QuickXDB: A Prototype of a Native XML DBMS*

Petr Lukáš, Radim Bača, and Michal Krátký

Department of Computer Science, VŠB – Technical University of Ostrava
Czech Republic
{petr.lukas, radim.baca, michal.kratky}@vsb.cz

Abstract. XML (extensible mark-up language) is a well established format which is often used for semi-structured data modeling. XPath and XQuery [16] are de facto standards among XML query languages. There is a large number of different approaches addressing an efficient XQuery processing. The aim of this article is to introduce a prototype of an XML database called QuickXDB integrating these state-of-the-art approaches. We primarily describe main concepts of QuickXDB with stress on our XQuery processor. Furthermore, we depict main challenges such as identification of patterns in an input query. Our experiments show strengths and weaknesses of our prototype. We outline our future work which will focus on a cost-based optimization of a query plan.

1 Introduction

XML data model is often considered as a useful alternative to a traditional relational data model. XML data model is more flexible and maintenance of an XML structure is more simple if we work with a very dynamic structure of a database. Therefore, an efficient XML database can be a very useful tool in many real-world projects. Many different approaches have been developed in recent years [1, 5, 6, 12, 18], however, putting all these ideas into a working XML database represents a challenging task with unsolved problems.

Query languages for XML databases such as XPath and XQuery [16] are considered as a de-facto standard. Our work focuses on an efficient processing of XQuery which includes semantics of XPath. There exist several query models which express a core functionality of XQuery. A twig pattern query (TPQ) is the most simple model used by many approaches [1, 5, 9, 17, 19]. A TPQ is represented by a rooted labeled tree (see the TPQ in Figure 1(c)). There is also a more general query model called GTP which includes more semantics of the XQuery language, such as semantics related to the output formatting, boolean expressions or optional parts of a query. These query models represent an important abstraction which enabled designing many efficient algorithms [1, 5, 9, 17, 19] that are independent of a semantic details of an XML query language. However, correct identification of a TPQ or GTP in a XQuery is not straightforward [12].

* Work is partially supported by Grant of SGS No. SP2013/42, VŠB-Technical University of Ostrava, Czech Republic.

V. Snášel, K. Richta, J. Pokorný (Eds.): Dateso 2013, pp. 36–47, ISBN 978-80-248-2968-5.

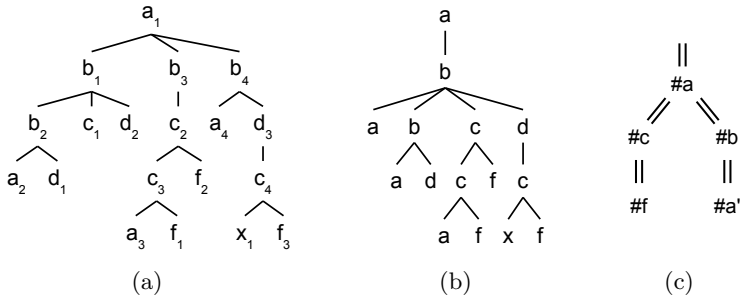

Fig. 1. (a) XML tree (b) DataGuide (c) TPQ

We can find three main areas in the XQuery processing: (1) index data structures and XML document partitioning, (2) join algorithms for a TPQ/GTP processing, and (3) query algebras and cost-based optimizations. Techniques from different areas are often closely related. For example, join algorithms usually need a specific type of index for its efficient run. In this paper, we describe a prototype of an XML database called QuickXDB from the perspective of all these three areas. We describe key concepts of our prototype which enable fast XQuery processing. Main challenges addressed by our prototype are as follows:

- Implementation of an XQuery algebra enabling creation of a query plan, query plan rewritings, and transformation to a physical query plan;
- correct identification of TPQs (more precisely GTP) in a query plan and incorporation of fast and optimal algorithms for the GTP processing;
- cost-based optimizations of a physical query plan that are dependent on XML document statistics as well as on indices available in a database.

The paper is organized as follows. In Section 2, we define basic terms. In Section 3, we summarize basic types of data structures and indices that are available in our database. Section 4 introduces main concepts included in our XQuery processor. In Section 5, we show results of our experiments where we compare efficiency of QuickXDB with other common XML databases. In Section 6, we summarize our results and outline our future work on our prototype of an XML database.

