XTCcmp: XQuery Compilation on XTC

Christian Mathis, Andreas M. Weiner, Theo Härder, and Caesar Ralf Franz Hoppen
Databases and Information Systems Group, Department of Computer Science
University of Kaiserslautern, P. O. Box 3049, D-67653 Kaiserslautern, Germany
{mathis | weiner | haerder | hoppen}@informatik.uni-kl.de

ABSTRACT

XTCcmp, the XQuery Compiler of a native XML database system, extends Starburst's well-known Query Graph Model to serve as an internal representation and basis for query restructuring of XQuery expressions. Furthermore, XTCcmp is able to generate execution plans supporting a wide range of both well-known and newly developed variants of core XML processing algorithms and indexes. Our demo visualizes all rule-based transformation stages, i. e., simplification, algebraic rewriting, and plan generation. Furthermore, via an interface to the extensible rule configuration, it allows interaction with the query compiler to vary configuration parameters and to control the compilation outcome.

1. MOTIVATION

Query processing—especially over XML data—is a complex task. In computer science, the answer to complexity is abstraction. Therefore, it is quite natural to split up query processing into system-independent problems (logical abstraction level) and problems that are system-dependent (physical abstraction level).

Considering XML query processing at the physical level, a plethora of algorithms has been proposed over the recent years, such as navigational primitives, structural joins, twig joins, and various forms of path, content, and hybrid indexes. Often these approaches content themselves with solving subproblems (e. g., twig pattern matching, path indexing) that are important to XML query processing, but they do not reason about the integration of their proposals into a full-fledged query processor.

At the logical level, there is still no commonly agreed logical XML algebra. However, we think the classification in [2] still holds. It basically identifies two primary movements: coarser-grained macro-level algebras that operate on complete sub-trees [10], and finer-grained micro-level algebras that operate on (tuples of) nodes [12].

Starting from this initial situation, the design and implementation of the XTCcmp query compiler for the XML Transaction Coordinator (XTC)—a native XDBMS—was driven by the vision to bring concepts from both communities, i. e., from physical and logical parties, more closely together. Therefore, at the physical level, we implemented a large variety of both well-known and newly developed variants of physical operators and index structures, like navigational axis evaluation primitives, structural joins, index-based and scan-based holistic twig joins, or content-and-structure (CAS) indexes, and seamlessly integrated them into a physical XML algebra. At the logical level, we extended Starburst's Query Graph Model (QGM) to serve as an internal representation for XQuery expressions. The resulting XML Query Graph Model (XQGM) is underpinned by a micro-level algebra [7] (however, as sketched in the next Section, the algebraic query rewriter can also detect and exploit tree-based macro-level algebra operators). To provide for extensibility, we follow the classical rule-based approach, in which all query transformations are defined by extensible sets of rules.

Our system demonstration traces the behavior of the rule-based XML query compiler. In particular,

- it shows how Starburst's Query Graph Model can be extended to support XQuery;
- it visualizes three important stages of query compilation, i. e., *Simplification*, *Algebraic Rewriting*, and *Plan Generation*;
- it presents how a large variety of XML evaluation algorithms cooperate in a physical algebra;
- it allows online modification of the rule configuration, thereby interacting with the query compiler to control the compilation outcome; and
- it serves as a visual explain tool for database users and developers facilitating tuning, debugging, and the development of new query transformations.

In the following, we provide an overview over the query compilation process, the XQGM, and the query transformation rules implemented in XTCcmp.

2. SYSTEM OVERVIEW

Figure 1a depicts the overall query compilation and execution process in XTC. Initially, an XQuery expression is sent to the parser which generates an Abstract Syntax Tree (AST) representation, on which the next four translation steps elaborate: Following the XQuery Formal Semantics [4], the query is first transformed to a variant[1] of the XQuery Core language (thereby removing syntactic sugar) on which

[1]We normalize as far as possible to facilitate the XQGM mapping.

