Visualizing Cost-Based XQuery Optimization

Andreas M. Weiner [1], Theo Härder [2], and Renato Oliveira da Silva [3]

Department of Computer Science, Databases and Information Systems Group,
University of Kaiserslautern, P. O. Box 3049, 67653 Kaiserslautern, Germany
[1]weiner@cs.uni-kl.de [2]haerder@cs.uni-kl.de [3]oliveira@cs.uni-kl.de

***Abstract*— Developing a full-fledged cost-based XQuery optimizer is a fairly complex task. Nowadays, there is little knowledge concerning suitable cost formulae and optimization strategies for exploring and constraining the tremendously large search space. To allow for a fair assessment of different optimization strategies, physical algebra operators, and indexing approaches, we developed an extensible optimization framework. The framework is accompanied by a supportive visual explain tool that enables user interactions to refine the inspection and the comprehension of the query plans proposed. Using this tool, the optimizer can be dynamically reconfigured and the impact of different optimization strategies on the final query execution plan can be immediately visualized.**

I. Introduction

In recent years, XML became the de-facto standard for exchanging structured and semi-structured data in business and in research. XML documents can be efficiently stored and processed using native XML database management systems (XDBMSs). Today, XQuery [3] is the predominant query language for XML. Even though cost-based query optimization contributed much to the success of relational database systems, this approach has only gained marginal attention in the context of XML. Among other things, this is due to a missing cost model for constraining the search space.

To understand how and whether established techniques of relational cost-based query optimization (e. g., reordering of join operators) can be reused and what new techniques have to be developed to make a significant contribution for accelerating XQuery execution in native XDBMSs, we follow a system-engineering approach. By using our cost-based XQuery optimization framework [11], [13], we can employ a testbed for exploring and comparing physical operators, optimization techniques, and indexing approaches under uniform and fair conditions. The framework is based on the XML Transaction Coordinator (XTC)—an efficient and transactional native XDBMS [5].

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. The optimization framework is accompanied by a visual explain tool that allows for an immediate assessment of the query optimizer.

II. Contribution

The specific contribution of this demonstration can be briefly described as follows:

- We introduce a flexible, rule-based, and cost-based optimization framework that allows developers to implement and evaluate cost-based XQuery optimization techniques in a native XML database management system.
- To allow developers to interact with the framework, we describe a visual explanation tool that allows for reconfiguring the optimization framework (e. g., a different search strategy) even during runtime.
- Because our framework tracks the complete XQuery optimization process, every modification of the logical algebra expression (query graph) or updates in any query execution plans (QEP) can be visualized.
- Besides visualizing query graphs and QEPs, several aspects of query execution are exemplified. For instance, after executing a QEP, a data-flow analysis is performed and each edge in the QEP is annotated with the number of actually processed tuples as well as with the estimated number of tuples according to the system's cost estimation component [1].

III. Cost-Based XQuery Optimization

Figure 1 shows the different stages an XQuery statement traverses during cost-based query optimization. Initially, the XQuery expression is parsed and mapped to an *Abstract Syntax Tree* (AST). Next, normalization maps the AST to a canonical representation according to the formal XQuery semantics and, therefore, removes all syntactic sugar. Afterwards, static type checking allows for type inference. The simplification step removes redundant parts of the query and prepares it for a translation to the *XML Query Graph Model* (XQGM)—an extended version of the seminal Query Graph Model (QGM) [8]. This representation serves as our logical XQuery algebra, which relies on nested tuples. Until the day of writing, our algebra supports a significant subset of the XQuery language, e. g., FLWOR expressions, path expressions, comparison and positional predicates, quantifications, and node-construction expressions. In a previous work, we introduced XTCcmp [7] that compiles XQuery expressions into this internal representation and maps them directly onto physical algebra operators, which form the QEP. Previously, these mappings were not performed based on cost-based decisions. In this contribution, we leave the XQuery-to-XQGM compilation unchanged and replace the static XQGM-to-QEP mappings by dynamic and cost-based transformations. For this purpose, we use our rule-based query optimization framework [11], [13].