2 Preliminaries

An XML document can be modeled as a rooted, ordered, labeled tree, where every tree node corresponds to an element or an attribute of the document and edges connect elements, or elements and attributes, having the parent-child relationship. We call such a representation of an XML document an *XML tree*. In what follows, we simply write 'node' instead of the correct 'tree node' or 'data node'. An example of the XML tree is shown in Figure 1(a).

The DataGuide tree [8] is a labeled tree where every labeled path from the XML tree occurs exactly once there. Figure 1(b) depicts an example of a DataGuide for the XML document in Figure 1(a).

We assign a label to every node of an XML tree. Node labels allow us to resolve basic XPath relationships between two nodes during the query processing. There are two types of labeling schemes: (1) *element scheme* (e.g., containment scheme [19] or Dietz's scheme [7]) and (2) *path scheme* (e.g., Dewey order [15]). The main feature of labels using a path labeling scheme is that we can extract labels of all ancestors and that they are more flexible during updates.

A *twig pattern query* (TPQ) can be modeled as an ordered rooted tree. Single and double edges between two query nodes represent parent-child (PC) and ancestor-descendant (AD) relationships, respectively (see an example of a TPQ in Figure 1(c)).

3 Indices

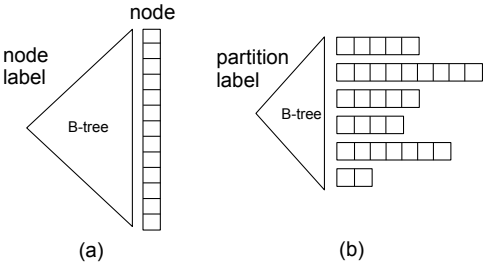

Fig. 2. (a) Document index (b) Partition index

There are two basic type of indices: (1) that having a node label as a key (document index) and (2) that having a node name as a key (partition index). Nodes corresponding to one node name are sorted according to the node label in the case of a partition index. An illustration of these indices can be found in Figure 2.

From the query processing perspective, a document index is very useful if we have a very small intermediate result and we want to use it to process the rest of a query. This type of the query processing can be considered as navigational [11]. However, many join algorithms are based on a partition index. These joins mainly focus on a merging during one sequential read of lists which removes irrelevant nodes. Selection of an appropriate index will be part of future cost-based optimizations [2].

QuickXDB contains also DataGuide, that can be used to speed up the query processing. The main idea is to preproccess the TPQ in a DataGuide and use the result to decrease a search space of a partition or a document index [3].

4 XQuery algebra

In this section, we present a brief introduction to the techniques we use to evaluate real XQuery queries, not only stand-alone TPQs. We use a modified XQuery algebra proposed in [13].

4.1 Processor architecture

There is a block schema of the processor in Figure 3. The traditional first step is to load and parse the input query. The first step yields a syntax tree[1] based on the XQuery grammar standardized by W3C. The next conventional step is usually a normalization phase [12], but we adapted the compilation rules so that the query is directly compiled into the tree of operators (see Figure 5). At the current prototype state, each operator has exactly one evaluating algorithm, so the operator tree has the same meaning as a physical query plan and we call it simply a *query plan*. However, a modular architecture of the prototype does not give us any limitations of using cost-base optimizations to select one of more low-level algorithms evaluating an operator in the future (see Section 6).

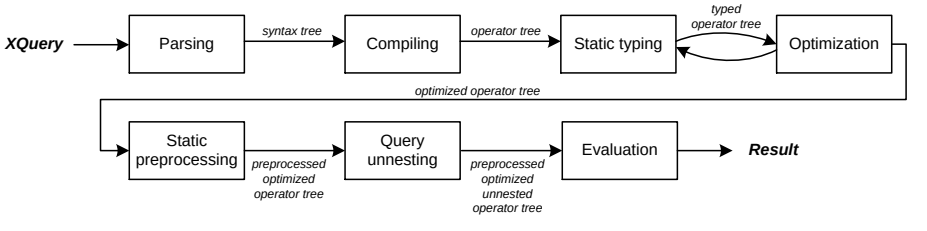

Fig. 3. Block structure of the processor

A compiled query plan is statically typed. The static typing is an important feature for some of the rewriting rules in the next optimization phase (see Section 4.5). None of static pre-computations uses the input XML document. During the optimization a set of rewriting rules is repeatedly applied on the query plan. Each rule searches for a specific pattern with some specific properties and replaces it by a single operator or a subtree of operators. This phase is also responsible for searching the largest possible TPQs in the query plan (see Section 4.6). The crucial attribute of all rewritings is that they are transparent from the result perspective. After applying a rewriting rule, output types of operators have to be reevaluated.

