Rules for Query Rewrite in Native XML Databases

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

ABSTRACT

In recent years, the database community has seen many sophisticated *Structural Join* and *Holistic Twig Join* algorithms as well as several index structures supporting the evaluation of twig query patterns. Even though almost all XML query evaluation proposals in the literature use one of those evaluation methods, we believe that (1) there is no internal representation that enables a smooth transition between the XQuery language level and physical algebra operators, and (2) there is still no approach that considers the combination of content-and-structure indexes, Structural Join, and Holistic Twig Join algorithms to speed up the evaluation of twig queries. To overcome this deficit, we propose an enhancement to Starburst's *Query Graph Model* as an internal representation for XML query languages such as XQuery. This representation permits the usage of simple (binary) join operators—such as Structural Joins—and complex (n-way) join operators—such as Holistic Twig Joins—as part of the logical algebra. For twig queries, we define a set of rewrite rules which initiate query graph transformations towards improved processability, e. g., to fuse adjacent binary join operators to a complex join operator. To enhance the evaluation flexibility of twig queries, we come up with further rewrite rules to prepare query graphs—even before query transformation—for making the most of existing joins and indexes.

1. INTRODUCTION

The advent of XML as a de-facto standard for the exchange of structured and semi-structured data led to enormous efforts in the database research community to propose adequate *Path Processing Operators (PPOs)* for efficiently evaluating structural relationships (e. g., `child/descendant`). These PPOs were optimized by various kinds of index structures enabling fast access to element, attribute, and text nodes. They even allow to obtain answers to complete *twig query patterns (TQPs)*. The PPO algorithms can be further partitioned into two classes: *Structural Joins (SJs)* [1] and *Holistic Twig Joins (HTJs)* [3]. SJ algorithms decompose a TQP into binary structural relationships, evaluate each of those relationships separately, and finally "stitch" the results together. In contrast, HTJ algorithms evaluate TQPs as a whole.

Basically, the indexing algorithms proposed so far can be partitioned into four classes: path indexes, element indexes, content indexes, and hybrid indexes. *Path indexes* [13] are using structural summaries such as *Dataguides* [5] for providing efficient access to nodes satisfying structural relationships like `child/descendant`. *Element indexes* [3], which are indexing element nodes, serve for efficient input to SJ and HTJ operators. *Content indexes* [11] provide efficient access to text or attribute value nodes. They can be implemented very efficiently using B*-trees or inverted lists. Finally, *hybrid indexes* [17], which are also called *content-and-structure (CAS) indexes*, are a promising approach for indexing content and structure at a time. CAS indexes can contribute in a cost-effective way to the evaluation of components of—or even complete—TQPs. Therefore, they are a challenging competitor for SJ/HTJ algorithms.

Besides the various indexing approaches, there exist three different classes of XML algebras that allow for an algebraic optimization of XML queries: tree-based algebras such as *TAX* [8] or *TLC* [14], tuple-based algebras such as the *Natix Algebra (NAL)* [2] or NAL$^{\mathrm{STJ}}$ [9], and finally hybrid approaches like the proposal of Re et al. [16].

1.1 The Query Evaluation Process

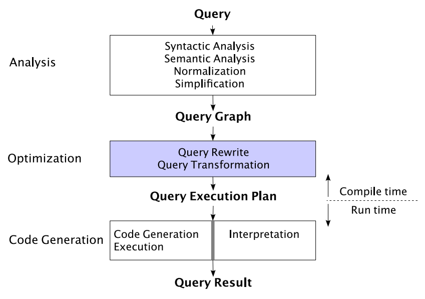

Figure 1: Overall query evaluation process

Figure 1 shows the three stages of the overall query evaluation process: *analysis*, *optimization*, and *code generation*. During the analysis stage, the query is checked for syntactical and semantical correctness. These checks are followed by a normalization phase, where semantically equivalent queries are mapped to a common normal form expression. The last step of this stage is formed by a simplification process that removes redundant parts of the query. The result of the first stage is delivered as a *Query Graph (QG)* which is equivalent to a logical algebra expression. During query rewrite, an algebraic optimization of the QG is performed by transforming it into a semantically equivalent structure which can be evaluated more efficiently than the initial expression. In the query transformation step, a rewritten QG is mapped to a *Query Execution Plan (QEP)* (physical algebra expression) using a heuristics-based or a cost-based plan generation process. The third stage, which is responsible for providing the query result, is executing the plan, either by direct interpretation or by translating it first to an executable module.

