Framework-Based Development and Evaluation of Cost-Based Native XML Query Optimization Techniques

Andreas M. Weiner
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern, P. O. Box 3049
D-67653 Kaiserslautern, Germany
weiner@cs.uni-kl.de

Supervised by: Theo Härder
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern, P. O. Box 3049
D-67653 Kaiserslautern, Germany
haerder@cs.uni-kl.de

ABSTRACT

Reflecting on the history of database management systems reveals that cost-based query optimization has been the dominating method for effectively answering complex queries on large documents. Native XML database management systems provide an efficient infrastructure for storing, indexing, and querying large XML documents. Even though such systems can choose from a huge set of structural join operators and value-based join operators as well as various index access operators to efficiently query XML data, the development of powerful native XML query optimizers is just emerging. Furthermore, it is not known how the aforementioned operators behave in complex XQuery evaluation scenarios, which occur frequently in real-word applications.

The extensible, rule-based, and cost-based XML query optimization framework proposed in this work, provides a basic testbed for exploring how and whether established techniques of relational cost-based query optimization (e. g., reordering of join operators) can be reused and which new techniques have to be developed to make a significant contribution for accelerating query execution. Using the best practices and an appropriate cost model that will be developed using this framework, it can be turned into a stable cost-based XML query optimizer in the future.

1. INTRODUCTION

Native XML database management systems (XDBMSs) can only become a respected competitor for relational-based XQuery evaluation engines, if they can make the most out of the sophisticated join operators and indexes, which have been proposed in recent years. Two important classes of join operators are: *Structural Joins* (SJs) [1] and *Holistic Twig Joins* (HTJs) [3]. SJ operators decompose structural patterns (e. g., twig query patterns) into binary relationships and evaluate each of them separately. Thereafter, the intermediate results are merged to get the final query result. On the other hand, HTJs can evaluate such patterns holistically.

To enable efficient evaluation of SJ and HTJ operators, a node labeling scheme [9] is required, that assigns each node in an XML document a unique identifier that (1) permits deciding for two given nodes—without requiring further accesses to the document—whether they are structurally related to each other, and (2) that does not necessitate relabeling even after modifying the document.

Along with SJ and HTJ operators, several approaches for indexing XML documents were proposed. These methods can be partitioned into three equivalence classes: primary, secondary, and tertiary access paths. *Primary access paths* (PAPs) serve as input for navigational primitives as well as for SJ and HTJ operators. The most important representative of this class is a document index that indexes a document using its unique node labels as keys. *Secondary access paths* (SAPs) allow for more efficient access to specific element nodes using element indexes [3]. They are mandatory for efficient evaluation of structural predicates by SJ or HTJ operators. *Tertiary access paths* (TAPs) like path indexes [16] use Dataguides [4] for providing efficient access to nodes satisfying structural relationships like `child` or `descendant`. Content indexes [15] support efficient access to text nodes or attribute-value nodes. Finally, hybrid indexes [21], which are also known as content-and-structure (CAS) indexes, are a powerful mechanism for indexing content and structure at a time. Compared to PAPs, which are available per default in a native XDBMS, SAPs and TAPs have to be manually created by the database administrator, because their maintenance causes considerable overhead for the XDBMS. Moreover, TAPs like path indexes or CAS indexes become first class citizens for query evaluation, because they are as powerful as HTJ operators.

1.1 Problem Statement

In the presence of the various operators mentioned in Section 1, it is not clear how they behave in real-world native XDBMS scenarios as provided by the XMark benchmark queries [19] or by the TPoX benchmark queries [17]. These queries are more complex than simple path expressions or twig patterns used for the evaluation of most SJ, HTJ, and index access operators. There is—to the best of our knowledge—no query evaluation framework that can employ SJ, HTJ, and various indexes in combination or exclusively for query evaluation. Furthermore, in present systems, cost-based query optimization using these operators is not possible yet, because there is little knowledge on the characteristics of SJ and HTJ operators in real-world scenarios as well as on the effectiveness of logical query rewrites like SJ reordering. Consequently, no widely accepted cost model for page-oriented XML storage is available, which serves for driving the query optimization process.