Before the evaluation is done, all the operators are statically preprocessed. For example, we can evaluate a pointer to the algorithm of a function call according to the name of the function. Before the final step we also perform a

[1] Syntax tree is an ordered, rooted, labeled tree, where the nodes stand for terminal / non-terminal symbols of a formal grammar of a language such as XQuery.

query unnesting (see Section 4.4). Finally, the optimized and unnested query plan is evaluated.

4.2 Algebra

A detailed description of an algebra which we use can be found in [13]. Here we only mention some important characteristics to make the following examples clear.

The algebra operates over two different types of sets: (1) a set of *sequences* containing *items* (atomic values or XML nodes), (2) a set of *tables* containing *tuples*. Tuple components are formed of single items or sequences. Furthermore, we have three groups of operators. Operators over sequences, operators over tables, and hybrid operators over both types of sets.

Each operator can have three groups of arguments: independent suboperators, dependent suboperators, and static attributes. Evaluation of the independent suboperators is usually the first step of evaluation algorithms of all operators. The purpose of the dependent suboperators depends on a purpose of particular operator. For example, the *Selection* operator has one independent suboperator computing its input table and one dependent suboperator deciding which tuples of the input table will be passed to the output.

Example 1. Let us consider the example in Figure 5. There are two possible plans of the Q1 query from Figure 4. Both plans lead to the same result. P1a is obtained after the optimization phase without using TPQ rewritings. If we include TPQ rewritings, we obtain the P1b plan. If we run the query against the XML tree from Figure 1, we get the XML node c_1 as the result.

Q1
```
for $x in //b, $y in $x//c
where $x//d and not($y/x)
return $y
```

Q2
```
for $x in //closed_auctions/closed_auction
for $y in //people/person
where $x/buyer/@person = $y/@id
return <out> { $x/type/text() }, { $y/name/text() } </out>
```

Q3
```
for $x in //item
where $x/@id = max(//item/@id)
return $x/name
```

Q4
```
for $x in //europe/item[mailbox/mail/date = "08/05/1999"]
  [description//parlist//parlist][1]
return $x
```

Fig. 4. Example queries

Now let us discuss the P1a plan to outline how the evaluation works. The instances of operators are distinguished by subscripts. Static attributes of operators are given in square brackets. Solid lines represent bindings to the independent suboperators, dashed lines stand for bindings to the dependent ones. The intermediate results of key operators can be seen above them.

The subtree starting with $MapFromItem_1$ represents the first `for` loop of the query Q1. It computes a single-column table with all XML nodes of the name b from the input XML. The $Select_1$ stands for the `$x//d` part of the `where` clause.

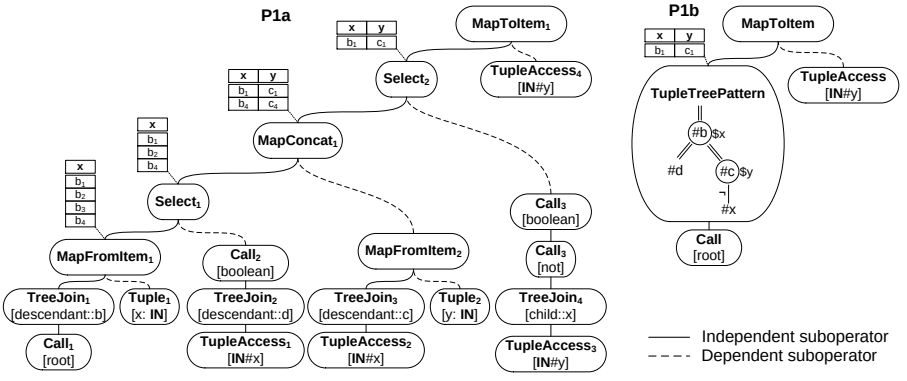

Fig. 5. Example of query plans

Select$_1$ restricts the table from the previous step to only those rows where there exists a d descendant for a particular b. The *MapConcat*$_1$ operator represents the second loop of the `for` clause. *MapConcat* is a crucial operator for this algebra. It evaluates its dependent suboperator for all tuples of the independent input table and joins the evaluated semi-results to the tuples of which they have been computed. Then we can evaluate the `not($y/c)` part of the `where` expression using the *Select*$_2$ operator. The final result is evaluated by the *MapToItem*$_1$ operator. It selects all values of the y column of the input table since we refer to the `$y` variable in the `return` clause.

