuitive database design and additional un-natural normalization. This in turn makes inserting and querying the business data more challenging for application developers or business reporting analysts.

Another problem with large numbers of attributes is that the set of attributes typically changes over time. This is the case whenever products, services, or business processes change, i.e. all the time. In many companies, modifications to relational schemas, such as adding or dropping columns in a table, are strictly regulated and can require lengthy approval processes. It is not uncommon that this slows down the introduction of new products or services. It can also delay or even prevent the availability of new insights from enriched business reports.

A potential but undesirable "solution" for dealing with evolving and large sets of attributes is the use of name-value pairs. This approach is often implemented as a relational table with three columns (`id`, `name`, `value`). In this design the attributes are not expressed as column *names*, but as *values* in the column "`name`". This approach has significant inherent drawbacks, including:

- It's very difficult and often impossible to define business rules and constraints for Name/Value Pairs. This is because the meaning of the entries in the column "`value`" changes from row to row.
- For the same reasons, the column statistics that usually aid relational optimizers in the selection of good access plans suddenly have a different meaning. Hence optimizing complex queries for name-value pair data is often challenging.
- Business objects are now represented in a form that is no longer understood by business experts. Hence, writing queries against name-value pair data is complex and often requires many self-joins. Reporting queries with many predicates prove to be particularly difficult.
- Name-value pair tables handle all data as strings (text). Since the `value` column can contain arbitrary data values, it cannot be typed as integer, decimal, date, or timestamp. This also means that any indexes and comparisons treat the data values as strings. If you search for cars with a price greater than "5000", you will also find cars with prices such as "600" or "900", because these strings are greater than the string "5000". You can solve this with appropriate cast operations in your queries, but those often preclude the use of indexes.

XML allows for a very natural solution of these problems and for a much more intuitive representation of business objects. A key feature of XML is that XML elements and attributes can be optional. If certain properties do not apply to a particular business object, the corresponding elements and attributes are simply absent, and not even represented by NULL values. This means that there does not need to be a predefined and reserved storage location for every attribute for every business object. This allows for a more compact representation and intuitive representation of business objects that is also easier to exchange between applications or organizations.

For these reasons, XML is being adopted in practically every industry to model business objects that have large and evolving sets of sparsely populated attributes.

Companies in every industry have launched consortia to define and standardize XML Schemas for the representation and exchange of data in their industry [3].

For example, the world's leading financial companies have developed more than a dozen XML vocabularies to standardize their industry's data processing [8]. FpML (Financial Products Markup Language), FIXML (Financial Information Exchange), SwiftML, IFX, MISMO (Mortgage Industry Standards Maintenance Organization), OFX, and XBRL (Extensible Business Reporting Language) are among the most popular. FIXML is an industry-standard XML Schema for trade-related messages such as trade capture reports, buy/sell orders, and many others [7]. The FIX protocol is used by more than 150 leading financial companies worldwide. The XML version, FIXML, has been developed to improve extensibility, application layer independence, message validation, and robustness. FIXML also enables straight-through processing, which reduces operating costs and improves the quality and timeliness of information [2].

The FIXML schema defines a very large number of optional elements and attributes. It consists of 41 XML Schema documents and contains 1310 type definitions, 619 element declarations, and 2593 attribute declarations. The vast majority of those are optional and only a small subset of them are present in any given instance document. Table shows that other industry standard XML Schemas define similarly large numbers of types, elements, and attribute.

	Version	Types	Elements	Attributes	XSD Files
ACORD	XMLife_2.16.01	1369	9378	1275	4
ARTS	1.0 - 3.0	4825	6305	2011	32
CDISC	00-9-03	98	84	71	1
FpML	4.2	686	1867	196	23
FIXML	4.4	1310	619	2593	41
HL7CDA	3	1953	945	477	6
IRS1120	2006v3.3	3415	11591	2632	600
MISMO	2.3 - 2.4	2899	1087	13733	31
MCJE/NIEM	1	415	936	46	7
OTA	2003/5	27293	24893	43141	234
STAR	5.0.4	5846	77319	625	192
TWIST	3.1	1016	2314	20	19
UBL	2	682	2665	253	43
UNIFI	1.01 - 2.01	5082	9747	127	71
XBRL	10/25/2006	1858	2847	383	45

Table 1: Characteristics of Selected Industry Standard XML Schemas

Even if many of the defined elements and attributes are used rarely, when they occur they do carry significant business information. To find and exploit this business information and to gain valuable insight from it, it is a common requirement to define

XML path indexes for a large number of these elements and attributes. In specific applications we have seen requirements to index anywhere from a dozen to a few hundred selected elements and attributes. At the same time, simply indexing *all* elements and attributes is an undesirable approach due to its significant overhead for insert, update, and delete operations.

2.2 A Concrete Scenario: Financial Application Logging

We have recently worked with several companies in the finance and insurance industry who have brought up requirements for *financial application logging*. Their requirements were remarkably similar and describe a recurring usage scenario for XML databases. In this section we describe the concrete requirements of one specific company. To protect their identity we simply call them *XYZ Bank* in this paper.