1.2 Contribution

In this work, we propose a framework, which serves at the beginning as a testbed, allowing to enumerate different query execution plans for a given XQuery expression using a rich physical algebra[1]. By executing all of them under realistic conditions provided by a complete native XDBMS infrastructure [10], we can test well-known concepts from relational query optimization w. r. t. their relevance for XML query optimization. Based on the best practices that will be distilled from an exhaustive experimental evaluation of the physical operators and different query-rewrite strategies, we will derive an appropriate cost model that describes the CPU and IO costs of each physical operator in a system-independent way. Finally, the optimization framework and the cost model are turned into a next-generation extensible, complete, and efficient cost-based native XML query optimizer.

1.3 Related Work

Selinger et al. [20] introduced the first cost-based query optimizer, which was part of *System R*—the prototype of the first relational database system. The optimizer was capable of optimizing simple and linear SPJ (join, project, and select) queries. The authors define a simple cost model based on weighted IO and CPU costs and use statistics on the number of data pages consumed by relations to bind the cost model's variables to concrete values. Their dynamic programming algorithm works in a bottom-up style where optimal operator fittings for access paths are selected first. Thereafter, an optimal join order is chosen based on a local optimality assumption. To early prune the search space, not all possible enumerations are considered. Instead, they only take so-called interesting join orders into account, i. e., orders that do not need additional introductions of Cartesian products. Graefe and DeWitt [7] introduced the *EXODUS Optimizer Generator*. This system is not tailored to a specific data model and supports specifying algebraic transformations as rules. These rules serve as input for an optimizer generator—together with a concrete data model—which creates a tailor-made query optimizer. The *Starburst project* [18] contributed important concepts to the emerging research field of query optimization. Among other things, they present the so-called Query Graph Model (QGM)—an extended relational algebra with a strong emphasis on structural relationships between language constructs. Beyond that, they introduce the concepts of rule-based query optimizers which can be easily modified by adding new rules for query transformation and query translation. Hence, this approach improves the extensibility of such systems very much. The *Volcano project* [8] as well as the *Cascades project* [6] are heirs of the EXODUS project. The authors distinguish between transformation rules, which serve for algebra-to-algebra transformations, and implementation rules describing the mapping from logical algebra expressions to operator trees. In contrast to the query optimization approach of System R [20], they use a top-down query optimization algorithm that takes a bird's eye view on QEPs.

Lanzelotte and Valduriez [13] contributed an extensible framework for query optimization that models the search space independent of a particular search strategy. Using this approach, developers can build highly-extensible plan enumeration frameworks. Kabra and DeWitt [12] proposed *OPT++* as an object-oriented approach for extensible query optimization. By combining an extensible search component with an extensible logical and physical algebra representation, they lift the work of Lanzelotte and Valduriez to the object-oriented world.

The classic work of McHugh and Widom [15] on the optimization of XML queries targets only at optimizing path expressions using navigational access paths and lacks support for SJ and HTJ operators. Wu et al. [23] proposed five novel dynamic programming algorithms for structural join reordering. Their approach is orthogonal to our work, i. e., it can be employed to choose the best join order in SJ-only scenarios. Compared to our work, they use only a very simple cost model for driving the join-reordering process and do not consider the combination of SJ and HTJ operators as well as different index access operators. Zhang et al. [24] introduced several statistical learning techniques for XML cost modeling. In contrast to our work, which will follow a static cost modeling approach, they demonstrate how to model the cost of a navigational access operator. Unfortunately, they do not cover SJ and HTJ operators. Balmin et al. [2] sketch the development of a hybrid cost-based optimizer for SQL and XQuery being part of *DB2 XML*. Compared to our approach, they evaluate every path expression using an HTJ operator and cannot decide on a fine-granular level whether to use SJ operators or not.