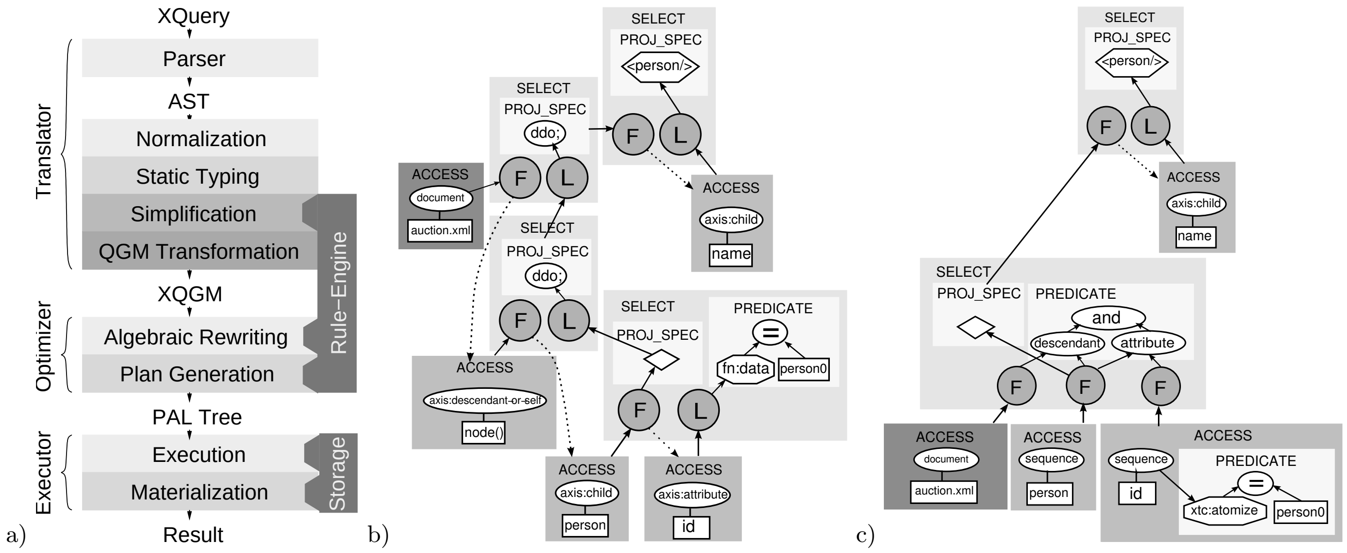

Figure 1: a) Query Evaluation in XTC, b) Canonical XQGM Instance, c) Rewritten XQGM Instance

then the static type is inferred. Static typing allows to detect errors in the query in an early stage and provides type information for the following simplification phase. Similar to [12], simplification removes unnecessary sub-expressions from the query, e. g., unnecessary calls to the `fn:data` function or unnecessary typeswitch expressions. In the last translation step, the query is mapped onto an XQGM instance.

The query translation phase is closely related to the suggestions from the Formal Semantics. The rationale is to keep a certain "proximity" to the specification to ensure correctness and to translate semantically equivalent queries to a common basis. Therefore, an initial (canonical) XQGM, as depicted in Figure 1b, can be seen as a graphical representation for XQuery expressions and can serve as an explanation model, e. g., for teaching.

As in classical query processing, the goal of the algebraic rewriting stage is to find an equivalent XQGM graph that hopefully leads to a more efficient evaluation. Therefore, we implemented primitives for query unnesting and selection push down. To address XML specifics, we furthermore developed means to identify tree-based (macro-level [2]) algebra operators.

Our intentions for plan generation were (1) to ensure that *every* XQGM instance can be mapped onto one or more plans; and (2) to address a large variety of evaluation algorithms in the physical algebra. These requirements are necessary, because cost-based XML query optimization is still an open issue and we plan to use our system as a testbed for comparison and measurements[2]. The optimization result is a Physical Algebra (PAL) operator tree that is based on the Open-Next-Close protocol to support pipelining.