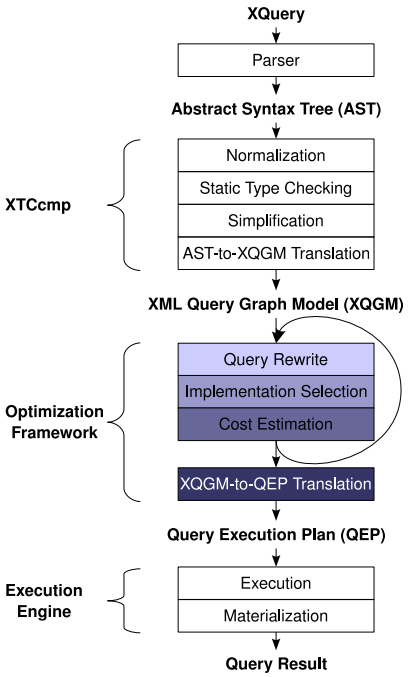


Fig. 1. The query optimization process

IV. System Overview

Figure 2 shows the main components of the framework, where the *Plan Generator* serves as glue between XTC and XTCcmp. At the beginning of query optimization, for every XQGM expression provided by XTCcmp, a corresponding state graph is generated. Such a graph serves as blueprint for logical-to-physical algebra mappings. It contains all *static properties* of a query plan, e. g., structural predicates, projection specifications, or orderings that must be valid for every possible alternative implementation and that remains unchanged during the complete query optimization process. In addition, each state contains *dynamic properties*, e. g., required sorting on inputs, cost estimation values, and the currently assigned plan operator(s). In contrast to static properties, dynamic properties may change during every state transition.

As in classical relational query optimization, our framework supports two classes of strategies for exploring the search space: bottom-up and top-down strategies. *Bottom-up strategies* exhaustively explore the search space and always find the cheapest solution, if the local optimality assumption holds and a universal distribution of values is obtained. In contrast, *top-down strategies* perform a probabilistic search and may miss the best possible solution. Anyhow, they effectively support the optimization of very large join trees [6].

The *Transformer* component performs state transitions using query rewrite to create semantically equivalent alternatives for given query plans. Every rewrite is specified as a *transformation rule* that 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 integration 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, Fig. 3(c) illustrates a reordered version of the query graph shown in Fig. 3(b), and both correspond to the XQGM instance of Fig. 3(a). For structural joins, we add an additional rewrite rule (*join fusion*) that permits to replace two or more adjacent structural join operators by a single twig join operator [12], [14]. Our cost model is a

