Approximate Query Answering and Result Refinement on XML Data

Katja Seidler[1], Eric Peukert[1], Gregor Hackenbroich[1], and Wolfgang Lehner[2]

[1] SAP Research CEC Dresden, Chemnitzer Str. 48, 01187 Dresden, Germany
{firstname.lastname}@sap.com
[2] Technische Universität Dresden, 01062 Dresden, Germany
wolfgang.lehner@tu-dresden.de

Abstract. Today, many economic decisions are based on the fast analysis of XML data. Yet, the time to process analytical XML queries is typically high. Although current XML techniques focus on the optimization of query processing, none of these support early approximate feedback as possible in relational Online Aggregation systems.
In this paper, we introduce a system that provides fast estimates to XML aggregation queries. While processing, these estimates and the assigned confidence bounds are constantly improving. In our evaluation, we show that without significantly increasing the overall execution time our system returns accurate guesses of the final answer long before traditional systems are able to produce output.

1 Introduction

The data volume and growing rates of today's business systems make approximate query processing an inevitable technique for fast analyses. Online Aggregation (OLA) has been proposed as an approach for analytical processing of relational data. In OLA, the database system quickly returns approximate answers to aggregation queries together with statistical guarantees on the error bounds. This computing paradigm allows users to more flexibly explore data: Query processing may be terminated at any time once sufficient accuracy has been reached; if exact answers are needed, they can be computed with little overhead as compared to traditional systems.

Current concepts for OLA are restricted to relational data. However, today a large and increasing amount of business critical data is represented in XML. Recently, XML gained strong momentum in business applications as business objects are represented and their interfaces are exposed in XML. As many economic decisions today are based on this XML data, a fast analysis of the underlying data sources is essential.

As an example, consider a company that stores sales data in XML format. To rapidly identify new sales strategies the regional manager needs to analyze historical data. He aggregates data based on product type, manufacturer, vending region and date of sale. An XQuery expression that counts all sales of mobile phones manufactured by Nokia and sold within Europe since January 01, 2008 takes the form:

```
let $sale := //sale[
   salesDetails[date > xs:date("2008-01-01")] and
   salesDetails[region = "Europe"] and
   product[type = "mobile phone"] and
   product[.//manufacturer = "Nokia"]]
return count ($sale)
```

In this paper, we show how OLA can be performed on XML data. We present a novel query processing system—referred to as XML Database Online (XDBO) System—which performs OLA on XML data in a scalable way. We have been faced with the following main challenges: Query processing on XML data is heavily dominated by the evaluation of so called twig patterns. This pattern matching process involves a multitude of structural joins and is a major bottleneck within XML query processing. Compared to relational databases the queries differ in both the number and the kind of joins. Additionally, in current XML processing systems an aggregate cannot be returned until the whole structural join operation is completed. This is a time-consuming task, especially for complex queries and/or queries over large data sets. We address these challenges and make the following contributions:

- We propose novel path pattern operators for the random and non-blocking selection and join of query path patterns. Further, the selection operator utilizes a novel index structure for scalable and efficient query processing (Section 2).
- We show how sideways information passing can be used for fast approximate query answering, and we propose the corresponding calculations of running aggregates and confidence bounds (Section 3).
- We introduce the architecture of the XDBO System and present our prototypical implementation (Section 4).
- We point out optimization possibilities and realization ideas (Section 5).
- With an extensive evaluation we demonstrate the feasibility and the efficiency of our approach (Section 6).

In Section 7, we give an overview over the related work, and we conclude the paper with a summary and an outlook in Section 8.

2 Indexing and Pattern Matching

In this section, we introduce the foundations of our proposed XDBO System that are based on the following requirements of OLA: First, for a scalable query processing and for fast first answers the pattern matching must be performed in a *non-blocking fashion*, and second, to guarantee statistical valid estimates and error bounds XML elements have to be processed in *random order*. We now describe two novel pattern matching operators and a special index structure that are designed to meet the given requirements. The operators reflect the general procedure of the approximate query processing in the XDBO System:

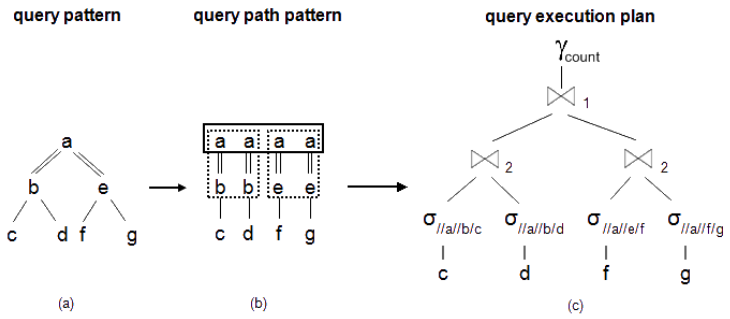

Fig. 1. Generation of the execution plan

Instead of processing a query pattern as a whole, we decompose it into a set of query path patterns. We identify all path patterns of the query and remember the positions of branching nodes that connect individual paths. With a novel selection operator (Section 2.2) we search for solutions to query path patterns; a special index structure (Section 2.1) facilitates its efficiency. Finally, a join operator (Section 2.3) connects the solutions of individual path patterns.