2. THE OPTIMIZATION FRAMEWORK

Our query optimization framework relies on—but is not restricted to—the *XML Transaction Coordinator* (XTC), which is a stable native XDBMS infrastructure [10] with full transaction support and various XML data-processing interfaces for SAX, DOM, and XQuery. XQuery expressions[2] are represented using the *XML Query Graph Model* (XQGM)—an extended version of the seminal Query Graph Model [18]—, which is our logical XML algebra. For this algebra, we already developed a heuristics-based query compiler [14]. Our framework is accompanied by a powerful visual explanation tool that allows to track the complete query optimization process from the beginning to the end. For example, Figure 1 as well as Figures 3(a)–3(c) were automatically created using this tool [14].

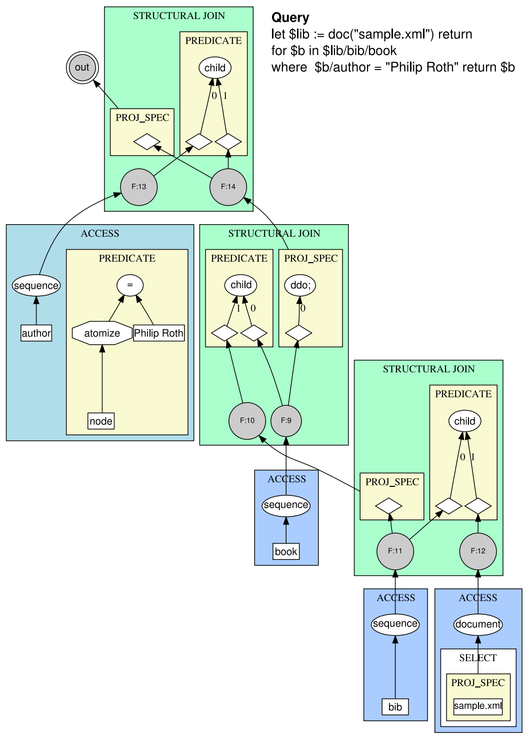
2.1 Motivating Example

Let us consider a simple XQuery statement throughout the rest of this work that retrieves all subtrees of books authored by *Philip Roth* from an XML document for managing a private book collection. Figure 1 shows the query and the corresponding logical algebra expression expressed

[1]It is worth mentioning that our framework has almost all capabilities of a full-fledged cost-based query optimizer. The only missing ingredient is a cost model to prune the search space.

[2]Please bear in mind that we can express all core features of the XQuery language such as FLWOR expressions, path expressions, comparison and positional predicates, quantifications, and node-construction expressions using our XML algebra.

as an XQGM instance. Here, the structural relationships are evaluated using logical SJ operators.



```
Query
let $lib := doc("sample.xml") return
for $b in $lib/bib/book
where  $b/author = "Philip Roth" return $b
```

Figure 1: Query Plan expressed as XQGM instance

2.2 Components of the Framework

Figure 2 shows the different components of the query optimization framework. The *Plan Generator* component provides the glue between the optimization framework and the XDBMS. It receives an XQGM instance (Query Graph) and returns one or many—depending on the plan enumeration strategy—Query Execution Plan(s) (QEPs) as result. Beyond that, it allows to influence the overall setup of the query optimizer by choosing the search strategy and the physical operators that shall be considered during plan generation.

2.2.1 State Space

Query optimization is a classical combinatorial optimization problem. Solutions to such problems can be described as states in a *state space* of semantically equivalent states. Finding such a solution is the main task of query optimization. A query optimizer starts at an initial state and manipulates it in such a way that the optimal or at least a near-optimal state is reached while the optimization goal (e. g., maximum throughput or minimal power consumption) is satisfied. At the beginning of query optimization, for every logical algebra operator in a query graph, a corresponding state graph is generated. It encompasses all *static properties* of a query plan, e. g., structural predicates, projection specifications, or orderings that have to be preserved and which are not changed during the query optimization process. Additionally, each state contains *dynamic properties*, e. g., required sorting on inputs, assigned cost-estimation values, and the currently assigned plan operator. In contrast to static properties, dynamic properties may change during every state transition.