In the last stage, the PAL tree is executed returning an intermediate result containing TID-style references to subtrees in the document. These references are resolved in the materialization step, before the result is returned to the client.

Of course, we cannot give a detailed discussion on all transformation stages. However, at least we want to provide a short introduction to XQGM and rewriting.

[2]This implies that the current optimizer uses heuristics only.

2.1 The XQGM in a Nutshell

Figure 1b depicts the canonical XQGM instance[3] for query:

```
let $auction := doc("auction.xml") return
for $b in $auction//person[@id = "person0"]
return <person>{$b/name}</person>
```

For readers familiar with Starburst's QGM and XQuery, we think XQGM is more intuitive and easier to read than the algebraic notations in [10, 12]: Basically, the graph is an operator tree, in which each operator consumes and produces sequences (of nested tuples [7]) and the data flows from bottom to top. The two basic operators are ACCESS (to access the document) and SELECT (to combine input sequences), which can carry a *predicate* (over input sequences), a *sorting specification* (for *order-by* queries), and an *output specification* (to compute output sequences). Tuple variables ("F" standing for `for` and "L" standing for `let`) control how input sequences are processed, where the semantics is borrowed from XQuery: "F" iterates over each item in the input sequence, whereas "L" passes the complete sequence to subsequent operations.

To read the XQGM instance, you can start at the document ACCESS operator (the darkest shaded operator that delivers the artificial document root R) and simply follow the arcs: R is bound to an "F"-variable. Then, the subtree of the adjacent "L"-variable has to be evaluated. Similar to nested SQL Selects, this subtree is a nested expression, requiring R as correlated input. With R as input (dotted line), the next ACCESS operator evaluates `descendant-or-self::node()` on a sequence S, over which the following "F"-variable iterates, passing on each node in S as correlated input to the subexpressions below. At the bottom, *person* and *@id* are evaluated and filtered by a SELECT operator that only returns those *persons*, where the *id* attribute is equal to "person0". On their "way up", intermediate results are collected and arranged in `distinct-doc-order` (`ddo`), before, in the final step, the

[3]This graph was generated by our demo system (and was only slightly hand-optimized for space efficiency).

name of each returned *person* is wrapped inside a new `person` element.

This evaluation model follows the XQuery Formal Semantics. However, evaluation on navigational primitives and nested subexpressions is often far from being efficient. Therefore, the query is rewritten to support bulk (sequence-based) evaluation algorithms. Figure 1c depicts a possible rewriting, in which most navigational primitives are unnested to structural joins, and in which the predicate expression is pushed down to the attribute access (directly atomizing the attribute). Note, the bottom-most access operators do not require a correlated input anymore, because they deliver complete sequences (e. g., all *persons*) from the document.

Based on this representation, plan generation would now start to find physical evaluation alternatives. For example, the above query could be evaluated starting with a content-and-structure index on `person/@id="person0"`, followed by a navigation from each delivered *person* to retrieve its *name* children.

2.2 Rule-Based Query Transformations

As indicated in Figure 1a, simplification, algebraic rewriting, and plan generation are specified using rules and are executed by a rule inference engine. These mechanisms are generic, i. e., the engine can be reused in each transformation stage. Rules consist of a pattern and an action part. The pattern is matched against an overlay tree (generated on top the AST or XQGM instance). Whenever a pattern match is found, the rule engine executes the action part on the match, thereby transforming the underlying tree. Table 1 provides an overview over the most important rules and their effects (note, due to the lack of space, precise conditions could not be presented).