Figure 1 illustrates the main steps, starting from the query pattern (a), all identified query path patterns (b) and the constructed execution plan (c) for the following XQuery-request:

```
let $pattern := //a[.//b[c]/d]//e[f]/g
return count($pattern)
```

2.1 Element Path Index

Traditional XML database systems utilize special numbering schemes to speed up query processing: They label all elements of an XML document and store these labels for each element into an index structure. The index is then used to accelerate a pattern matching operation. To support pattern matching in a system that meets the OLA requirements a special index is needed. Based on a thorough state-of-the-art analysis, we decided to utilize the extended Dewey numbering (EDN) scheme [10]. Like other numbering schemes it encodes the structural relationship into the element labels. Additionally, the EDN scheme is able to encode the whole root-to-node path of each element into the label. This allows to easily retrieve the root-to-node path from the label without accessing the original data tree.

Similar to other labeling techniques, EDN consists of an encoding and decoding phase. To encode the root-to-node path into a label EDN uses context knowledge available from an XML schema or, if no schema is available, from a pre-scan of the XML document. Specifically, for each element type EDN uses information on the types and order of all child elements and captures this information in a finite state transducer (FST).

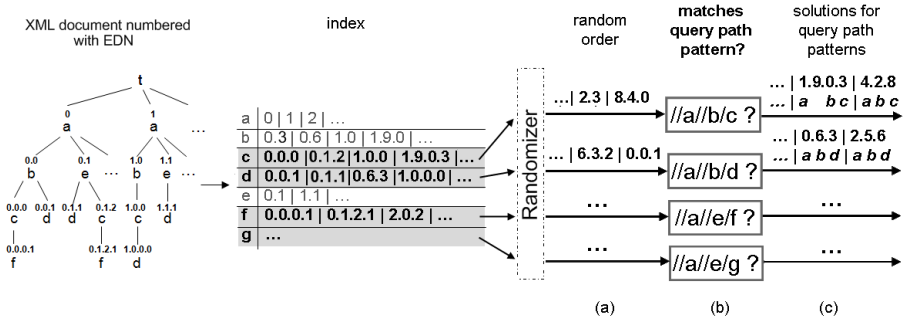

Fig. 2. Selection process

One advantage of the EDN for the pattern matching operations is that only the label of the leaf node element of a query pattern need to be accessed. The solutions of the query pattern $//a//b/c$ can be found by examining all labels of c elements and check if they match the query pattern. Hence, only the decoding of EDN labels and the comparison of the retrieved path patterns with the query path pattern have to be done. This enormously speeds up and simplifies pattern matching.

2.2 Path Pattern Selection Operator

Based on an index that stores EDN labels for each element type, we propose a novel selection operator σ_{XPath} (see Figure 1(c)). The characteristic of this operator is the processing of input elements—the EDN labels from the index—in random order. On the right part of Figure 2, we show how this is achieved (the XML document and the corresponding index are given in the left part): The element labels of the leaf nodes of the query are extracted from the index in random order (a). If the index structure does not guarantee randomness a randomizer has to be used. The selection process is visualized in (b). Here, the operator translates each label of the random input stream with the FST into a root-to-node path and compares it with the query path pattern. If there is a match a solution is found. For further processing, not only the selected labels but also the positions of the query path pattern elements are memorized (c).

2.3 Path Pattern Join Operator

To find solutions for the whole query pattern the results of the individual query path patterns need to be joined at branching nodes. Branching nodes (specifically their position *pos* in the path pattern) are defined by the join operator $\bowtie_{pos}$ (see Figure 1(c)). The join process is much more complex than the relational join operation. Rather than comparing columns of different tuples positions of labels are compared to join the records. Two labels can only be joined if they coincide from root position up to the position of the associated branching node. This

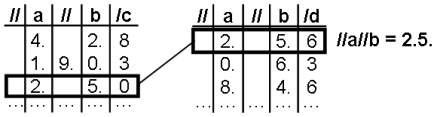

Fig. 3. Join process

join process is illustrated in a relational table style in Figure 3. Labels that have been identified as solutions for the query patterns $//a//b/c$ and $//a//b/d$ are matched to tables with the query pattern nodes depicted as columns and the element labels as tuples. Only the records with matching element labels from the first column up to the join column (here column b) can be joined.

3 Query Processing

This section describes how the XDBO System effectively uses the presented index and the two pattern matching operations to provide early estimates of the final aggregate of a query. To achieve high scalability we employ the concept of sideways information passing. According to this concept, operations at a single level of a query plan are performed concurrently while allowing to share some of its intermediate results with other operations at the same level. Thereby, preliminary result tuples can be generated on the fly on each layer of the query plan and are used to provide an estimate for the final query answer. Sideways information passing was introduced as a design paradigm for OLA of relational data [7]. We show below how sideways information passing and related design principles can be adapted and combined with EDN to allow scalable OLA of XML data. As in [7], we refer to all of the joins at one level of a query plan, i.e., joins that are evaluated concurrently, as a *levelwise step*.

3.1 Overview

Figure 4 depicts the execution process of a query plan in the XDBO System: The two join operations of the lowest level are processed concurrently (a); this is the first levelwise step. By passing information among the join operations an estimate N_1 of the final answer of the aggregation query is maintained. This estimate becomes more and more accurate as the levelwise step progresses. The results of the join operations of the first levelwise step are used as input to the second levelwise step (b). In this step, a second online estimate N_2 is produced and combined with N_1 into a single estimate for the answer. During the final merge process one more estimate N_3 is calculated. When the query execution process terminates, N_3 equals the exact query result and is outputted in step (c). To calculate early estimates it is essential that all path patterns that contribute to the whole query pattern are included in each of the levelwise steps. If the query execution plan cannot guarantee that property for the lowest levels the XDBO System adds, similar to [7], a special re-randomization operation to the query execution plan.