1.2 Problem Statement

Today, in the world of XML query languages, it is not clear how to perform a mapping from QGs to QEPs, because a divergence exists between the contributions of the algebraic optimization community and the proposals of the query evaluation community. Algebraic optimization leads to sophisticated ideas on normalizing and simplifying XQuery expressions, which are mostly driven by the formal semantics of XQuery. On the other hand, query evaluation primarily focuses on efficient evaluation algorithms for TQPs such as PPOs and indexes. To build query optimizers that are as powerful as their relational counterparts, this gap has to be closed by an appropriate internal representation that is flexible enough to handle both the node-at-a-time query processing inherent to XQuery Core expressions and set-at-a-time query processing performed by most PPOs and index operators. Additionally, this internal representation must support an effective mapping of logical operators to their physical counterparts. Using indexes for query evaluation as much as possible is not only a good heuristics for the relational world. We believe that an XML query evaluation engine should also follow this approach. Nevertheless, there exists—to the best of our knowledge—no query evaluation process which reduces the usage of PPOs for TQP evaluation to a minimum, provided there exist indexes that can deliver the results for components of—or even complete—TQPs.

1.3 Our Contribution

The contribution of this paper can be summarized as follows: We propose an enhancement to the Query Graph Model of Starburst—called the *XML Query Graph Model (XQGM)*—which is tailor-made for being used in the context of XML query languages such as XQuery. It provides a new logical n-way join operator which supports the evaluation of structural and positional predicates. We introduce a set of rules for query rewrite which allow to fuse as much adjacent binary SJ operators as possible to a single complex join operator. Those rewrites are solely performed at the logical algebra level and prepare the QG for a simpler mapping to a QEP. To make the most of available indexes, we define additional rules which permit to decompose n-way join operators during query transformation, if the query optimizer recognizes matching indexes. Using these rules, we can use existing indexes as far as possible and employ join operators only where it is absolutely necessary.

1.4 Related Work

Pirahesh et al. [15] introduced the *Query Graph Model* (QGM) in the context of Starburst as an extensible internal representation for relational queries. The data model of the QGM is strongly related to the relational data model which has to be adjusted to satisfy the needs of XML query languages. Additionally, their QGM does not offer any support for the evaluation of structural relationships expressed as XPath axis steps.

The classic work of McHugh and Widom [11] focuses only on optimizing path expressions using navigational access methods and lacks support for sophisticated indexing methods like CAS indexes.

Chen et al. [4] proposed *General Tree Patterns (GTPs)* as a generalization of TQPs enabling the representation of a large XQuery subset. Compared to our proposal, they perform a direct mapping of GTPs to QEPs. We believe that this approach is not very flexible, because some XQuery expressions consist of parts which can be mapped to GTPs and others that cannot be mapped to such structures. Our approach enables the evaluation of XQuery expressions having subtrees corresponding to TQPs and subtrees that do not belong to this class of query patterns.

Hidders et al. [7] are focusing on finding TQPs as early as possible during query optimization of XPath expressions. Their rewrite rules work on XQuery Core. The tree patterns found are directly mapped to the so-called *Tree Pattern Normal Form* which is their internal representation. By using XQuery Core as a baseline, they loose immediate access to the `where` clause of a FLWOR expression which makes the evaluation of value-based join conditions very hard. In contrast, our internal representation can deal with *structural*, *value-based*, or *positional* predicates.

Re et al. [16] introduce a set of rules for mapping XQuery expressions to their logical algebra. Additionally, they introduce a set of rewrite rules which enable an algebraic optimization. Unfortunately, they lack support for the evaluation of n-way joins and do not consider promising indexing approaches like CAS indexes.

Michiels et al. [12] propose a set of rewrite rules for XQuery expressions which have only one output node. Their rewrites are based on XQuery Core and consist of a normalization and a simplification step. During normalization, they map semantically equivalent XQuery expressions to a common XQuery Core expression. Simplification helps to remove unnecessary parts of XQuery Core expressions.