4.3 Relational rewritings

We can see in the P1a plan that there is no operator evaluating the *and* boolean expression of the `where` clause. The *and* expression is divided into two separated selections: *Select*$_1$ and *Select*$_2$. We can push the selection operators in order to evaluate them as early as possible. This is a well-known rewriting rule from the relational databases. Application of such rule has two advantages: (1) evaluation of the selection as early as possible speeds up evaluation of the entire query, (2) this rewriting extends the possibilities of locating TPQ patterns in the query plan.

There are some other important relational rewritings enabling us to use traditional join operators. In the Q2 query, we can see two independent `for` loops and a typical joining `where` clause. If we use some of the advanced joining algorithms such as hash-join or merge-join, we can significantly speed up the query evaluation. Selection of the proper joining algorithm can be ensured by a cost-based optimization (see Section 6).

4.4 Query unnesting

Note that Q3 is a simple query returning a name of an item with the maximum id. The query can be run over the XMark testing documents [14]. The `where`

clause of Q3 will be compiled into a *Selection* operator. Such operator evaluates the restricting condition over all input tuples carrying individual items in our case. There is a relatively complicated expression computing the maximum id of all items in the right hand side of the equality comparison. Query unnesting means that since an expression is not dependent on any context value (e.g., the `$x` variable of the `for` clause), it can be evaluated only once because its resulting value is always the same.

4.5 Static typing

The static typing step evaluates types of output values of operators in a plan wherever it is possible. Static typing is a crucial feature making some important static rewritings possible. For example, Q4 from Figure 4 contains all three possible types of XPath predicates[2]. The first predicate selects items matching a given equality boolean expression. The second one selects such items where a sequence of XML nodes in the predicate is not empty. The third one passes only the first item matching the above two predicates. Since the purpose of a predicate depends on the type of its expression, there has to be an alternative construction in the unoptimized query plan taking the type into account. However, the output type can be statically evaluated in many cases. Therefore, we are able to rewrite the query plan and use directly one branch of an alternative operator.

Moreover, since there is a possibility to query the order of a node (the last predicate), we need to add (or more technically *map*) an extra column containing sequential numbers. Such column represents a special *positional variable*. If we simplify the query plan knowing we surely do not access the positional variable (the first two predicates), we are able to remove the positional mappings. There are several other rewritings using the statically pre-evaluated types. The goal is to simplify the query plan in order to be able to locate patterns that can be rewritten into a form of TPQ.

4.6 TPQ rewritings

There is still a gap between XQuery algebras and algorithms evaluating TPQs. A lot of algebras have been proposed, but the tree patterns are not supported by them. Only several approaches such as [12] are oriented to search tree patterns in XQuery queries.

A static rewriting of a query plan is the only method how we detect TPQs or more precisely GTPs. Unlike [12], we do not perform any rewritings of the query before the compilation phase. There is a set of rewriting rules searching for the largest possible subtree of operators in the query plan and replacing it by a single tree pattern. A special operator *TupleTreePattern* ensures the use of holistic join algorithms. This operator has been already proposed in [12], but our implementation uses a GTP as its static attribute instead of a linear sequence of path steps.

[2] An XPath predicate is a restricting condition written in square brackets.

The rewriting rules are based on two basic principles: (1) replace single *TreeJoin*[3] by *TupleTreeJoin* with simple tree patterns, (2) merge individual *TupleTreeJoin*s together.

Let us consider the TPQ of the *TupleTreePattern* operator in the P1b plan. All output (circled) query nodes have a reference to the corresponding component of output tuples denoted by $\$q$, where q is the name of the component. We can see that the single *TupleTreePattern* operator in the P1b plan is able to provide the same functionality as the whole subtree of the $Select_2$ operator in the P1a plan. Since the *TupleTreePattern* uses the GTPStack [4] holistic algorithm, the evaluation of P1b is much more efficient for large XML documents than the evaluation of P1a.