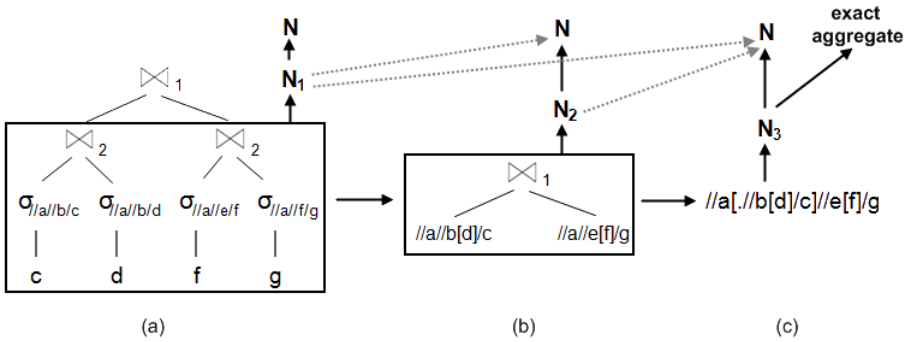

Fig. 4. Levelwise steps

3.2 Levelwise Steps

Each levelwise step is partitioned into two phases: a scan and a merge phase. The scan phase sorts the labels of all individual path solutions and finds early solutions for the whole query pattern. In the merge phase, the actual join is performed. To ensure scalability the set of path pattern solutions that are pipelined into the join operations are divided into equal-sized runs. The size of the runs is adjusted to ensure that the join can be performed in main memory.

At the start of the scan phase, one run of results for each path pattern is read into memory (Figure 5(a)) by the already introduced selection operator. Subsequently, all runs are sorted with the help of a hash function. This hash function simplifies joining and additionally bears a great meaning for the randomization of the output of the join operation. It only takes into account the parts of the labels that are significant for the join operation (specified by *pos*). To guarantee an unbiased label sorting the hash function is initialized with a different seed for each single binary join operation. All records present in memory are immediately joined (in-memory join) thus yielding early total join solutions to the join conditions specified in the query pattern. Based on these early solutions, estimates and error bounds for the final result are generated at each step of the query execution. After this initialization step the path pattern solutions are processed in a round-robin fashion (Figure 5(b)). Specifically, the following tasks are sequentially done for all n path patterns of the levelwise step (assuming an arbitrary ordering) beginning with $i = 1$:

1. Write the sorted run (being currently in memory) of the i-th path pattern and the corresponding hash value back to disk.
2. Read a new run for the i-th path pattern into memory and sort it based on the hash function.
3. Execute a new in-memory join and update the estimate for the final aggregate based on the newly found on-the-fly records.
4. Set $i = \begin{cases} i+1 & i < n \\ 1 & i = n \end{cases}$ and go on with 1.

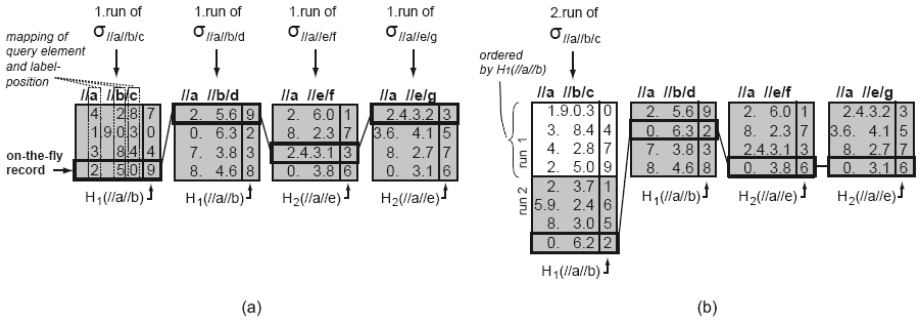

Fig. 5. Scan phase of the first levelwise step

The algorithm will terminate once all solutions of the path patterns have been sorted and the last runs have been written back to disc.

The merge phase is directly connected with the scan phase of the subsequent levelwise step. It provides the following two characteristics: (i) it guarantees a (semi-)random output order and (ii) it produces output partitioned into equal-sized runs that fit into main memory. The partitioning is controlled by the values of the hash function from the scan phase. Based on equal-sized hash ranges the merge process produces runs of joined pattern in a round-robin fashion. The results of the join operations are directly streamed into the next levelwise step. The random output order is guaranteed by the hash function.

The scan phase of the following levelwise steps differs from the first one in the following way: The input elements are not read by the selection operator; instead, they are either obtained from the merge phase of the preceding levelwise step or—if the join input is gained from a re-randomization operation—from the temporarily stored re-randomized solutions of the path pattern.

3.3 Online Estimation

During query processing, XDBO maintains both estimates for the final query result and confidence intervals for the statistical error bound. The computation of these quantities proceeds along the lines described for the DBO System [7]. For completeness, we summarize the main principles of the approach and the specific aspects related to the processing of XML. We do this using the example of a `COUNT` aggregate N.