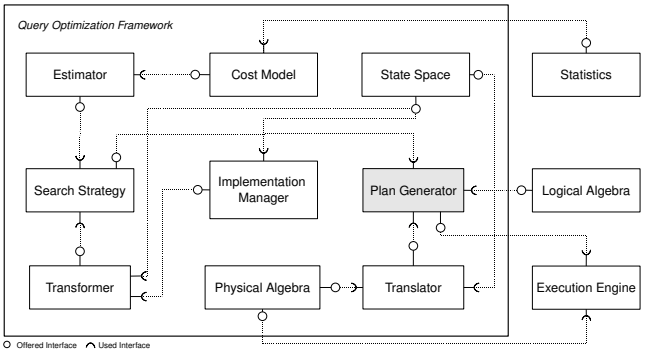
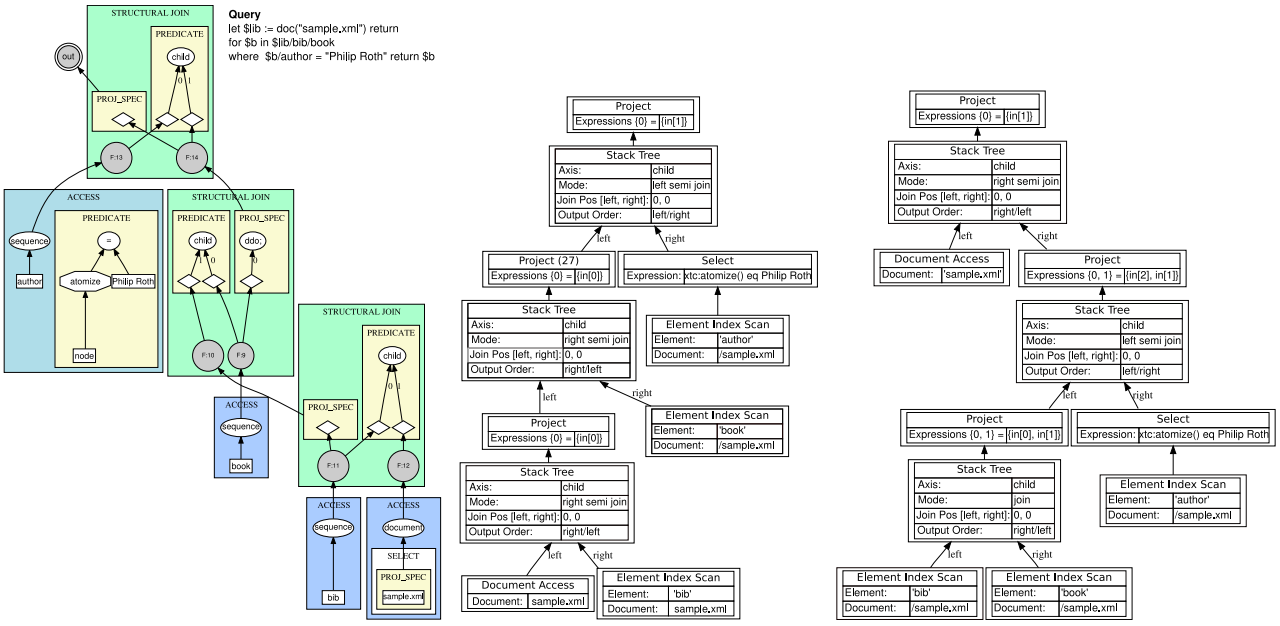


Fig. 2. The query optimization framework

system-dependent set of formulae describing the costs of every physical algebra operator in our system. By means of statistics from the system catalog and cardinality estimates provided by a specific component [1], the cost model allows to assign costs to every possible QEP. Because of the exponential growth of alternatives, not every semantically equivalent query plan delivered by the Transformer component can be considered 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 that is estimated using the cost model. Only the most promising plan is kept for future steps; the remaining $n - 1$ query plans are not considered.

Using the *Translator*, the remaining XQGM instance is mapped to a QEP. Each translation of a logical operator to one or more physical operators is guided by translation rules. A *translation rule* consists of a structural pattern as the condition and an action part. During query translation, a query plan is traversed in left-most depth-first order. If a rule matches, the affected subtree 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.

Which set of physical operators is actually chosen to evaluate a subtree of the query, depends on the cost model. At the present date, our framework has a repertoire of approximately 50 physical algebra operators, which can be used for query evaluation. For example, we can use structural joins, holistic



(a) XQGM instance and corresponding query (b) Left-deep QEP (c) Reordered QEP

Fig. 3. A logical algebra expression and possible query execution plans

twig joins, and various indexing operators[1].

V. Demonstration Scenario

For the demonstration, we come up with a predefined set of all XMark benchmark queries [9] and XMark documents of varying sizes to run these queries on.