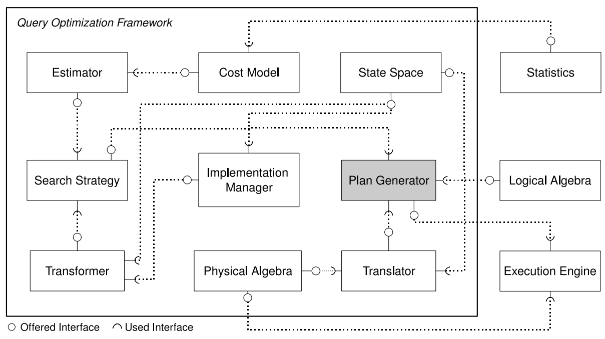
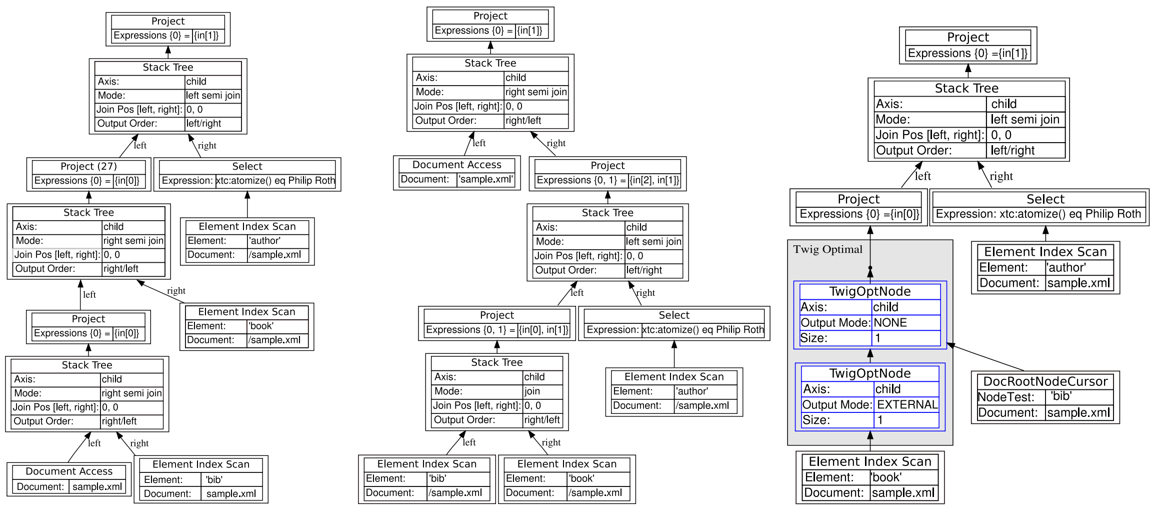


Figure 2: The Components of the Framework

2.2.2 Search-Space Exploration Strategies

Today, we do not know whether the local optimality assumption made in classical relational query optimization is still satisfied in the context of XML databases. Moreover, it is not clear which *search strategy* is the default choice for an XML query optimizer. Consequently, our framework supports different algorithms for exploring the search space, which we partition into bottom-up search strategies and top-down search strategies. *Bottom-up search strategies* perform an exhaustive exploration of the search space and are guaranteed to find the cheapest solution. They iteratively construct a search tree formed by all possible alternatives for a (sub)-tree and prune expensive plans[3]. For building this search tree, they start at the leaf nodes of a query plan and enumerate all possible access paths. For consecutive operators (e. g., a cascade of join operators), they create all possible combinations by applying query rewrites (e. g., join reordering). This process stops when the root of the query plan is reached. This approach is employed by the System R query optimizer [20]. Our dynamic programming algorithm follows this classical principle. In advantage over the System R optimizer, which was only capable of creating left-deep join trees, our algorithm also supports the generation of bushy plans and right-deep plans [5]. The *top-down search strategies* currently available in our framework can perform a probabilistic search. Accordingly, there is no guarantee that they find the best possible solution. Anyhow, they allow for the optimization of very large join trees, because the space complexity of bottom-up strategies is bound by $n!$, where n is the number of input operators involved. The prominent top-down search strategies currently supported by our framework are: Iterative Improvement, Simulated Annealing, and Two-Phase Optimization [11]. *Iterative Improvement* carries out down-hill moves in the search space and can get stuck in local cost minima. Contrariwise, the *Simulated Annealing* algorithm makes down-hill and up-hill moves and consequently increases the probability to find the optimal solution. Finally, *Two-Phase Optimization* is the combination of the aforementioned strategies. In the first phase, a plan with locally minimal cost is obtained using