The calculation of N_i for each level i of the query plan is based on data samples processed up to this level. Accordingly, throughout the computation the values of these aggregates are subject to the sampling procedure, turning the N_i into random variables. The mean and variance of these quantities can be expressed in terms of the contributions obtained in individual runs. Provided m

results of such individual runs are available, we have

$$N_i = \frac{1}{m}\sum_{j=1}^{m} X_{i,j}, \quad \text{and} \quad \sigma^2(N_i) = \frac{1}{m^2}\sum_{j=1}^{m}\sigma^2 X_{i,j} + \frac{1}{m^2}\sum_{j,k=1,k\neq j}^{m} Cov(X_{i,j}, X_{i,k}),$$

where $X_{i,j}$ denotes the estimate of the aggregate of the j-th run of the level i, σ^2 the variance and Cov the covariance of the respective random variables. Similar to [7], we ignore the covariance term (see [7] for a justification and the restrictions of this procedure).

To arrive at early estimates for the final query result, we scale up results of individual runs into estimates $X_{i,j}$ for the eventual query result. If the system breaks data processing into p runs each of which involves joins over n query patterns, the in-memory result must by scaled up by a factor p^n. It is a non-trivial task to perform this scaling so that the resulting estimate is unbiased; [7] provides efficient algorithms that compute unbiased estimate at all levels of the query plan.

In the query execution, XDBO computes estimates for the layers of the query plan. For a query plan comprising d layers we combine the estimates with

$$N = \sum_{i=1}^{d+1} w_i N_i, \quad \text{and} \quad \sigma^2(N) = \sum_{i=1}^{d+1} w_i \sigma^2(N_i)\,,$$

into an estimate for the final answer where the weights w_i take on the value

$$w_i = \frac{1}{\sigma^2(N_i)\sum_{j=1}^{d+1}\frac{1}{\sigma^2(N_j)}}\,.$$

The estimate for the query answer is updated after each in-memory join. First, mean and variance of the join result are computed and scaled up according to the number of runs and the number of joins; second, the estimators of the levelwise step are updated with the new estimates and finally, the new total estimate and variance of N are retrieved.

4 The XDBO System

We now present the architecture of the XDBO System (Section 4.1) and provide some insight into its prototypical implementation (Section 4.2).

4.1 Architecture

The XDBO System comprises two main parts which are the Data Import and Indexing Component (Part (I) of Figure 6) and the core XML Database System (Part II). They are supplemented by an underlying index storage.

Data Import and Indexing. The Data Import and Indexing Component comprises three sub-components: the Data Import Interface, the Numbering

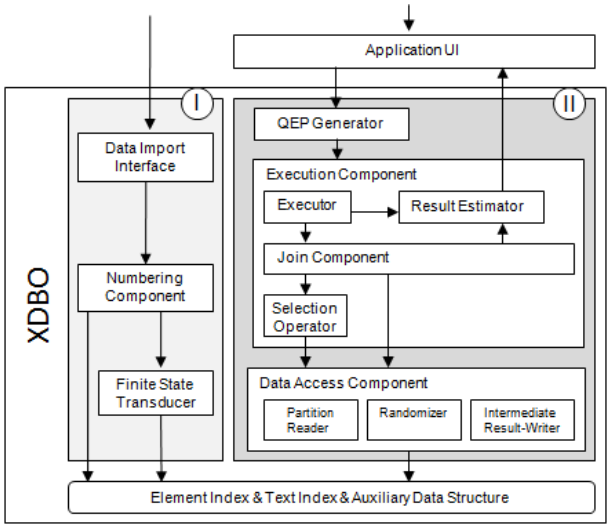

Fig. 6. Architecture of the XDBO System

Component and a Finite State Transducer. The Data Import Interface is used to load XML data into the system. The Numbering Component creates an element path index based on the Finite State Transducer that is set up for an XML document or a predefined XML schema. For each element this index lists the EDN lables of all nodes of this element type. Additional text indexes are created that store the values of the text nodes.

Database System. The database system follows a layered architecture. It consists of three components: a Query Execution Plan (QEP) Generator, an Execution Component and a Data Access Component. A user interface for posting queries against the system using structural XML query languages such as XQuery or XPath can be realized on top of XDBO. The QEP Generator accepts a structured query pattern as input and generates a QEP; it converts the given query pattern into join trees of query paths. The QEP is than handed on to the Execution Component. Additional query optimization is possible but is left for future work. The Executor Component manages the processing of the given join tree. Joins are executed in a levelwise fashion as described in Section 3.2. On-the-fly records produced throughout the query execution are used to estimate the final aggregate with the help of the Result Estimator. All operators retrieve and store data through the Data Access Component. Due to memory restrictions, data processing cannot be performed in a single run but is done in a partitioned fashion and in random order guaranteed by the Randomizer.

4.2 Prototype

We implemented the main concepts of the XDBO System in a Java prototype. As shown in the next section, this prototype is able to demonstrate the feasibility

and the efficiency of our solution. However, it is designed as proof of concept and thus limitted in its functionality. In more detail, the prototype has the following restrictions:

- XDBO evaluates value predicates with additional (inverted) indexes and allows to conjunctively or disjunctively combine those predicates. However, in the prototype we only implemented the element path index for the structural information. As a consequence, the prototype only answers queries without value predicates.
- Another restriction regarding query types concerns group-by queries which can be handled in XDBO by concurrently running a query for each group separately. This feature has not yet been implemented in the prototype.
- While XDBO in general can handle indexes over multiple documents our prototype currently only supports indexes for one (large) XML document.
- We picked out `COUNT` as the aggregation operator.
- We did not yet put much effort in the randomization process. Instead of a randomized access or an effective index structure that guarantees semi-randomness, we used sampling to select a random subset of the element labels. All the labels that have been omitted by the first sample are buffered in an auxiliary data structure for later runs.