A. Configuring a Query Optimizer

To use a concrete instance of the framework, we provide a visual explain tool that allows for the configuration and the assessment of the XQuery optimizer[2]. Basically, the search space is formed by two orthogonal dimensions: semantically equivalent query graphs gained by query rewrite and alternative physical operator fittings. Therefore, each available transformation rule (e. g., join reordering) can be switched on and off to increase or decrease the size of the search space. Furthermore, the user can select all physical alternatives he wants to be considered during cost-based optimization. For example, inputs for a Structural Join Operator [2] can be provided by i) the document index [5], ii) the element index [5], or iii) by exploiting simple path indexes (like `//a`).

Today, we are not sure which search strategy serves best in the context of XQuery. Therefore, the query optimizer can be dynamically re-configured with different types of bottom-up strategies and top-down strategies. Currently, we provide two kinds of bottom-up strategies: Full Enumeration and Dynamic Programming. For queries with a small search space, *Full Enumeration* allows for an inspection of all possible QEPs. If the search space is large—which is true for almost all real-world queries—*Dynamic Programming* with cost-based pruning helps to find an optimal solution [10]. Moreover, our framework supports three top-down strategies: Iterative Improvement, Simulated Annealing, and Two-Phase Optimization [6]. *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 Iterative Improvement. This first-phase result, in turn, serves as input for the second phase, where the plan is further optimized using Simulated Annealing.

[1]More details on the different types of physical operators supported by our framework can be found in [11], [13].

[2]We provide a set of default configurations that can be customized by the user.

B. Using the Query Optimizer

When the configuration is finished, the resulting framework instance can be used for query optimization. To show the impact of the selected configuration on the quality of resulting QEPs, all logical rewrites performed by XTCcmp as well as the final cost-optimized QEP can be visualized. For example, the XQGM instance shown in Fig. 3(a) and the QEPs depicted in Fig. 3(b) and Fig. 3(c) were generated using our tool.

After executing a QEP, a data flow analysis can be performed. For example, all edges in the QEP are annotated with the number of actually processed tuples and join operators provide information about their estimated and real selectivities. If Full Enumeration is chosen as the search strategy, all possible QEPs are shown. For example, Figure 4 illustrates a screenshot of our visual explain tool. Here, only one of 72 possible QEPs for XMark query Q11 is shown.