Mathis presents NAL^{STJ} [9] as an extension of the Natix Algebra [2], which introduces an SJ operator as a logical algebra operator. Additional rules for unnesting algebra expressions permit *set-at-a-time* query processing along with the *node-at-a-time* processing approach inherent to the nested version. Our work extends these ideas by introducing rules for fusing adjacent logical SJ operators to complex n-way join operators. This idea is driven by the evidence given in [3], that, in most cases, TQPs can be evaluated more efficiently using HTJ rather than SJ operators.

2. THE XML QUERY GRAPH MODEL

The *XML Query Graph Model (XQGM)* is inspired by Starburst's Query Graph Model [15]. All XQGM graphs are so-called *operator graphs*, where the nodes represent *Tuple Sequence Operators (TOs)*. The data flow of a query is described by the edges between the nodes.

We define a *tuple* as a mapping from a set of attributes to a set of values. Every attribute has an assigned type which corresponds to an XQuery node type or atomic type, e. g., `element` or `double`. A value can be formed by any atomic value in the value space of atomic types. All tuples, produced by a TO as output, form a so-called *tuple sequence* which is an ordered sequence of tuples. In contrast to the relational model, we additionally permit that an attribute value can be a sequence of tuples, too. We say a tuple is *nested*, if at least one tuple contains a tuple sequence as an attribute value. A TO is a generic object that consumes

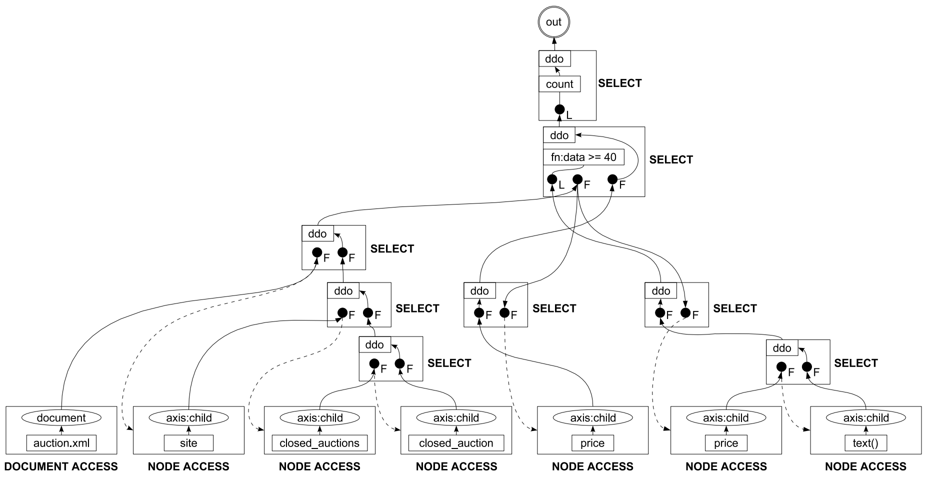

Figure 2: Query graph after translation

tuple sequences, transforms them according to its inherent evaluation strategy, and finally delivers a tuple sequence as output. In this paper, we concentrate on the following operators: *Select* filters tuples by means of value-based and positional predicates. Additionally, it supports the evaluation of aggregate functions. *Join* evaluates n-way structural joins. It is introduced after applying query unnesting as described by Mathis [9]. There exist various subtypes for the evaluation of semi-, anti-, and left-outer joins. The *Node Access (NA)* operator accesses a sequence of element nodes satisfying a given predicate. In contrast, the *Document Access (DA)* operator provides access to the root node of a document to supply an initial context for query evaluation.

All TOs inherit from a *Generic Tuple Operator (GTO)* that provides basic functionality needed to create QGs. They are permitted to add additional features such as new predicate types, e. g., for evaluating structural relationships. All TOs except the access operators consist of a *Head* and a *Body* (Figure 3). The Head forms a projection specification encompassing every attribute (*Head Attr.*) that is sent to a consuming operator[1].

An attribute calculation (*Attr. Calculation*) may call the `ddo` function. It has the same semantics as the `fn:distinct-doc-order` function introduced by the formal semantics of XQuery during query normalization. It sorts the outgoing tuple sequence in document order and performs duplicate elimination. The Body provides the procedural description of how output tuples can be derived from input tuples using an inner graph whose nodes are quantified tuple variables (*Tuple Var.*). The edges of this inner graph can be partitioned into three classes: *Definition edges* connect TOs with the head of the TO that provides input tuples for the current TO. *Predicate edges* describe predicate expressions defined for the connected tuple variables. *Output edges* connect tuple variables, which have to be made accessible to other TOs, with the head of the TO.