Table 1: Implemented Query Transformation Rules

	Rule	**Description**
Simplification	TypebasedRem	Removes typeswitch, data(), or distinct-doc-order() expression when type known
	LetSubs	Substitutes "let" when variable singleton reference to variable
	OrderBySimpl	Removes order specs with no effect
	QuantSimpl	Removes unnecessary bindings in quantfified expressions
	ForAtRem	Removes unnecessary "at" particles from "for" bindings
Rewriting	DescOrSelfSubs	Substitutes "`descendant-or-self::node()/child::`" with "`descendant::`"
	PredPushDown	Pushes predicates to (access) operators
	SelectFusion	Merges select with its input if possible
	Unnest	Unnests to enable bulk operators
	StructJoinFusion	Creates multi-way joins for twig patterns
PlanGen	Access*Trans	Maps to navigation and scan for access
	SelectTrans	Generic mapping for all select operators
	StructJoinTrans	Maps to structural join algorithms
	TwigJoinTrans	Maps to holistic twig join algorithms
	IndexTrans	Maps to path index access

During rewriting, the two most important rules are *Unnest* and *StructJoinFusion*, because they transform the initial XQGM instance which contains many nested subexpressions and navigations, into a form facilitating the mapping onto bulk (set-based) evaluation operators, such as the holistic twig join or a path index access (as depicted in Fig. 1b and 1c).

2.3 Physical Algebra

As sketched in Table 1, our plan generator can consider a large variety of different evaluation strategies at the physical level. In particular, the generator supports 1. *Navigational Primitives*, which can either be evaluated directly on the document or on a special navigation support index; 2. *Structural Joins* with stack-based [3], hash-based [8], and navigational implementations [9]; 3. *Holistic Twig Joins* implemented as TwigOpt [6] and TwigList [11]; and 4. *Indexes* in the form of pure content indexes, pure path indexes, and hybrid content-and-structure indexes.

3. DEMONSTRATION OUTLINE

Our demonstration prototype consists of a graphical user interface (GUI) that connects as client to the XTC database server. The GUI client serves as a visual explain tool for database administrators and allows to influence the behavior of the XTCcmp. Additionally, it permits to follow the complete query evaluation process. The database administrator interacts with the GUI client by choosing the document to be queried, by selecting the rules that have to be applied, and by entering an XQuery expression. Next, these information are sent to the XTC query processor, which performs the query evaluation. During query optimization, only those rules selected before are used. Finally, the query result and additional information on query evaluation are sent back to the client and are presented to the database administrator.

To summarize, our GUI client can

- visualize XQGM instances and the final query execution plan;
- track the query compilation process, i. e., how query graphs evolve with the translation and optimization progress according to different rewrite strategies; and
- present in-detail statistical information on the query compilation process.

3.1 Making Query Compilation Visible

The GUI client is able to visualize an XQuery expression and its corresponding logical algebra expression—expressed as XQGM instance—as well as the final query execution plan, which solely consists of physical algebra operators. By allowing the selection of different evaluation strategies, we can see immediately how they influence the whole query compilation process in terms of applicable optimization rules as well as supported physical algebra operators.

To clarify XTCcmp's internal actions, the GUI client allows to track the query compilation process. Thereby, it visualizes every modification of the query graph and gives an impression of the graph's evolution over the time. Every modification is highlighted by color for easy recognition of changes even in large query graphs.

3.2 Impact of Rewrite Rules

As mentioned in Section 2.2, our query compiler follows a rule-based approach providing three different sets of rules for simplification, rewrite, and transformation. We can choose between two strategies for rule application: (1) arbitrary selection of rules and (2) configuration-based selection. Using the first strategy, we can turn on and off every rule. The second option allows to choose from predefined configurations encompassing useful evaluation scenarios.

Simplification and rewrite rules can be applied in two different modes: (1) interleaved and (2) rule-at-a-time. The first alternative applies different rules in an interwoven manner, i. e., always the rule is applied that complies best with the chosen evaluation strategy. On the other hand, switching to the rule-at-a-time mode allows to visualize the impact of a single rule on the whole XQGM instance by applying it as long as its left-hand side matches a (sub)tree of it. Both alternatives allow to assign a priority to each rule. Whenever the left-hand side of two or more rules match the same tree pattern, always the one with the higher priority is applied.