Overview
The internet banking system at XYZ Bank is required to log every event in any of their internet banking applications. Events include clicks that take a user to a new web page or dialog, entry of user data, as well as every click that initiates a banking transaction. This “logging” happens across a set of diverse applications such as checking accounts, loans, investment management, and others. Currently, these applications write "log entries" into a relational database table. This table contains columns for the application id (INTEGER) and a timestamp (TIMESTAMP) as well as a VARCHAR column that contains the actual log entries in a proprietary string format. This string format contains a concatenation of dozens of individual values. This format was chosen because the information captured in the log entries can vary widely from one event to the next. It’s the number of different types of events as well as the diversity of the banking applications which lead to hundreds of possible attributes that can potentially occur in an event. However, any particular instance of an event only carries a few dozen attributes.

The purpose of the application logging includes the support of troubleshooting and problem resolution as well as auditing and compliance regulations for certain applications. Each log entry (event record) contains a fixed set of “header” fields, such as user ID, application ID, session ID, date, and timestamp. These fields exist for every log entry and can easily be stored in fixed relational columns. However, the body of each log entry is highly variable and application dependent, and cannot be mapped to a reasonable relational database schema. Additionally, different application owners require autonomy and flexibility in deciding what information to include in the log records. They need to be able to change existing applications or introduce new applications at any time without causing schema changes in the logging infrastructure.

The Problem
The key problem is that the variable part of the log entries (VARCHAR) is hard to query with adequate performance. Current relational database technology does not allow easy indexing of individual pieces of strings in a VARCHAR column. The applications that read this data typically use SQL "LIKE" predicates on these

VARCHAR values. This results in limited queryability, limited index usage, and sub-optimal performance.

The Desired Solution
The IT department at XYZ Bank has decided to use XML as the new data format for the event records. The benefits of XML for this scenario include that XML tags allow proper labeling of all data fields in a log entry, easy extensibility whenever applications add or change certain fields in the event records, and the ability to index and query individual elements and attributes in the log records.

The Challenge
The application logging workload is very insert heavy. There are approximately 10M to 20M inserts in a 24-hour day, with peak insert rates of 500 events per second. The XML documents that represent the events are 4kb to 20kb in size. The documents have a "header" with identical structure for every document, and a "body" which is highly variable. Due to the variability, XYZ Bank requires a very large number of XML indexes (at least 100). Many of these indexes contain entries for only a few percent of the rows in the table, some indexes less than 1%. Hence, although the number of indexes is large, many indexes are quite small compared to the data. This is because unlike relational indexes, XML indexes in DB2 contain zero, one, or multiple index entries for each document (row), depending on how often the indexed path exists in a particular document. Nevertheless, due to the large number of indexes, the cost of maintaining these indexes during the high insert rate was a key concern for XYZ Bank.

There will be no offline maintenance window for this system since the logging of the internet application has to run 24 x 7. Applications that read the data for troubleshooting and auditing purposes have to support at least 100 users who expect query response times of 1 sec or less. However, the query rates are significantly lower than the insert rates.

In the following we describe how the insert and indexing requirements of this scenario can be fulfilled.

3 XML Insert Processing and Index Maintenance

In this section we review the XML index support in DB2 for Linux, UNIX and Windows and discuss different strategies for maintaining XML indexes efficiently.

3.1 XML Indexes in DB2

Native XML databases and XML-enabled databases have employed different approaches towards indexing XML data, often as a consequence of the different XML storage approaches[1]. Some vendors have chosen to use a "shred and index all" approach where the XML data (stored in a large object in various forms) is first mapped into relational side tables and then either partially or fully indexed in these tables [6]. This can potentially lead to problems caused by the additional processing effort, database logging overhead, and additional space consumption. Hence we believe that a "shred and index all" approach is not an efficient solution to the performance and indexing requirements of the application logging scenario and would not scale to the required database size.

In DB2 a different approach to XML indexing was chosen. Users can index selected elements and attributes and avoid the additional overhead for indexing data items that don't need to be indexed. In DB2, an XML index is created by specifying an XPath-like expression called `XMLPATTERN` that identifies the nodes to be indexed. The specified pattern expression can contain namespace declarations similar to XQuery prolog. It can also contain wildcards and the descendant-or-self axis.

Figure 1 shows how an index on the `applicationID` element within an `event`'s `header` is created. The values are represented as numbers because the data type `DOUBLE` is chosen for the index. DB2 allows for the creation of XML indexes with different data types [5]. DB2 does not attempt to derive the data type from an XML Schema because an application may not have an XML Schema or may store documents for different XML Schemas in the same column.

```
CREATE INDEX appIDidx ON appLogs (log)
GENERATE KEY USING XMLPATTERN
'/event/header/applicationID' AS SQL DOUBLE
```

Figure 1: Example of an XML Index Definition in DB2

When a `CREATE INDEX` statement is issued, the syntax is checked, including the `XMLPATTERN` syntax. Then the index metadata is stored in the system catalog and the physical index structures, including metadata on their own *for runtime use*, are created on disk in the index object. It is noted that every index stands by itself and no combined metadata for all indexes on a table or column is available. This creates additional challenges as we will see later.