A *Correlated edge* connects tuple variables with the body of other TOs, for which the current evaluation context has to be provided, e. g., for evaluating existential predicates. Correlated predicate edges are drawn as dotted lines in the graphical representation. In contrast, all other edges are illustrated as solid lines.

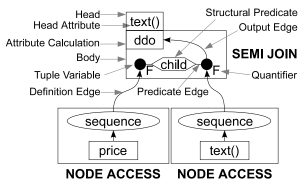

Figure 3: A simple query graph

Throughout the rest of this paper, we use the following example—Q5 of the XMark benchmark queries[2]—to show the application of our rewrite rules:

```
let $a := doc("auction.xml") return
count(
  for $i in $a/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price)
```

Figure 2 shows the QG—which is similar to a corresponding XQuery Core expression—after finishing the analysis stage of Figure 1. Consider the left subtree: The DA operator sends the root node of document `auction.xml` to the left-most select operator (incoming solid line) which sends it as evaluation context to the left-most NA operator (dotted line). Every `site` node that fulfills the structural relationship `doc("auction.xml")/site` is transfered to the next select operator. Every qualified `site` node provides the context for the evaluation of the following structural relationship `site/closed_auctions` and so on. Finally, the NA operator accessing `closed_auction` nodes propagates the qualified tuples to the cascade of consuming select operators. To enable a set-at-a-time processing rather than a node-at-a-time processing of the query, the unnesting rules presented by Mathis [9]—which we omit due to space restrictions—are used to introduce a logical SJ operator to the internal representation. Figure 4 shows the QG after unnesting. For the discussion of our rewrite rules, it will be used as a baseline and serves as input for the query optimization stage.

[1]In our graphical query representation, we hide the head of a TO, if it contains only one head attribute.

[2]`http://www.xml-benchmark.org/`

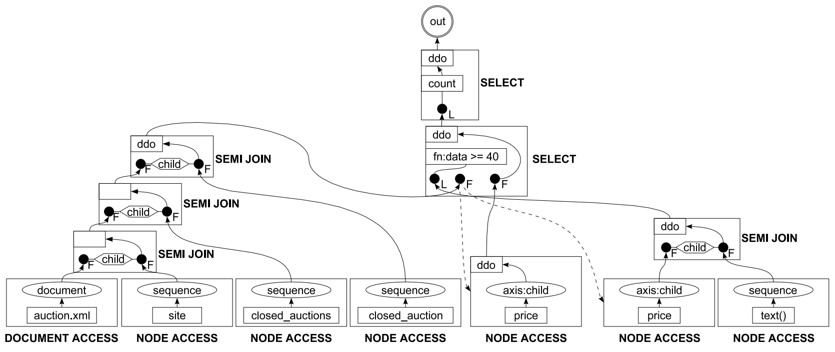

Figure 4: Query graph after unnesting

2.1 Semantics of the TOs

After discussing the constituents of a TO, we are now ready to take an in-detail look at the semantics of unnested QGs. A body of a TO can contain tuple variables with two different quantifiers: An F-quantified tuple variable works similar to the `for` expression of XQuery: For every tuple provided by an incoming output edge, it sends the tuple to a connected predicate. In parallel, it can provide the evaluation context for a nested sub-expression (dotted line). On the other hand, an L-quantified tuple variable corresponds to the `let` expression of XQuery. It provides the complete tuple sequence associated with it and does not iterate over it.

As mentioned in Section 2, we provide two different access operators: NA and DA. A DA operator provides the root node of a document as a context node that serves as a starting point for query evaluation. An NA operator supplies a sequence of all nodes fulfilling the selection predicate, e. g., a node name. For example, the left NA operator in Figure 3 provides all `price` element nodes found in the document.

The join operator is responsible for the evaluation of SJs and is introduced during the unnesting process as described by Mathis [9]. A join operator contains two or more F-quantified tuple variables. A *binary* join operator contains exactly two tuple variables that are connected to a structural predicate (XPath axis). The operator produces an output for each combination of incoming tuples (associated with the tuple variables), if they fulfill the structural predicate. We call a join operator *complex* if it contains three or more F-quantified tuple variables connected to structural predicates which may be linked to logical operators. We call a join operator a *semi-join* operator if at least one tuple variable is not connected to the head of the operator.