[3]Even though we currently perform pruning based on the local optimality assumption, our framework is flexible enough to handle arbitrary pruning strategies.



(a) Left-deep QEP (b) QEP after SJ reordering (c) QEP with SJ and HTJ

Figure 3: Some possible query execution plans

Iterative Improvement. This plan furnishes as input for the second phase, where the plan is further optimized using Simulated Annealing.

2.2.3 *Enumeration of Alternative Plans*

The Transformer component performs state transitions using query rewrite to create for given query plans semantically equivalent alternatives. Every rewrite is specified as a *transformation rule* which contains a condition part and an action part. If the condition is satisfied, the action is applied to the query graph. Using this rule-based approach, the implementation of new rewrite rules is straightforward. The most important rewrite rules in the relational world are join commutativity and join associativity. In the XML world, both rules can be applied for rewriting structural joins and value-based joins. The *join commutativity rule* simply replaces the left with the right join partner and vice versa. The *join associativity rule* exchanges the order in which two adjacent join operators are evaluated. For example, Figure 3(b) illustrates an reordered version of the query graph shown in Figure 3(a), which both correspond to XQGM instance of Figure 1. For structural joins, we add an additional rewrite rule, which we call *join fusion* [22]. It permits to replace two adjacent structural join operators, which only evaluate the `child` or `descendant` axis by a single twig join operator that evaluates both joins holistically. For example, Figure 3(c) shows a QEP resulting from the application of this rewrite rule[4].

Table 1 shows the number of possible alternatives for the XMark benchmark queries [19] generated using the join-associativity rule and the join-fusion rule[5].

[4]In Figure 3(c), the DocRootNodeCursor operator corresponds to `doc("sample.xml")/bib`.

[5]Please note, the search space of a full-fledged XML query optimizer for these queries will increase tremendously when value-based join reordering, different access paths, and plan operators are taken into account, too.

Query	# of alternatives
Q1–Q5, Q16–Q18	6
Q6, Q14	3
Q7	1
Q8	36
Q9	432
Q10	143
Q11, Q12	72
Q13	12
Q15	3,071
Q19	5
Q20	10,367

Table 1: Alternative QEPs for XMark queries

2.2.4 *Management of Metadata*

The orthogonal dimension to query rewrite, where cost-based optimization is performed, is the optimal selection of access paths and the best plan-operator fittings. To accomplish this task, the Transformer component creates all valid combinations of input operators for every equivalent query plan. It uses the Implementation Manager component that encapsulates the metadata describing the characteristics of all physical algebra operators available in the system.

2.2.5 *Cost Model and Cost Estimation*

The cost model is a system-dependent set of formulae describing the costs of every physical algebra operator in the XDBMS. In combination with statistics provided by the system catalog as well as with the cardinality estimates provided by the system's cardinality estimation framework, the cost model allows to assign costs to every possible query plan. Because of the exponential growth of alternatives, not every semantically equivalent query plan created by the Transformer component can be taken into account for subsequent optimization steps. Hence, expensive query plans are eliminated early. The cost estimator assigns to each query plan (logical algebra expression and its corresponding state) a cost, which is estimated using the cost model. Only the most promising plan is memorized for future steps; the remaining $n - 1$ query plans are dismissed.

2.2.6 Query Translation

Finally, when the optimal query plan for a given query is determined using the cost model, this plan is mapped to a query execution plan. The translator component performs rule-based logical-to-physical operator mappings.