[1] It is also worth noting that not all of the so-called "native XML storage" techniques would support the financial application logging scenario. The reason is that XML schema flexibility is required, i.e., the ability to store any well-formed XML document without knowing its schema in advance. Systems that use optimized shredding of XML into (object-) relational structures either need the XML schema to generate more efficient mappings or have to employ a general and less efficient mapping approach.

3.2 Differences to Relational Index Maintenance

In relational database systems it is a possible technique to chain the existing indexes in a linked list per table. The reason is that during insert, delete and possibly update operations every single index that is defined on a table needs to be maintained. For every index, one key needs to be processed for every row. Thus, walking over a chain of indexes is a straight-forward way of processing the index maintenance for all affected indexes.

The situation is quite different with XML indexes in DB2. If the path defined by the `XMLPATTERN` does not exist in a given document, no key for that specific index is produced. Hence, an insert or delete of the document does not affect this index. While the insert or delete of a relational requires *all* relational indexes for the table to be updated, the insert or delete of an XML document may affect only a subset of the defined indexes. This is a significant difference. If a very large number of indexes is defined, as in the application logging scenario described earlier, it is critical for performance to efficiently identify the subset of indexes that need to be updated when a new document of unknown structure is being inserted.

If the `XMLPATTERN` defined by the index exists exactly once in a given document, the index will have one index entry for that document. If a document contains multiple occurrences of the indexed path, then multiple keys are processed for a single index as a result of an insert or delete operation. Update operations may or may not affect the defined XML indexes, depending on whether the indexed paths are modified or not.

What is even more different for maintaining XML indexes is how the index keys are produced. For relational data the entire row is present at once. Additionally, a relational row has a given size and a fixed number of fields and fits into a data page. Thus, keys can be easily extracted from the row and one index key per index can be generated. XML documents are variable in nature and in the absence of a schema their structure is not known until parsed. XML documents are parsed and the resulting data is often available token by token, or in smaller or bigger chunks. Additionally, XML documents can be large and can span many data pages. A naïve implementation might first receive and insert the entire document, and then navigate the document to extract all required index keys. However the two-step process must be avoided for performance reasons. Instead, index keys need to be obtained concurrently with the data processing. More precisely, index keys need to be generated while the XML document is being parsed and formatted into the data pages. This process needs to work in a streaming fashion, so that index entries and data pages are produced for the "beginning" of a large document while the "end" of the document has not been reached and the full structure and size of the document is still partially unknown.

In order to find the relevant parts of the document, the different `XLMPATTERN`s are applied to the document. For relational data the indexed columns are clearly identified as part of the index definition. But, the definition of XPath-based indexes over XML data can contain * and // in the `XMLPATTERN`. This means that the exact location of the index key values is not always known in advance. For example, while the index definition in Figure 1 uses a fully-specified path, a database administrator could have used any of the `XMLPATTERNs` shown in Figure 2. And Figure 2 is not even an exhaustive list of all possible `XMLPATTERNs` that include the `applicationID` element.

```
//applicationID
/event/*/applicationID
/event//applicationID
//header/applicationID
//*/applicationID
//header//applicationID
```

Figure 2: Other Examples of XMLPATTERNs for an XML Index in DB2

Maintaining indexes that are defined with wildcards and descendent-or-self axis require finding and extracting relevant index key data from the XML documents. This adds to the complexity of index maintenance and can have impact on the scalability and performance.

3.3 XML Index Maintenance Strategies

In this section, we first discuss general requirements for efficient XML index maintenance and then describe two possible approaches. Subsequently we look into techniques for improving one of the described approaches in order to meet very high performance and index requirements such as in the application logging scenario.

Chaining vs. Trees
For the decision about how to best manage indexes over XML data, some key requirements as well as overall system architecture need to be taken into account. For the design, the following general principles apply:

- The design and code should be kept simple, so that it can be maintained and is less prone to bugs.
- The memory consumption of the data structure needs to be carefully considered as databases can have thousands of tables and indexes, as well as hundreds or thousands of concurrent insert, update and delete transactions.
- As index maintenance is critical to the insert, update and delete performance, it needs to perform and scale well.

In addition, a balance between setup and execution costs needs to be kept as we will explain in the following. Every index is an object of its own, including some metadata, and only at runtime of an insert, update or delete operation the set of participating indexes can be determined. It is only at this time that combined metadata for the participating indexes can be computed, e.g., to optimize the actual index maintenance[2]. We can therefore distinguish between a runtime preparation/setup phase for index maintenance and its actual execution. Because of the large variety of XML documents and formats that DB2 strives to supporting, it is not possible to assume that either only very large or only very small documents are processed. However, we do observe that applications that process large numbers of small documents (less than 50kb) are a lot more common than applications that process large documents (multiple or even hundreds of MB). Additionally, many applications require high insertion rates. Therefore, it makes sense to optimize for small to medium-sized documents. This means that setup costs for performing the index maintenance for a single document should be kept small as the runtime part will be small as well.