5 Experiments

In this section, we compare QuickXDB with some other commonly used XML databases. We choose two standard relational databases Oracle 11g[4] and Microsoft SQL Server 2012[5], three native XML databases Monet DB4[6], Saxon 9.4[7] and BaseX4[8]. Both Microsoft SQL Server and Oracle database have the possibility of indexing XML. Oracle database supports one type of XML index, SQL Server supports one type of primary and three types of secondary XML indices. Also BaseX provides optional XML indexing. Our QuickXDB is able to work with or without any XML index and persistent data structures. In what follows, we write *indexed* or *non-indexed* QuickXDB.

For testing we choose XMark (1.1 GB) [14] and TreeBank[9] (86 MB) data collections and 20 XQueries (10 queries per each collection). A complete set of chosen XQueries can be found in [10]. The queries were divided into two groups according to their purpose: (1) structure oriented queries and (2) content oriented queries.

All experiments were performed on a machine with Intel Xeon E5-2690@2.9GHz processor and Microsoft Windows Server 2008 R2 Datacenter (SP1) operating system. The time of a query execution was the main factor we were focused on.

Every query was run 5 times for each database and each indexing variant. A measured time includes both query execution and query preprocessing (parsing, compiling, etc.). The results are given in Table 1. Each value represents an arithmetic mean of 3 measured times (without the worst and the best case) in seconds.

[3] A *TreeJoin* operator performs elementary path steps in an XML document.
[4] `http://www.oracle.com/products/database/`
[5] `http://www.microsoft.com/sqlserver/`
[6] `http://monetdb.project.cwi.nl/monetdb/XQuery/`
[7] `http://saxon.sourceforge.net/`
[8] `http://www.basex.org/`
[9] XML Data Repository, `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

	Oracle wo/index	Oracle w/index	SQL Server wo/index	SQL Server prim/idx	SQL Server sec/idx	Saxon	Monet DB	BaseX wo/index	BaseX w/index	Quick XDB non-idx	Quick XDB indexed
Structure oriented XMark queries											
XM1	7.509	DNF	DNF	201.57	DNF	0.374	0.171	1.129	1.203	-	0.066
XM2	DNF	DNF	DNF	DNF	DNF	MEM	E	DNF	DNF	-	0.076
XM3	DNF	DNF	DNF	DNF	DNF	MEM	E	DNF	DNF	-	E
XM4	29.272	DNF	DNF	DNF	DNF	0.837	0.52	1.468	1.444	-	0.47
XM5	24.067	DNF	DNF	DNF	DNF	0.853	0.676	1.843	1.945	-	0.477
Content oriented XMark queries											
XM6	DNF	DNF	DNF	DNF	DNF	DNF	0.394	DNF	18.877	-	10.051
XM7	E	DNF	-	-	-	2.694	-	DNF	DNF	-	8.86
XM8	DNF	DNF	DNF	DNF	51.6	52.942	0.226	252.459	3.141	-	6.271
XM9	E	E	DNF	DNF	DNF	9.048	2.079	9.044	9.409	-	3.829
XM10	DNF	DNF	-	-	-	DNF	0.184	DNF	DNF	-	E
Structure oriented TreeBank queries											
TB1	8.345	DNF	DNF	DNF	DNF	0.385	0.317	1.799	1.808	0.93	0.043
TB2	8.233	DNF	DNF	DNF	DNF	0.494	0.313	2.543	2.655	0.504	0.027
TB3	84.59	DNF	247.662	116.554	358.584	0.281	0.335	11.083	11.246	1.13	0.045
TB4	11.561	DNF	DNF	DNF	148.258	0.499	0.447	2.448	2.239	0.695	0.079
TB5	2.199	DNF	7.072	2.101	407.059	0.759	0.234	1.880	1.724	0.66	0.38
TB6	19.895	DNF	E	E	E	0.25	0.33	7.385	7.242	0.493	0.427
TB7	10.947	DNF	E	E	E	1.492	0.383	8.464	8.308	0.846	0.74
TB8	DNF	E	-	-	-	0.374	1.336	3.058	3.228	0.947	DNF
Content oriented TreeBank queries											
TB9	E	DNF	-	-	-	1.347	0.358	4.897	5.104	0.479	1.746
TB10	1.08	DNF	4.233	2.303	2.189	0.26	0.332	1.236	1.264	0.725	0.52

Table 1. Execution times [s]

The DNF symbol means that a query execution exceeds 10 minutes, E means that a query execution finished with an error. The MEM shortcut stands for the cases when we had to manually stop a query execution due to the protection of the server operating system because of using unacceptably high amount of an operating memory (over 50 GB). A hyphen mark means that a query could not be run because of an unsupported construction or a database was not able to load a data collection.