During plan generation, for every logical operator only one physical counterpart can be chosen at a time. Nevertheless, we provide for every logical operator different physical operators to choose from. Using these options, we can play around with different combinations of physical operators, e. g., employment of structural joins vs. holistic twig joins. For accessing the document to be queried, we allow to choose between the employment of different available indexes or navigational access methods.

3.3 Display of Statistical Information

The demonstration prototype provides in-detail statistical information on the query compilation process. Using these statistics, we can get an impression of the time spent for parsing, normalization, and type checking of the XQuery expression. Presenting the timings for query simplification and rewrite permits to figure out how much time was consumed by heuristics-based query optimization. Finally, the overall query evaluation time is shown which allows to compare different rule configurations w. r. t. their influence on query evaluation performance. In addition to the absolute numbers presented as milliseconds spent for certain tasks of the query compilation process, we additionally provide its share of the overall query evaluation time in percentage, so we can easily recognize which stage of the query evaluation process consumed the lion's share of time.

4. IMPLEMENTATION

Our demonstration prototype is implemented using the *Java* programming language (version 1.5) and uses its Swing API for the GUI. The prototype connects to the XTC server using Java RMI. For every modification of a query graph during query optimization, the query processor generates a corresponding textual representation using the *dot* language [5] and sends it to the client—together with the query result and statistical information. On the client side, the textual representations of the query graphs are transformed into SVG instances using the *GraphViz*[4] graph visualization software [5], which serves for automatic layouting of huge graphs. Finally, the SVG descriptions are rendered using the *Batik SVG Toolkit* [1].

5. CONCLUSIONS

In this demonstration, we show how flexible rule-based XQuery compilation is done in the XTC system. Our system covers the overall query evaluation process beginning at the translation of XQuery statements into the XQGM and ending at the delivery of the final query result.

Besides the main purpose of the visualization prototype, providing an interface to XTC's query optimizer, it also serves as a means for illustration of XQuery compilation. By activating the appropriate rules, we can get a feeling of how different evaluation strategies influence query evaluation performance. Finally, choosing between different transformation rules allows to play the role of a query optimizer by defining which physical operators should be used for query evaluation. Future work will focus on defining appropriate cost models and the integration of a statistics component, which enable the query optimizer to dynamically choose promising rules and physical operators automatically.

6. REFERENCES

[1] The Apache XML Graphics Project—Batik SVG Toolkit. `http://xmlgraphics.apache.org/batik/`, 2008.

[2] S. Al-Khalifa and H. V. Jagadish. Multi-Level Operator Combination in XML Query Processing. In *Proc. CIKM*, 2002.

[3] 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*, pages 141–154, 2002.

[4] D. Draper, P. Frankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics, 2004.

[5] J. Ellson, E. Gansner, E. Koutsofios, and S. N. G. Woodhull. Graphviz and Dynagraph—Static and Dynamic Graph Drawing Tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[6] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing Cursor Movement in Holistic Twig Joins. In *Proc. CIKM*, pages 784–791, 2005.

[7] C. Mathis. Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility. *Informatik – Forschung und Entwicklung*, 21:3:147–164, 2007.

[8] C. Mathis and T. Härder. Hash-Based Structural Join Algorithms. In *Proc. DataX, LNCS 4254*, pages 136–149, 2006.

[9] C. Mathis, T. Härder, and M. P. Haustein. Locking-Aware Structural Join Operators for XML Query Processing. In *Proc. SIGMOD*, pages 467–478, 2006.

[10] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree Logical Classes for Efficient Evaluation of XQuery. In *Proc. SIGMOD*, pages 71–82, 2004.

[11] L. Qin, J. X. Yu, and B. Ding. TwigList: Make Twig Pattern Matching Fast. In *Proc. DASFAA*, 2007.

[12] C. Re, J. Siméon, and M. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. ICDE*, 2006.

[4]See: `http://www.graphviz.org/`