Critical for the performance of XML index maintenance is to efficiently identify the subset of indexes that are affected by the insertion of a new document. This requires matching of the `XMLPATTERN`s in the index definitions against the nodes of an incoming XML document. When considering data structures for this process, trees come to mind. The paths from all `XMLPATTERN`s of all indexes that are defined on the XML column could make up the branches in a pattern tree with some nodes having attached actions (generate key). The actions could be both on leaf and non-leaf nodes as it is possible to index both `/event/header` (atomize the entire subtree) and `/event/header/applicationID`. It is possible to navigate this pattern tree in parallel with and based on the current context of the inserted document. To give an example, with the insertion of a root element `event`, the context pointer would move to the node `event` in the tree. When a child element `header` and later `applicationID` is seen, we first move to `header`, then to `applicationID` in the tree. The attached action to `applicationID` would direct us to generate an index key for the index in Figure 1.
Such a tree can provide very good runtime performance because the current context of the incoming document immediately determines the indexes to be maintained. In contrast to it, the setup costs to create such a tree are high and the integration of patterns with wildcards would make it complex to handle. For each step of every pattern some processing is needed to construct the tree because the tree is assembled from all patterns which need to be inserted into the tree, step by step resulting in the tree's nodes. Wildcards would either need to be expanded with all possible combinations or handled separately.

Another approach is similar to the handling of relational indexes, i.e. to manage index information on a dedicated chain, i.e., a linked list. For every node of the incoming document, the chain would be traversed and every index pattern compared to the current node. Because indexes are already chained in DB2, the setup cost is minimal. The downside is that the run time cost is high and on the order of $\mathtt{n} \times \mathtt{m}$, where $\mathtt{n}$ is

[2] We distinguish between index metadata in the catalogs – seen by the user – and that metadata stored in the index object's header (b-tree) itself. For performance reason only the metadata from the index object, not from the catalog is used which might not be obvious.

the number of indexes and `m` the number of nodes in the incoming document. In the following we discuss techniques to reduce this run time cost.

Starting from the Leaves

To determine whether an index key needs to be generated we have to check whether the `XMLPATTERN` matches the current node and its path. This could be done in an ongoing fashion top-down, similar to the pattern tree, or bottom-up for the currently processed node. In the bottom-up approach, we first check the last step in the pattern against the current node's name. If it matches, the node's parent is checked against the parent step in the pattern and so forth till the root is reached and we have a match.

Since DB2 internally uses 32 bit integer values (stringIDs) to encode node names, the comparison of the names is fast. Because the most common case is a non-match of the current node, the pattern checking usually is only a single integer comparison per index and node (as opposed to comparing entire paths). However, considering the potentially large number of nodes per document and the required number of indexes, the effort per document is still high.

Late In, Early Out – Aggregated Path Information

When looking at applications and their XML documents we observe that usually only leaf nodes are indexed because they are the nodes where relevant business data is located in a document tree. This observation can be used to speed up index maintenance because for any given node in the document the system only needs to check for index matches if there are any `XMLPATTERN`s of the same depth. During the index definition at DDL time the database system can analyze the XPath regarding its minimum and maximum level and for the presence of wildcards.
For the path `/event/header/applicationID` both the minimum and maximum path level is 3 because it is an absolute path. For the path `/event//applicationID` the minimum level is 2 and the maximum is the largest supported document depth, which is 125 in DB2.

At runtime, when the information for all present indexes is available and the environment for index maintenance is being initialized, the aggregated minimum and maximum path level across all participating indexes can be computed. Later, during the actual insert processing when a document is parsed, we only need to check for matching indexes if the current node of the document is within the minimum and maximum level of the defined indexes.

Given that many XML documents fan out over the first few levels and that wildcards are rarely used in indexes –at least for first steps in a pattern– we can avoid index processing for those levels with our "Late In, Early Out" strategy. We start as late as possible (minimum) and try to get out of path matching as early as possible (maximum).

4 Performance Evaluation

If only few indexes are present – typically up to 20 in environments with only one document type per XML column – then the above described strategy based on index chaining with some of the discussed performance enhancements works fine and has a good balance between setup costs and runtime costs. However, it will not scale well for the extreme use case with possibly hundreds of indexes.

Test Scenario:
Since the actual XML data of the financial application logging scenario was not available to us in sufficient quantities, we designed a test scenario that exhibits similar characteristics. This test scenario is based on the open-source benchmark TPoX [9] which simulates a financial online brokerage scenario. It exercises XML inserts and XML index maintenance (among other things). The TPoX data set includes data compliant with FIXML, the XML implementation of the Financial Information eXchange (FIX) standard which contains hundreds of optional attributes and elements. These characteristics as well as document sizes ranging between 1KB and 20 KB in size make TPoX a suitable substitute to mimic the insert and index requirements that we found at XYZ Bank.