5 Extension: Optimization

While the main principles applied in the design of the XDBO System ensure scalability and performance of the overall system, we identified a number of optimization opportunities—both general for the XDBO System and specific for our prototype. Table 1 gives a brief summary of the optimization approaches, their possible realization and the key optimization goals. Our analysis mainly focused on the following issues (last three columns of the table):

- How can we increase accuracy and thus minimize the relative confidence interval width?
- Which techniques help to minimize the total execution time?
- How can the time until the first estimate is returned be reduced?

A main factor for the first goal—the quality of the estimates—is the number of runs. The accuracy of the individual estimates decreases if the data is split into many runs. A minimum number of runs, each of which fits into memory, provides the best estimates with small confidence intervals. Hence, to realize this optimization, we need a good estimate of the number of runs based on the expected memory requirements. This optimization also reduces the total execution time.

To speed up query processing in general we can use standard optimization techniques such as pruning and data statistics. The query optimization may further incorporate properties of the Dewey numbering, e.g., all element labels that are either too long or too short to match the query pattern can be eliminated without further consideration.

Optimization	**Realization ideas**	**Minimization of**		
		relative interval width	total execution time	time until first output
Improve estimation	Minimization of number of runs	x	x	-
Speed up query processing	Pruning, data statistics, execution plan optimization	x	x	-
Optimize in-memory join of 1st levelwise step	*Mini-chunks* [7]	-	-	x
Efficiently ensure random input order	Sophisticated index structures like [7]	x	x	x

Table 1. Optimization approaches

With a growing size of individual runs the time to produce the first estimates during the first levelwise step increases. To reduce the amount of time a user waits for the first feedback [7] suggest the usage of so-called *mini chunks*: Rather than reading the full first run into memory for all query patterns, the runs for one of the query patterns are split into k equal-sized mini-chunks. These mini-chunks are then sorted and joined as in the regular processing. Since less data has to be read and joined at the beginning, the startup time will be reduced.

Currently, the main bottleneck of the XDBO System is caused by ensuring random input order. XDBO samples the data every time a new run is being read. This technique impacts the total execution time as well as the time interval needed to produce first output. Jermaine et al. examined special index structures to efficiently handle the random input order problem [7]. Similar techniques could be used for the XDBO System to boost the overall performance.

6 Evaluation

For the evaluation of the XDBO System, we used different sized XML documents (113 MB to 11.1 GB) generated with the XMark [13] data generator (scaling factors 1 to 100) and compared main characteristics of XDBO with a scalable implementation of the TwigStack System. All experiments were conducted on a 2.4 GHz processor with 4 GB RAM running a 32-bit Windows Vista Enterprise operating system. The evaluation focused on the following questions:

- What are the implications of using the extended Dewey numbering scheme on the index generation time and the memory requirements? Is there a significant drawback with respect to traditional systems?
- How fast can the XDBO System produce first results for aggregation queries?
- How are the estimates and the related confidence intervals evolving over time? Does the confidence interval decrease over time and how fast?

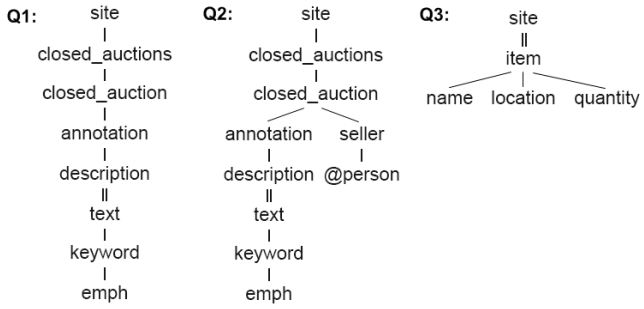

Fig. 7. Query patterns Q1-Q3

– How does the total execution time compare with traditional XML query processing systems?

We used several `COUNT` queries with different characteristics (selection rate, size of intermediate results) and picked out three of them for this evaluation; the respective query patterns are shown in Figure 7. Query Q1 has a high selectivity (7% of all *closed_auction*-elements and 1% of the *emph*-elements) and represents a query pattern without branching nodes. Query patterns with branching nodes are evaluated with the queries Q2 and Q3, both featuring different number of query path patterns and selectivities.

6.1 Index Evaluation

We first evaluate the index performance. Both the XDBO index and the TwigStack index are realized as simple text files. Figure 8(a) shows the time for building the complete index for different XMark document sizes. For XDBO, the build-up time is evaluated as the sum of the time spent on numbering and performing all necessary pre-scans. If schema information is available, as it is the case for the XMark documents, the pre-scan can be skipped and yielding a significant performance improvement. For TwigStack, the time needed to sort the labels was not taken into account but should be minded as additional overhead. As can be seen, the time for the index build-up linearly increases with the document size for both systems. In the case that schema information is available the XDBO System will create the index in a time comparable the TwigStack System; the overhead is roughly given by a factor 1.2. Given the benefits of the EDN labeling for query execution, this overhead is negligible.