The select operator evaluates value-based and positional predicates which can be assigned to F- or L-quantified tuple variables. Additionally, a select operator can apply functions, e. g., `fn:count`.

Let us have a look at our running example which is shown in Figure 4: The left subtree provides a sequence of `closed_auction` tuples that satisfy the structural relationships evaluated by the semi-join operators. For each tuple, the lower select operator provides the evaluation context for the right subtree which establishes a sequence of tuples fulfilling the structural predicate `price/text()` w. r. t. to the evaluation context. Each `text()` tuple is sent to the top-most select operator if it fulfills the predicate `fn:data >= 40`. The top-most select operator applies the `fn:count` aggregate function to the tuple sequence and outputs the number of tuples in it.

3. RULES FOR QUERY REWRITE

Compared to query rewrite in relational database systems, our approach considers the characteristics of cost-effective PPOs (evaluation of TQPs as a whole) and indexes (evaluation of components of—or even complete—TQPs). For example, join operators can be fused to a complex join operator that can be efficiently evaluated using an HTJ or a CAS index access. Being semantics-preserving transformations, the rules for query rewrite are described according to the following textual representation: **IF** (Condition) **THEN** (Action). Condition describes the preconditions for applying the rule. In contrast, Action is a sequence of operations which have to be performed on the GTO and its subtypes. Each operation is described by a function call whose semantics is self-explanatory. The overall rewrite philosophy that drives the rewrite process can be condensed into the following statement: *Whenever possible, a query should be converted to a single join operator.* We believe that this strategy is helpful for the evaluation of TQPs, because they can often be evaluated more efficiently using HTJ operators rather than SJ operators [3].

3.1 Fusion of Join Operators

Rewrite rule 1 allows to fuse two adjacent join operators to a single—but probably more complex—join operator. It is only applicable for two adjacent join operators that evaluate structural predicates with `child/descendant` axes and which are connected over an F-quantified definition edge. The correctness of this rule is obvious: Fusing two adjacent join operators along the definition edge—by copying the body of the upper operator into the lower operator's one—does not change the semantics of the structural relationships, because we still perform a left-to-right evaluation of the predicates. Figure 5 shows the QG presented in Figure 4 after applying rewrite rule 1 which results in a complex 4-way join operator.

Rewrite rule 1. **IF** ((t is Join with F-quantified tuple variable v) $\wedge$ (v references b) $\wedge$ (b is Join) $\wedge$ (b and t only contain structural predicates evaluating child/descendant axes)) **THEN** ({CopyBody(b,t), CopyHead(b,t), CopyPredicateEdge(b,t), UpdatePredicateEdge(v), UpdateDefinitionEdge(v), Delete(b), UpdateHeadAttributes(t), UpdateAttributeCalculation(t)});

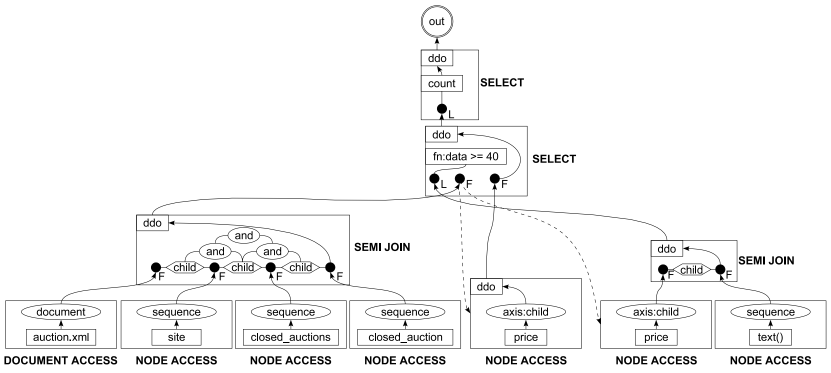

Figure 5: Query graph after join fusion

3.2 Fusion of Select and Join Operators

Rewrite rule 2 permits to fuse a join operator and an adjacent select operator. The correctness of the rule is obvious: Consider a join operator that receives its inputs from a select operator. The structural predicate is evaluated on a tuple sequence that has already been filtered by the select operator. After rewrite, the join operator evaluates the predicates added during the fusion, in addition to its own ones. Those additional predicates filter the tuple sequence in the same way, as it is done before rewrite.