Each translation of a logical operator to one or more physical operators is described by so-called translation rules. A *translation rule* has a condition part (a structural pattern) and an action part. During query translation, a query plan is traversed in left-most depth-first order. If a rule matches, the affected sub-tree is translated according to the action part and memorized. To make non-deterministic behavior impossible, for two given translation rules r_1 and r_2, the condition part of r_1 and r_2 must not be in conflict, i. e., both rules must not match at the same time.

2.2.7 The Physical Algebra

Currently, our framework can dispose of approximately 50 physical algebra operators for query evaluation. All of them belong to one of the following operator classes:

Access operators, for which our framework provides two different types, are available in numerous variations. First, the Document Access operator makes the virtual document root of an XML document available that provides the initial evaluation context. Second, we can employ different kinds of operators that provide access to document nodes via navigational primitives or index accesses. The so-called Document Index operator serves as primary access method and provides navigational access to the document. We additionally provide a kind of inverted-list index which is called Element Index. This secondary access method provides efficient access to element nodes[6]. As tertiary access methods, we provide different kinds of path, content, and content-and-structure indexes.

Navigational operators evaluate structural predicates like the attribute axis using a nested-loops join algorithm.

Structural Join operators can be considered as a kind of stack-based merge-join operator that works on two sequences of element-node streams and provides all nodes fulfilling the structural predicate sorted by the left or right input stream. Whenever possible, these operators are semi-join operators. Currently, the only SJ operator supported by our framework is the StackTree operator [1].

Holistic Twig Join operators are stack-based n-way join operators. Since they are managing a global state, rather than Structural Join operators, they reduce the amount of intermediate results not being part of the final query result and outperform Structural Join operators in some situations. Some Holistic Twig Join operators can perform jumps on the input streams to minimize IO costs. Our framework provides an extended version of the TwigOptimal operator [3] which is, among other things, enriched with capabilities for evaluating positional predicates.

Besides structural join operators, we can use different kinds of *Value-Based Join operators*: nested-loops join, sort-merge join, and a hash-join operator for equality predicates. *Filter operators* allow for the evaluation of aggregate functions like count or for the evaluation of value-based predicates like equality of text values. Since XML relies on an ordered data model, *Sort operators* allow for sorting a tuple sequence if this property cannot be preserved by the physical operator that produced it. For example, during structural-join reordering, additional sorting can become necessary. *Group-By operators* are used for the evaluation of positional predicates. *Set operators* serve for the evaluation of set operations (union, intersection, and difference) on ordered tuple sequences. *Unnest operators* allow for unnesting tuple sequences for example after the evaluation of positional predicates.

In more complex XQuery expressions, variables bound by let expressions are reused several times. Therefore, we employ a so-called *Split operator* that allows to calculate the path expression only once, instead of multiple times, and provides input for the referring sub-trees. The so-called *Merge operator* serves as a counterpart to the Split operator and finally merges the different tuple sequences according to a predicate check.

[6]For example, the QEP in Figure 3(a) uses such operator instances as access paths.

3. FUTURE WORK

In Section 2, you could convince yourself concerning the capabilities of our query optimization framework. Backed by our powerful plan generation infrastructure, we will tackle research topics in various areas:

- *How can we efficiently use SJ and HTJ operators in complex XQuery evaluation scenarios?*
  First experiments revealed that joint query processing with SJ and HTJ operators can speed-up query evaluation tremendously and can be even more effective than SJ reordering. Therefore, our cost model must allow to decide whether the application of join fusion reduces the costs or not.

- *Which impact do value-based join operators have on the overall query execution performance?*
  Reconsidering value-based join operators in the context of XML is necessary, because they explicitly require access to the document for every predicate check. Consequently, we assume that value-based join reordering can reduce query evaluation costs by several orders of magnitude.

- *Which is the optimal set of access paths?*
  Finding the optimal access paths is even more complex in the XML context because of their confusing variety (see Section 1). Therefore, we have to develop cost formulae for PAPs, SAPs, and TAPs serving as heuristics to solve this NP-complete optimization problem.