For our tests we defined different numbers of indexes (60, 110, 160, and 210), such that for any given document there are always 10 indexes that match nodes in the document, plus 50, 100, 150, or 200 indexes that do not match any nodes in the document. The number of non-matching indexes is the variable aspect in our experiments.

Each of the matching indexes requires an update after each document insert, whereas any non-matching index does not need to be updated during a document insert. Using the multi-user TPoX insert workload, we executed tests in which 100 concurrent users inserted 7000 documents each, i.e. 700,000 documents total. The TPoX workload driver captures the throughput in inserts per minute as well as other statistics [4].

Test equipment and configuration
Tests were run on the following hardware:

• **Processor:**	4 dual-core 1.9 GHz POWER5 processors
• **Memory:**	32 GB
• **Operating System:**	AIX v5.3
• **Storage:**	32 external disks spread over two DS4700 disk array subsystems with a capacity of 72 GB per disk

4.1 Simple Index Chaining

In a first set of tests we measured the performance of the simple chaining approach (described in the beginning of section 3.3) with the "Late in, Early out" enhancement applied. The results for different numbers of indexes are shown in Figure 3.

The vertical bars in Figure 3 show the throughput in inserts/min for different numbers of non-matching indexes. Not surprisingly, the highest insert throughput of 96,463 inserts per minute was achieved for the smallest number of indexes, i.e. just 10 indexes which match nodes in every document. Then we increased the number of indexes so that for any given document there are 50, 100, 150 or 200 indexes that did not match nodes. For the largest number of indexes the insert throughput was reduced by 41% to 56,615 inserts per minute. The percentage by which the throughput is reduced is shown by the curve in Figure 3. The throughput reduction is caused by increasing CPU consumption to compare nodes of the incoming XML documents against the chained list of XML indexes.

Clearly, for a very insert-intensive scenario such as the financial application logging, the performance overhead of the large number indexes is not acceptable. Thus, further optimizations are required to meet the performance requirements.

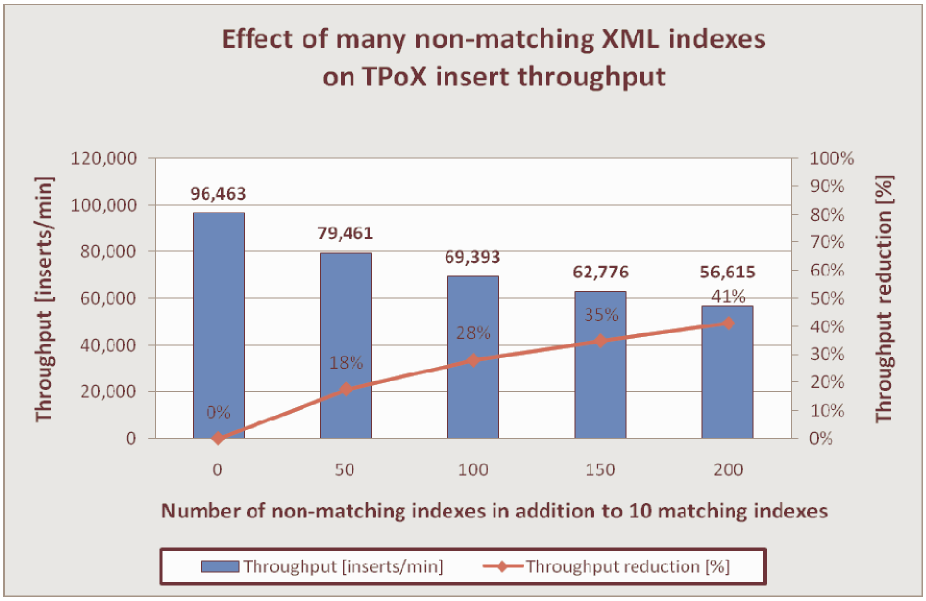

Figure 3: Insert Performance relative to the number of Indexes (simple index chaining approach)

4.2 Multi-level Chaining

The pattern tree approach discussed earlier is expected to meet the performance requirements for the runtime execution part. The reason is that at runtime we would move up and down in the tree based on the context of the incoming XML data. Non-matching indexes impact the navigation slightly because they add additional choices when moving to a child node in the tree and contribute to the overall size of the pattern tree. However, during the runtime setup when the metadata from the physical objects can be combined, the more patterns are added to such a tree during this phase, the more time it takes. Thus, pattern trees are not our preferred choice as they don't keep the desired balance between setup and execution costs.