In addition to the index creation time, we analyzed the memory requirements. Figure 8(b) shows the index size as a function of the document size. Surprisingly, the XDBO index needs less space than the TwigStack index. We attribute this to the fact that the TwigStack index represents the labels as strings. We further compared the index size for a single XML document with sets of multiple XML documents (each of them approx. 11MB). Figure 8(b) shows the case of all multiple XML documents stored in one single index with an additional three-character

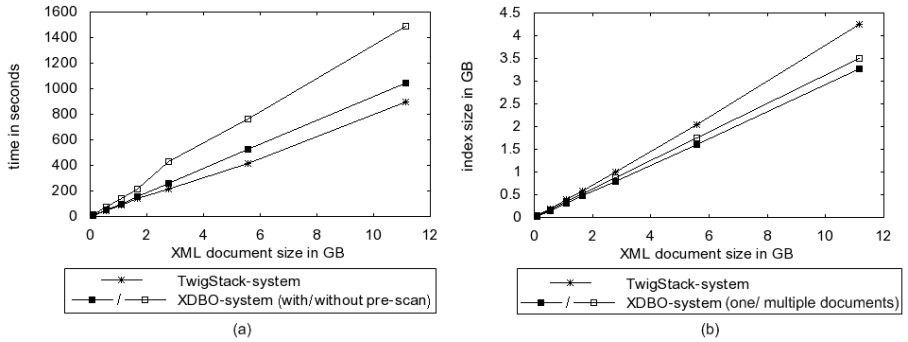

Fig. 8. Time and memory requirements for the element-label-index

document id added to each label. Since the EDN labels itself are getting smaller the total size does not increase significantly. If the multiple XML documents were stored in separate indexes we would even save memory.

6.2 Query Processing Evaluation

To evaluate the query execution performance we ran all three `COUNT` queries on the different sized XMark documents. We ran each query three times and—as each execution of a query will result in a different estimation chain which prevents averaging the results—we picked the one with the medial total execution time.

Confidence Interval ratio. First, we analyzed the relative confidence interval width defined as the ratio of the 95% confidence interval width and the query estimate. Figure 9(a) shows the relative confidence interval width as a function of the processing time for a 2.8 GB XMark document (XMark scaling factor 25). A value of 0.1 implies that the 95% confidence interval equals 10% of the current estimate. The relative interval width decreases with time for all queries. There are significant differences in the size of the relative interval widths. The very small confidence interval of the query Q1 demonstrates very accurate estimates. In comparison, the queries with branching nodes (Q2 and Q3) show larger relative interval widths, especially at the early phase of query processing. As discussed in Section 5, the relatively wide confidence intervals are due to the non-optimal size of the runs and the (simple) sampling-based randomizer and unnecessary intermediate results result in an improvable performance.

Comparison of XDBO and TwigStack. Figure 9(b) shows how the XDBO query execution compares with the TwigStack System for the 2.8 GB XMark document. The TwigStack System returns the exact query result while XDBO outputs early estimates and error bounds as well as the exact result at the end of the query evaluation. For all queries the XDBO System was able to give first output before the TwigStack System did. For the query without branching nodes (Q1) the XDBO System not only provided fast accurate estimates but also finished before the TwigStack System was able to generate an

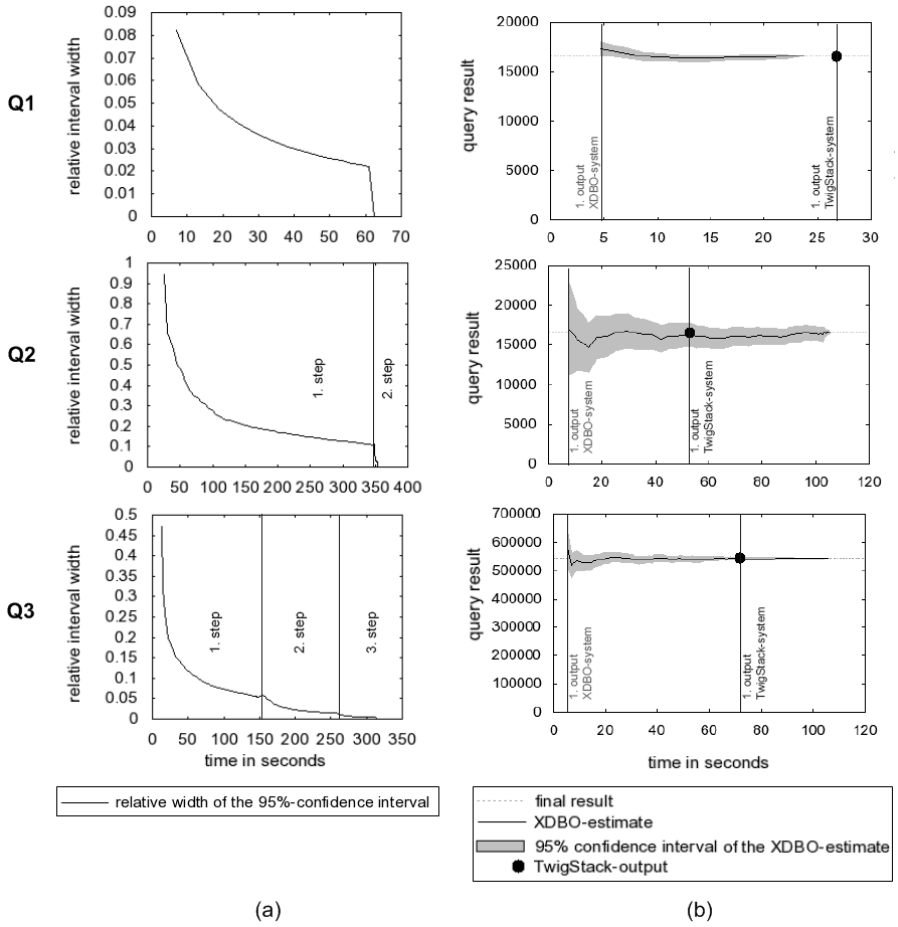

Fig. 9. Time evaluation of relative confidence interval width and comparison of XDBO and TwigStack System (for 2.8 GB XMark document)