- *How can we make the most out of CAS indexes?*
  CAS indexes are a powerful means for indexing content and structure at a time. Today, it is not known whether their contribution to the reduction of query evaluation costs justifies their existence in the presence of probably high maintenance costs or not. Furthermore, we do not know under which conditions they can replace HTJ operators.

- *Which query optimization strategies are required when paying attention to new hardware developments?*
  Green computing is gaining more and more attention in industry and research. This development must also

be considered in the context of query optimization. For example, the reduction of energy consumption as a new optimization goal will require new cost models reflecting the access patterns of new hardware such as flash disks.

- *How can we speed-up node construction?*
  For providing the final query result, the DeweyIDs assigned to tuples during query processing must be dereferenced to get the actual XML nodes. Accelerating this task can further reduce the query processing time by orders of magnitude.

4. SUMMARY

In this work, we introduced a query optimization framework that relies on an efficient native XDBMS infrastructure and support plan generation for a large fragment of the XQuery language. The framework is very flexible and supports different strategies for search-space exploration. Because we follow a strictly rule-based approach, adding new transformation rules (logical query rewrites) and translation rules (e. g., for considering new plan operators as alternatives) is simple. The system proposed in this work will serve as our testbed for the development and evaluation of the system-independent cost model encompassing approximately 50 physical operators. Amongst others, this cost model will make the most out of structural joins, holistic twig joins, value-based joins, and various competing index access operators. In the end, the framework can be turned into the first full-fledged cost-based native XML query optimizer whose query evaluation performance can be compared to relational-based XQuery evaluation engines.

Acknowledgment

Financial support by the Research Center (CM)^2 (http://cmcm.uni-kl.de) of the University of Kaiserslautern is appreciated.

5. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE Conference*, pages 141–154, 2002.

[2] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. E. Simmen, M. Wang, and C. Zhang. Cost-Based Optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD Conference*, pages 310–321, 2002.

[4] R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB Conference*, pages 436–445, 1997.

[5] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[6] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[7] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proc. SIGMOD Conference*, pages 160–172, 1987.

[8] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. ICDE Conference*, pages 209–218, 1993.

[9] T. Härder, M. P. Haustein, C. Mathis, and M. Wagner. Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data & Knowledge Engineering*, 60(1):126–149, 2007.

[10] M. Haustein and T. Härder. An Efficient Infrastructure for Native Transactional XML Processing. *Data & Knowledge Engineering*, 61(3):500–523, 2007.

[11] Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. SIGMOD Conference*, pages 312–321, 1990.

[12] N. Kabra and D. J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal*, 8(1):55–78, 1999.

[13] R. S. G. Lanzelotte and P. Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proc. VLDB Conference*, pages 363–373, 1991.

[14] C. Mathis, A. M. Weiner, T. Härder, and C. R. F. Hoppen. XTCcmp: XQuery Compilation on XTC. In *Proc. VLDB Conference*, pages 1400–1403, 2008.

[15] J. McHugh and J. Widom. Query Optimization for XML. In *Proc. VLDB Conference*, pages 315–326, 1999.

[16] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. ICDT Conference*, pages 277–295, 1999.

[17] M. Nicola, I. Kogan, and B. Schiefer. An XML Transaction Processing Benchmark. In *Proc. SIGMOD Conference*, pages 937–948, 2007.

[18] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD Conference*, pages 39–48, 1992.

[19] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conference*, pages 974–985, 2002.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD Conference*, 1979.

[21] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proc. SIGMOD Conference*, pages 110–121, 2003.

[22] A. M. Weiner, C. Mathis, and T. Härder. Rules for Query Rewrite in Native XML Databases. In *Proc. EDBT DataX Workshop*, pages 21–26, 2008.

[23] Y. Wu, J. Patel, and H. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proc. ICDE Conference*, pages 443–454, 2003.

[24] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proc. VLDB Conference*, pages 289–300, 2005.