5.1 XMark queries

The 1GB XMark collection is a representative of large XML documents. This kind of documents are a problem for memory-oriented[10] processors such as Saxon and non-indexed QuickXDB. Saxon processor was able to load the XMark collection, but it consumed over 6 GB space of operating memory. Since the non-indexed QuickXDB (without using persistent structures) is limited to use up to 2 GB of operating memory, it was not able to load the entire XMark document.

Let us see the results of the structure oriented queries (XM1 – XM5) in Table 1. Oracle was able to process 3 of the queries without using index, SQL Server processed only 1 of them using primary XML index. The most problematic query seemed to be XM3 which could not be processed by any of the processors due to the unacceptable high query result.

[10] Memory-oriented processors load the entire XML document into the operating memory and do not use any persistent data structures.

We can observe that indexed QuickXDB outperforms all the other databases for every structure oriented query. The key reason is a detection of a GTP in an XQuery and application of the GTPStack holistic algorithm.

The XMark testing documents contain human-readable data, so querying content may be desirable. Since the current prototype of indexed QuickXDB do not use any content-based index, it performs many random disk accesses. That is the main reason, why the indexed QuickXDB is slower than Saxon and MonetDB in the most content oriented queries (XM6 – XM10).

5.2 TreeBank queries

TreeBank is a relatively small XML document (86 MB) with complex recursive and irregular structure. TreeBank can be loaded into the operating memory by any of the memory oriented query processors.

TreeBank does not have human-readable data so the most of the queries (TB1 – TB8) are structure oriented, where holistic algorithm ensures the fast evaluation. Only 2 of the TreeBank queries (TB9 – TB10) are content oriented, where the non-indexed QuickXDB can give a better performance when compared to indexed QuickXDB.

5.3 Comparison results

We can find four graphs showing relative results of the indexed QuickXDB compared with the other processors in Figure 6. Each value is computed as $100 - 100G$, where G is a geometric mean of relevant quotients t_{xdb}/t_{other}, where t_{xdb} is an execution time of the indexed QuickXDB processor and t_{other} is an execution time of a compared database for the same query. For instance, a value 90 means that the indexed QuickXDB runs ten times faster than the compared database.

Generally we can say that our prototype implementation of indexed QuickXDB runs evidently faster on structure oriented queries. As mentioned before, the key reason is the detection of TPQs and evaluating them by an GTPStack holistic algorithm.

6 Conclusion and future work

In this article, we describe a prototype of our XML database called QuickXDB. We outline the core data structures representing our database and we focus on a description of the XQuery algebra that support a query processing. XQuery algebra enables integration of the state-of-the-art techniques such as holistic joins with other common techniques well known from relational databases. We performed an experiment, where we compare QuickXDB with two major database systems and two native XQuery processors. QuickXDB outperforms all databases for structure oriented queries. As expected, QuickXDB was less successful for