answer. Moreover, an aggregate with a relative confidence interval width lower than 10% was produced in less than 20% of the time needed by the TwigStack System to yield the result. The queries with branching nodes required longer total execution time. In addition, and as already discussed, the confidence intervals are wide at the beginning of query processing (especially for query Q2). Still, the estimates converged to the final result in acceptable time for all queries.

Impact of document size. Next, we analyzed the performance of the XDBO System over XMark documents of different size. Figure 10 displays how much time elapsed until the first output, the first output with an relative confidence interval width lower than 10% and the final result was returned. The graphs show that for all queries the time for the first output slightly increases with the document size. Ignoring the overhead of the sampling process, exper-

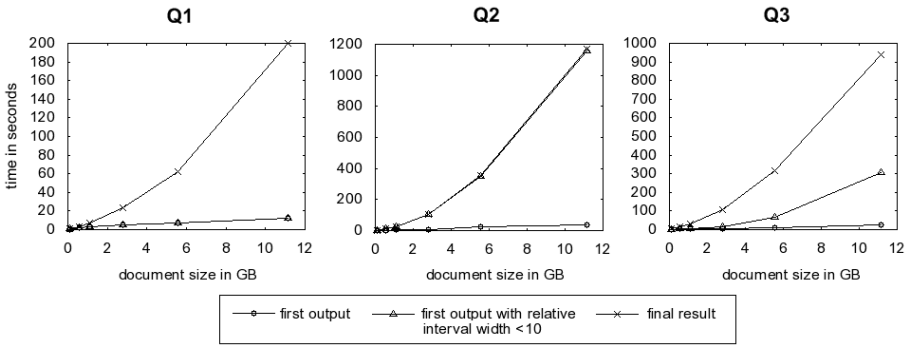

Fig. 10. Query processing over different sized XMark-documents

iments have shown that the time does not increase once the runs reached the maximum size that fits into memory. Furthermore, the XDBO System was able to generate a first accurate output for small XML documents very soon. With increasing document size the behavior varied for the different queries. For Q1 the first output, independent of the document size, had a relative confidence interval width lower than 10%. All other queries ran longer until the relative confidence interval width drop below 10% of the overall estimate. For query Q3, the output time increased with document size but XDBO was able to generate accurate output clearly before the final result was calculated. In comparison, the performance of Q2 declined more heavily with increasing document size. Here, the XDBO System produces first output very fast while the time to achieve the exact result clearly increases. The reasons for this behavior have been analyzed in detail and also solution approaches have been suggested.

Impact of optimization Finally, we present the performance impact that we achieved by minimizing the number of runs for query Q2. The other optimizations had not been examined so far, but seem to offer great opportunities. Our prototype roughly estimates the number of runs based on the expected memory requirements derived from the maximal depth $maxDepth$ of the XML tree (utilizing the fact that the label length depends on the element depth). We observed that this argument often overestimates the memory consumption as typically only a fraction of the elements are located at the lowest level of the XML tree. We analyzed the impact of basing the estimates upon $w \cdot maxDepth$ for different $w < 1$ instead of just $maxDepth$. Figure 11 shows the effect of this modification for query Q2 on the 2.8 GB XMark document for $w = 0.25$. The results for the original calculation are depicted in (a), the optimized computation is displayed in (b). As can be seen, the relative confidence interval width significantly decreases. This way the XDBO System is able to provide fast accurate estimates even for queries with branching nodes. Additionally, we could reduce the total execution time significantly. As a drawback, the time period to provide a first estimate increases. As described in Section 5 this can be attributed to the larger runs and can be compensated by the usage of mini-chunks [7].

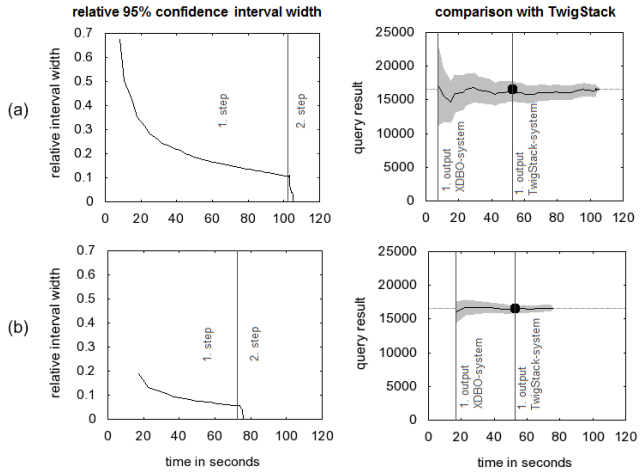

Fig. 11. Impact of minimizing the number of runs for Q2: original (a) & optimized (b)

7 Related Work

In this section, we give an overview over the different areas of related work.