We mentioned earlier that we found chains to work well for a small number of indexes. In situations where a large number of indexes is required, documents are typically highly variable and the elements and attributes that need to be indexed are distributed across several different levels in the documents. We can use this observation to split the index chain and manage it in smaller chunks. Thereby, we can get back to the short chains which are efficient and meeting the simplicity requirement. One obvious way of splitting up the index chain is by the matching level, i.e., the document depth that the `XMLPATTERN` of an index definition points to. For every possible level, in DB2 up to 125 levels, we maintain a separate chain. In the application logging scenario the documents vary in structure and depths with a typical maximum depth of 10 to 15, with indexed nodes being located anywhere from level 2 to 15. This means that the list of indexes is split into 14 chains. To account for indexes with * and //, the chain for level 0 is used and is always checked. Level 0 can be used for this special purpose because it does not contain any other indexes. The multi-level chain design is illustrated in Figure 4. Additionally, the "Late in, Early out" optimization is used to process the chains.

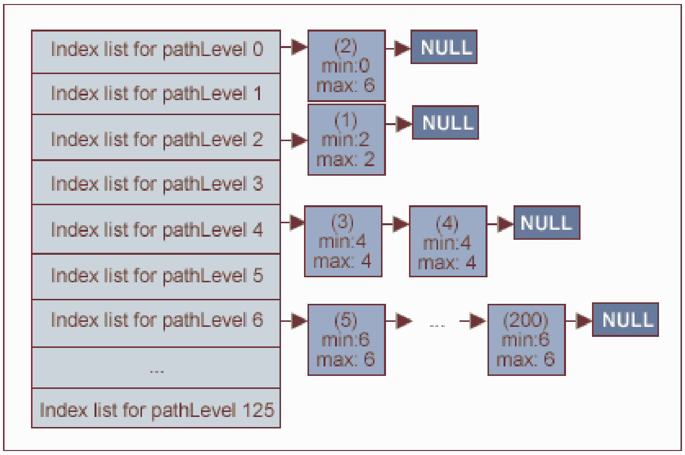
Index list for pathLevel 0
Index list for pathLevel 1
Index list for pathLevel 2
Index list for pathLevel 3
Index list for pathLevel 4
Index list for pathLevel 5
Index list for pathLevel 6
...
Index list for pathLevel 125
(2) min:0 max: 6
NULL
(1) min:2 max: 2
NULL
(3) min:4 max: 4
(4) min:4 max: 4
NULL
(5) min:6 max: 6
...
(200) min:6 max: 6
NULL

Figure 4: XML Index lists for each path level

Performance results

With the new approach of managing the indexes in separate chains for each path level we repeated the insert performance tests. As illustrated in Figure 5 the maximum overhead of index maintenance is reduced to 6%. Also for more moderate numbers of non-matching indexes, such as 50, the impact of indexes that do not match nodes in every document is as low as 2%. What is also visible is that even the case with zero non-matching indexes benefits from the separation.

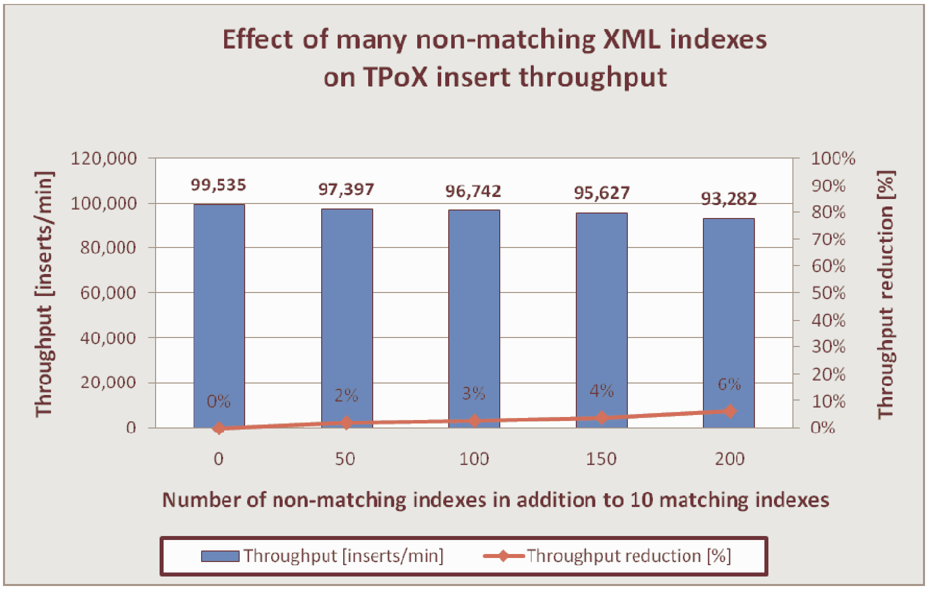

Figure 5: Insert Performance relative to the number of Indexes (multi-level chaining approach)