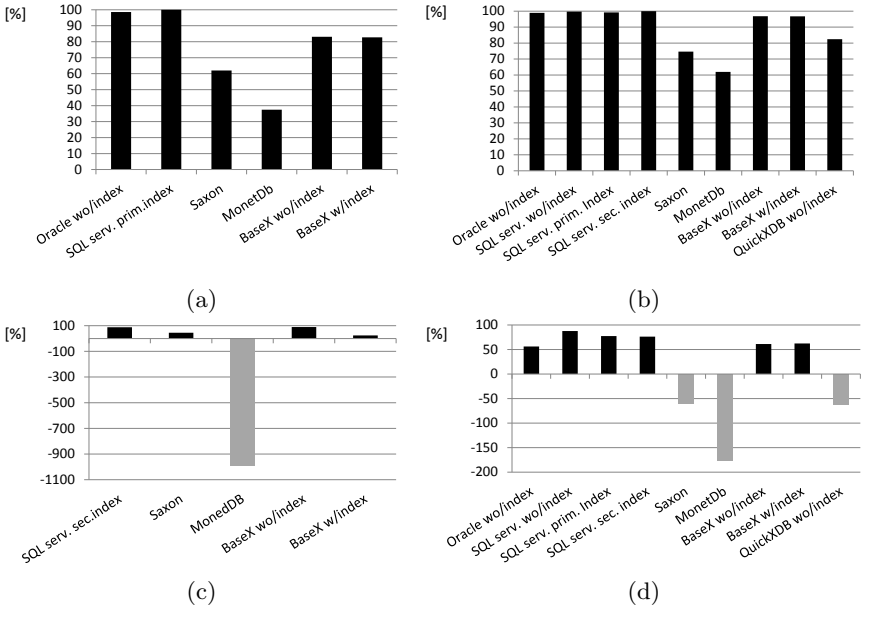

Fig. 6. Comparison of indexed QuickXDB with other XML databases. (a) Structure oriented XMark queries (b) Structure oriented TreeBank queries (c) Content oriented XMark queries (d) Content oriented TreeBank queries

content oriented queries, however, these queries can be accelerated by a common content-based index.

Our algebra contains wide set of rewritings which will support cost-based optimizations in a future extension of the prototype. The major substance missing in our solution are statistics that could help us to automatically select appropriate query plan using all available indices.

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 *Proceedings of ICDE 2002*, pages 141–152. IEEE CS, 2002.
2. R. Bača and M. Krátkỳ. A Cost-based Join Selection for XML Twig Content-based Queries. In *Proceedings of the 2008 EDBT workshop on Database technologies for handling XML information on the web*, pages 13–20. ACM, 2008.
3. R. Bača and M. Krátký. On the Efficiency of a Prefix Path Holistic Algorithm. In *Proceedings of Database and XML Technologies, XSym 2009*, volume LNCS 5679, pages 25–32. Springer–Verlag, 2009.
4. R. Bača, M. Krátký, T. Ling, and J. Lu. Optimal and efficient generalized twig pattern processing: a combination of preorder and postorder filterings. *The VLDB Journal*, pages 1–25, 2012.

5. N. Bruno, D. Srivastava, and N. Koudas. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of ACM SIGMOD 2002*, pages 310–321. ACM Press, 2002.
6. S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig2Stack: Bottom-up Processing of Generalized-tree-pattern Queries Over XML documents. In *Proceedings of VLDB 2006*, pages 283–294, 2006.
7. P. F. Dietz. Maintaining order in a linked list. In *Proceedings of 14th annual ACM symposium on Theory of Computing (STOC 1982)*, pages 122–127, 1982.
8. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB 1997*, pages 436–445, 1997.
9. J. Lu, T. Chen, and T. W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: a Look-ahead Approach. In *Proceedings of ACM CIKM 2004*, pages 533–542. ACM Press, 2004.
10. P. Lukáš, R. Bača, and M. Krátký. QuickXDB: A Prototype of a Native XML DBMS. *Technical report No. CS/DBRG/2013-001*, 2013, `http://db.cs.vsb.cz/TechnicalReports/CS-DBRG-2013-001.pdf`.
11. N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *ACM Transactions on Database Systems (TODS)*, 31:968 – 1013, September 2006.
12. P. Michiels, G. Mihaila, and J. Siméon. Put a Tree Pattern in Your Algebra. In *Proceedings of the 23th International Conference on Data Engineering, ICDE 2007*, pages 246–255. IEEE, 2007.
13. C. Re, J. Siméon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 14–14. IEEE, 2006.
14. A. R. Schmidt et al. The XML Benchmark. Technical Report INS-R0103, CWI, The Netherlands, April, 2001, `http://monetdb.cwi.nl/xml/`.
15. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of ACM SIGMOD 2002*, pages 204–215, 2002.
16. W3 Consortium. XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, `http://www.w3.org/TR/xquery/`.
17. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a Dynamic Index Method for Querying XML data by Tree Structures. In *Proceedings of the ACM SIGMOD 2003*, pages 110–121. ACM Press, 2003.
18. A. M. Weiner and T. Härder. Using Structural Joins and Holistic Twig Joins for Native XML Query Optimization. In *Advances in Databases and Information Systems*, volume 5739 of *LNCS*, pages 149–163. Springer - Berlin Heidelberg, 2009.
19. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of ACM SIGMOD 2001*, pages 425–436, 2001.