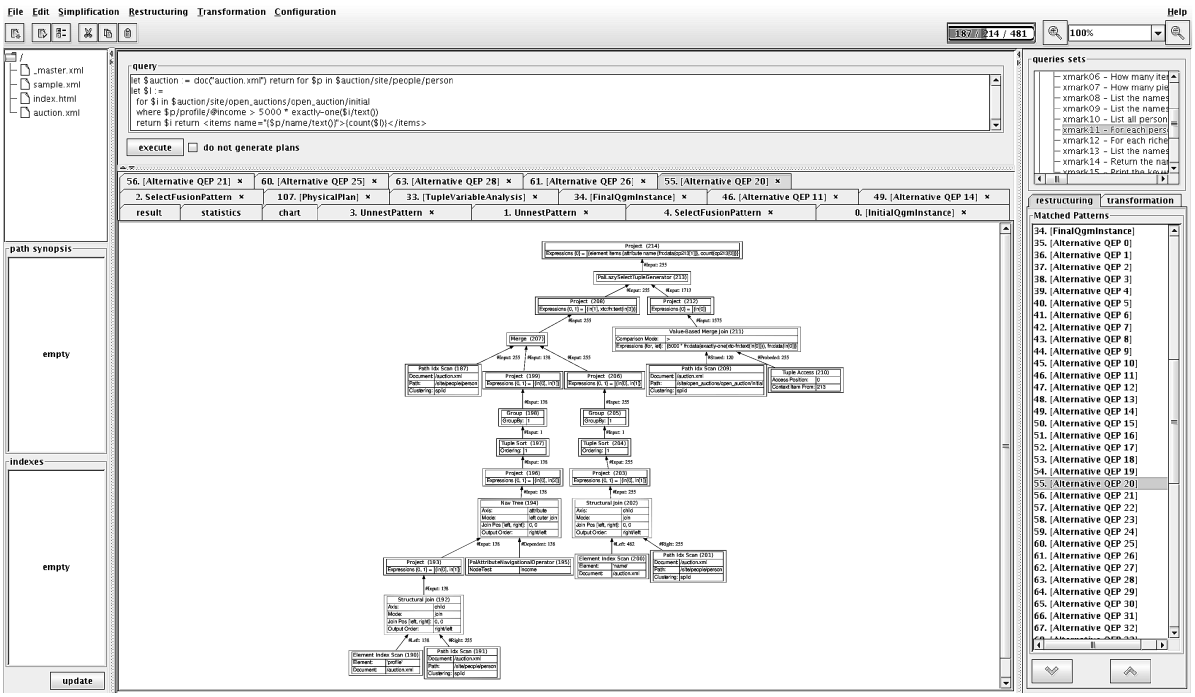


Fig. 4. Visually explaining cost-based XQuery optimization

VI. Implementation

Our framework is completely implemented using Java Version 1.6.0_06. Our graphical user interface (GUI) relies on the Java Swing API. A connection between the explain tool and the XTC server is established using Java RMI. The server receives a query request from the client and forwards it to the optimization engine that processes the different stages shown in Fig. 1. For every logical XQGM rewrite and every QEP, a textual representation of the resulting graph—a so-called *dot graph*—is created. When the query execution has finished, the query result, different statistics for the data-flow analysis, and all dot graphs are passed back to the explain tool. At the client side, we employ the *GraphViz* visualization framework [4] for layouting the graphs. Here, all dot plans are converted into *Scalable Vector Graphic* (SVG) instances and are rendered in our GUI using the *Apache Batik SVG Toolkit*[3].

VII. Summary

In this contribution, we presented a cost-based XQuery optimization framework that can be configured and assessed using a supportive visual explain tool. Due to our own experience, it helps to rapidly develop and evaluate cost-based optimization techniques and supports novices in getting a basic understanding of XQuery compilation, optimization, and execution.

References

[1] J. Aguiar Moraes Filho and T. Härder, "EXsum—An XML Summarization Framework," in *Proc. IDEAS Conference*, 2008, pp. 139–148.

[2] 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*, 2002, pp. 141–154.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon, "XQuery 1.0: An XML Query Language—W3C Recommendation 23 January 2007," http://www.w3.org/TR/2007/REC-xquery-20070123/, 2007.

[4] J. Ellson, E. Gansner, E. Koutsofios, and S. N. G. Woodhull, "Graphviz and Dynagraph—Static and Dynamic Graph Drawing Tools," in *Graph Drawing Software*, (Eds.) M. Junger and P. Mutzel, pp. 127–148, Springer, 2003.

[5] M. Haustein and T. Härder, "An Efficient Infrastructure for Native Transactional XML Processing," *Data & Knowledge Engineering*, vol. 61, no. 3, (2007), pp. 500–523.

[6] Y. E. Ioannidis and Y. C. Kang, "Randomized Algorithms for Optimizing Large Join Queries," in *Proc. SIGMOD Conference*, 1990, pp. 312–321.

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

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

[9] 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*, 2002, pp. 974–985.

[10] 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, pp. 23–34.

[11] A. M. Weiner, "Framework-Based Development and Evaluation of Cost-Based Native XML Query Optimization Techniques," in *Proc. VLDB PhD Workshop*, 2009.

[12] A. M. Weiner and T. Härder, "Using Structural Joins and Holistic Twig Joins for Native XML Query Optimization," in *Proc. ADBIS Conference*, LNCS 5739, Springer, 2009, pp. 149–163.

[13] A. M. Weiner and T. Härder, *Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization, and Data Query Technologies*, chap. A Framework for Cost-Based Query Optimization in Native XML Database Management Systems, IGI Global, 2010.

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

[3] `http://xmlgraphics.apache.org/batik`