5 Conclusions and Outlook

In this paper, we discussed the special requirements coming from novel XML-based database application – application logging. It poses special requirements towards native XML database systems because of the mix of all kinds of different XML documents that are stored in a single XML column. In order to support search performance indexes need to be defined for the different document types. This leads to the presence of sometimes multiple hundred XML indexes, something unknown from the relational world. Because application logging is an insert-driven application the question is how the scalability is impacted by so many indexes. As we have shown in this paper, both the performance requirements as well as the design principle of simplicity and maintainability can be met with an approach that is based on what is done in the relational world, index chaining. We have extended this to multi-level chaining and enhanced how and when the index patterns are matched against the currently processed document. With these techniques DB2 scales well in this rather extreme application scenario.
The application logging does not only create challenges for the indexing in native XML database systems, but also for the storage components. The log data from dozens or hundreds of application generates a tremendous amount of XML data that needs to be processed – inserted first and after a retention period moved out of the

database system again. The size of the storage structures, the size of the generated database transaction logs, and many more factors impact the scalability. We therefore are looking forward to meeting those challenges by continuously improving XML storage and indexing technologies.

References

[1] Balmin, A. et al.: On the Path to Efficient XML Queries, *32nd International Conference on Very Large Databases*, September 2006.

[2] Chicago Mercantile Exchange: “*The Business Case for FIXML*”, 2004, http://www.cme.com/files/BusinessCaseFIXML.ppt
http://www.cme.com/clearing/cm/stan/index.html

[3] Malaika, S.: Get started with Industry Formats and Services with pureXML, *IBM developerWorks*, http://www.ibm.com/developerworks/db2/library/techarticle/dm-0705malaika/

[4] Nicola, M., Kogan, I., Schiefer, B.: An XML Transaction Processing Benchmark, *ACM SIGMOD Conference* , June 2007.

[5] Nicola, M., Van der Linden, B.: Native XML Support in DB2 Universal Database, *International Conference on Very Large Data Bases (VLDB),* 2005.

[6] Pal et al.: Indexing XML Data Stored in a Relational Database, *International Conference on Very Large Data Bases (VLDB),* 2004.

[7] The Financial Information eXchange Protocol, FIXML 4.4 Schema Specification 20040109, Revision1 2006-10-06
http://www.fixprotocol.org/specifications/fix4.4fixml

[8] XML on Wall Street, *Financial XML Projects*, http://lighthouse-partners.com/xml

[9] XML Database Benchmark, *Transaction Processing over XML (TPoX)*, http://tpox.sourceforge.net/

IT-Infrastrukturen für flexible, service-orientierte Anwendungen - Ein Rahmenwerk zur Bewertung

Stephan Buchwald[1], Thomas Bauer[1] und Rüdiger Pryss[2]
[1]Abteilung für Daten- und Prozessmanagement, Daimler AG,
{stephan.buchwald, thomas.tb.bauer}@daimler.com
[2]Institut für Datenbanken und Informationssysteme, Universität Ulm
ruediger.pryss@uni-ulm.de

Zusammenfassung: Service-orientierte Architekturen (SOA) sind in vielen Unternehmen ein zwar noch entstehendes, aber bereits auch sehr wichtiges Thema. Ein entscheidender Aspekt jeder SOA stellt die Standardisierung der IT-Infrastruktur des Unternehmens dar. Dadurch sparen Unternehmen Kosten, da sie die IT-Anbieter leichter austauschen können. Ferner reduziert sich durch Standardisierung der Wissensaufwand bei der Softwareentwicklung und die Funktionalität der IT für die Fachanwender wird so vereinheitlicht. Dies bedeutet, dass für jede im Unternehmen benötigte Funktionalität ein Produkt ausgewählt werden muss, das dann für IT-Applikationen als Implementierungsplattform fest vorgegeben ist (bzw. eine kleine Anzahl unterschiedlicher Produkte ggf. verschiedener Hersteller). Obwohl für eine solche Entscheidung die potentiell relevanten SOA-Komponenten bekannt sein sollten, gibt es in der Literatur keine systematische und produktunabhängige Darstellung und Bewertungsgrundlage von IT-Infrastrukturen für eine SOA. Da jedes Unternehmen unterschiedliche Anforderungen an die Flexibilität einer SOA hat, ist zudem eine Betrachtung unterschiedlicher Ausbaustufen der einzelnen SOA-Komponenten sinnvoll. Die Anforderungen der Unternehmen erstrecken sich von mehr Funktionalität für die Benutzer bis hin zu mehr Funktionalität bei der Prozesssteuerung. Darüber hinaus sollen Kosten infolge redundanter Implementierungen vermieden werden. Aus diesem Grund betrachtet dieser Beitrag Komponenten einer SOA ebenso wie eine Darstellung ihres Zusammenspiels in einer IT-Gesamtinfrastruktur. Der Fokus liegt auf Geschäftsprozessen, die geeignet durch IT-Applikationen unterstützt werden sollen.