Online Aggregation. Online Aggregation was introduced by Hellerstein et al. in [6]. A major technical challenge in OLA is to combine efficient join processing with unbiased guaranteed accuracy estimates. To address this task, various algorithms such as the Ripple Join [5] or the SMS Join [8] have been proposed. Jermaine et al. [7] observed that the result inaccuracy for these algorithms significantly increases with the number of tables to be joined; their DBO System ensures scalable query processing for OLA by sharing information across relational operations at different levels of the query plan. Especially from the DBO System we adopted some concepts; however, we made significant effort to adapt these concepts to the fairly different style of query processing of XML data.

Indexing and numbering schemes for XML data. Indexes are a well-known technique to speed up query processing; clearly, this also holds for XML data as shown in [11]. Additionally, to speed up pattern matching a multitude of numbering techniques and algorithms have been proposed like prefix- [12] and interval-based [1] numbering schemes. Interval-based numbering methods assign ranges to the elements and indicate structural relationships via containment relationships. Prefix-based numbers encode children and descendant relationships through equal prefixes. The extended Dewey numbering scheme [10] additionally encodes the complete root-to-node paths into the labels; a finite state transducer allows for an efficient encoding and decoding. We incorporate the EDN for highly efficient path pattern matching.

Pattern matching. Recent algorithms for XML pattern matching exploit the characteristics of various numbering schemes to significantly improve the processing speed of structural joins. The first proposed structural join operations [1,

15] focus on binary query patterns; PathStack [2] extends the analysis to query path patterns. These solutions suffer from the need of additional stitching steps when applied to more complex query patterns. To process query patterns as a whole Bruno et al. presented the holistic TwigJoin algorithm [2]. Further optimizations of this approach have been done in [4, 9]. While some of the proposed pattern matching algorithms are non-blocking they generally rely on the processing of labels in a sorted order. Accordingly, none of these algorithms supports early result feedback and processing with guaranteed statistical error bounds. Looking at the numbering techniques, most of the proposed solutions are based on an interval-based numbering scheme. Like our solution, TJFast [10] is based on a prefix-numbering scheme; it exploits the features of the extended Dewey numbering to significantly speed up query processing. However, as it is based on a lexicographical sorted and moreover blocking processing of the labels it does not meet the OLA requirements.

Synopses. Alternative approaches to support early feedback are synopses and data summaries [3, 14]. These methods rely on precomputed data structures that only cover statistical properties of the overall data set. They do not support the paradigm of query processing that improves with execution time and that eventually converges to the exact result.

8 Conclusion

In this paper, we presented the concepts and the architecture of a system capable of performing Online Aggregation over XML data. This XDBO System is able to give fast feedback to aggregation queries by approximating and refining the final answer throughout query processing. Additionally, it provides accuracy guarantees by attaching confidence information to the estimates. We introduced a novel query processing approach that splits query patterns into query path patterns; efficient query processing is realized by novel operators for selecting and joining path patterns in combination with an appropriate index structure. For accurate estimates we adapted principles of the DBO engine to the XML query processing. We prototypically implemented the XDBO System to demonstrate the efficiency of our solution. Within our extensive evaluation, we have shown that our system returns accurate guesses of the final answer long before traditional systems are able to produce output. Furthermore, good estimates are gained very fast for query patterns without branching nodes. We identified some limitations for more complex queries, but presented and demonstrated first optimization approaches to address these limitations.

Our future work will focus on the optimization and the extension of further functionalities. In more detail, we will address the following questions:

- *How can a random input order efficiently be guaranteed?* Here, a sophisticated solution will significantly speed up query processing.
- *What is the potential of other XML numbering schemes?* While we have shown that the extended Dewey numbering scheme is highly efficient and

only entail low overhead looking at other numbering schemes in more detail might be promising.
- *How can we efficiently support XML updates without renumbering the whole XML document?* We did not address this issue yet but we will do to make the XDBO an even more sophisticated solution.
- *Which additional optimizations can be utilized?* We already focused on this question; however, there is sill potential, e.g. by handling queries with high join selectivities—and thus, with many unnecessary intermediate results—with special care.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE '06*, pages 141–152, 2002.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD '02*, pages 310–321, 2002.
3. K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *VLDB '00*, pages 111–122, 2000.
4. S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. C. Twig2stack: bottom-up processing of generalized-tree-pattern queries over xml documents. In *VLDB '06*, pages 283–294, 2006.
5. P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.
6. J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD '97*, pages 171–182, 1997.
7. C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, 33(4):1–54, 2008.
8. C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *SIGMOD '05*, pages 563–574, 2005.
9. Z. Jiang, C. Luo, W.-C. Hou, Q. Zhu, and D. Che. Efficient processing of xml twig pattern: A novel one-phase holistic solution. In *DEXA '07*, pages 87–97, 2007.
10. J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended dewey: on efficient processing of xml twig pattern matching. In *VLDB '05*, pages 193–204, 2005.
11. J. McHugh and J. Widom. Query optimization for xml. In *VLDB '99*, pages 315–326, 1999.
12. V. Sans and D. Laurent. Prefix based numbering schemes for xml: techniques, applications and performances. *Proceedings of the VLDB Endowment*, 1(2):1564–1573, 2008.
13. A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: a benchmark for xml data management. In *VLDB '02*, pages 974–985, 2002.
14. J. Spiegel, E. Pontikakis, S. Budalakoti, and N. Polyzotis. Aqax: a system for approximate xml query answers. In *VLDB '06*, pages 1159–1162, 2006.
15. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD '01*, pages 425–436, 2001.