1 Einleitung

Ein entscheidender Erfolgsfaktor für Unternehmen ist ihre Anpassungsfähigkeit auf Änderungen ihrer Umgebung [MRB08, Rei00, RMRD04, RMR07]. Diese Fähigkeit wird zum Wettbewerbsvorteil, wenn die jeweiligen Anpassungen schneller und kostengünstiger realisierbar sind als bei Konkurrenten. Um diesen Vorsprung zu erreichen, wird die geforderte Informationstechnologie (IT) immer mehr zum Schlüsselfaktor. Unternehmen haben erkannt, dass die geforderte Anpassungsfähigkeit mit monolithischen IT-Systemen nicht realisierbar ist. Um Anpassungen dennoch durchführen zu können, wurden im Laufe der Zeit zahlreiche Technologien und Methoden in den Unternehmen etabliert. Diese wurden eingesetzt, um die Abläufe zwischen den IT-Systemen abzubilden und damit die Integration zu ermöglichen.

Das kontinuierliche Anwenden neu entwickelter Technologien, mit dem Ziel der Integration von IT-Systemen, führte zu immer komplexer werdenden Unternehmenslandschaften. Die Komplexität ist in der Nutzung unterschiedlicher Integrationslösungen begründet, was zu intransparenten Abläufen geführt hat [RD00]. Erschwerend kommt hinzu, dass zusätzlich zur fehlenden Transparenz der Abläufe im eigenen Unternehmen auch die Beziehungen mit Partnern immer komplexer werden. Um dennoch die Transparenz dieser komplexen Abläufe zu erhöhen, werden diese losgelöst von den IT-Systemen dokumentiert, d.h. auf einer fachlichen Ebene. Dies bildet auch die Grundlage für Optimierungen. Diese Abläufe stellen die fachlichen Anforderungen dar, die von den IT-Systemen erfüllt werden müssen, um die Geschäftsfähigkeit des Unternehmens sicherzustellen.
Das Problem der Anpassungsfähigkeit resultiert aus der Notwendigkeit zur effizienten Abbildung der fachlichen Anforderungen, repräsentiert durch die Abläufe im Unternehmen, auf die IT-Systeme. Die fachliche Sicht muss sehr viel stärker betont werden als bisher. Dazu werden die Fachanwender des Unternehmens häufiger in die Anpassung der IT-Systeme einbezogen. Die bisher lang andauernden Software-Entwicklungszyklen werden durch diese Maßnahme kürzer, da Änderungen des Fachanwenders gezielter in die IT überführbar sind. Aus diesen Gründen müssen die fachliche und technische Sicht eines Unternehmens enger aufeinander abgestimmt werden.
Getrieben durch diese Erkenntnis versuchen Unternehmen ihre vorhandene Systemlandschaft service-orientiert auszurichten [Erl05]. Aus fachlicher Sicht beschreibt ein Service eine Funktionalität, die im Unternehmen angeboten oder verwendet wird. Die Einführung von Services und dazu notwendiger Technologien und Methoden führt zu einer service-orientierten Architektur (SOA). Darunter versteht man ein Architekturparadigma, welches das Modellieren von Services, das Ausführen von Services sowie das Kapseln von Funktionalität durch Services und die service-orientierte Softwareentwicklung unterstützt. Hierunter fallen vor allem die Ausrichtung der IT an fachlichen Anforderungen und die schnelle Reaktion auf (geänderte) fachliche Anforderungen. Darüber hinaus bilden Service-Prinzipien wie Kapselung, lose Kopplung, standardisierte Schnittstellen, Auffindbarkeit, Wiederverwendbarkeit und Autonomie von Services die Basis für eine SOA [Erl05].
Wesentlich für die Realisierbarkeit dieser Funktionalitäten ist eine standardisierte, service-orientierte IT-Infrastruktur. Diese beinhaltet funktionale Komponenten zur Modellierung von fachlichen Anforderungen und deren Abbildung auf IT-Systeme. Die IT-Infrastruktur muss so gestaltet sein, dass die genannten SOA-Prinzipien umgesetzt werden können. Um die fachlichen Anforderungen abzubilden, werden Geschäftsprozesse modelliert, die die Abläufe im Unternehmen beschreiben [RD00]. Änderungen an Geschäftsprozessen müssen gestützt durch die IT-Infrastruktur flexibel auf die Implementierung abgebildet werden können. Sowohl die Geschäftsseite (z.B. Änderungen von gesetzlichen Rahmenbedingungen) als auch deren fachliche Umgebung (z.B. organisatorische Umstrukturierung) lösen Änderungen aus. Außerdem können Änderungen durch die Umgebung der Implementierung (z.B. dem physischen Ort der Service-Implementierung) initiiert werden.
Um Unternehmenslandschaften und deren IT-Systeme möglichst flexibel in einer SOA betreiben zu können, ist eine IT-Infrastruktur notwendig, welche die dafür geeigneten Komponenten verwendet. Obwohl Hersteller, Gremien und Autoren zahlreiche SOA-Produkte und -Konzepte beschreiben, gibt es keine konzeptionellen und plattformunabhängigen Betrachtungen von Komponenten sowie deren Funktionalität und Zusammenspiel. Des-