Rewrite rule 2. **IF** ((t is Join with F-quantified tuple variable v) ∧ (v references b) ∧ (b is Select) ∧ (b contains no L-quantified tuple variable)) **THEN** ({CopyBody(b,t), CopyHead(b,t), CopyPredicateEdge(b,t), UpdatePredicateEdge(v), UpdateDefinitionEdge(v), Delete(b)});

3.3 Extending the Search Space for a Cost-based XML Query Optimizer

In the relational world, commutativity is an important property of binary join operators which allows to exchange the left and the right join partner. Rewrite rule 3 defines how we partially can make use of this property to provide an import operation for query transformation to extend the search space for join reordering. This rule is beneficial, e. g., for a hash-based structural join operator as described by Mathis and Härder [10], where an exchange of the left and right join partner may lead to better performance, because the hash table might be created for the smaller input sequence rather than the larger one. The commutativity rule holds for almost all XPath axes, except for the `attribute` axis, because it has no reverse axis. For all other axes, there exists a corresponding reverse resp. forward axis. Exchanging join partners for a join operator that evaluates a `self` axis is even trivial, because it is reflexive.

Rewrite rule 3. **IF** ((t0 is binary Join) ∧ (t0 contains tuple variables v1 and v2) ∧ (v1 and v2 are connected to a structural predicate p) ∧ (p does not evaluate the attribute axis) ∧ (v1 references TO t1) ∧ (v2 references TO v2)) **THEN** ({ExchangeRef(v1:t1, v2:t1), ExchangeRef(v2:t2, v1:t2), ReversePredicate(p), UpdateOutputEdge(v1), UpdateOutputEdge(v2)});

An associativity rule empowers a relational plan generator to perform movements in the search space of semantically equivalent queries by changing the join order. However, in the world of XML query languages, a single associativity rule is not sufficient, due to the dualism of content and structure. Instead, we will need a rule for reordering content-based joins and a set of associativity rules for structural joins that take combinations of several axes as well as early duplicate elimination and sorting into account. Every TO that forms the root node of a tree-structured query graph has to perform duplicate elimination and sorting, independent of its operator type. On the other hand, TOs that have incoming and outgoing edges potentially need to perform duplicate elimination. Fortunately, not every join operator needs additional duplicate elimination operations. For example, a full-join TO will not create any duplicates, independent of the structural predicate it evaluates. Due to space restrictions, we cannot discuss the large set of associativity rules we developed. Instead, we provide them as an appendix to this paper on the web [18].

3.4 Complex Join Decomposition

The main idea that justifies our rewrite philosophy of Section 3 is used for physical HTJ operators that can efficiently evaluate n-way joins. If the plan generator recognizes the availability of path or CAS indexes that can evaluate a branch of—or even a complete—TQP, this index should be used instead of evaluating the twig using a join operator[3]. To enable a decomposition of complex (n-way) join operators, we have to consider two cases, depending on how many twig paths can be answered using indexes. If there are several indexes available for a given path, then the one is chosen that can answer the largest fraction of the path. To make use of this index for query evaluation, we apply rewrite rule 4 to split the complex join operator into two parts: one part that is evaluated using the index and the other part that will be evaluated using an HTJ or an SJ operator.

Rewrite rule 4. **IF** ((t is n-way Join) ∧ (t contains one path that can be answered using an available index) ∧ (t contains tuple variables v0, v1, and v2) ∧ (v0 is the twig's root node) ∧ (all paths from v0 over v1 can be answered by no index) ∧ (at least one index can answer a path from v0 over v2)) **THEN**({CreateJoinOp(tb), MovePath(v0, v2, t, tb), UpdateHead(tb), InsertDefEdge(v0, tb)});

If we can answer more than one twig path using available indexes, then the path of those indexes overlaps at the twig

[3]Using HTJ or SJ operators for twig query evaluation always serves as a fallback strategy, if no index matches the twig.

query's root node. To get rid of this overlap, we use rewrite rule 5 to split the operator into two new join operators and connect them using the old join operator now evaluating a structural self-join to perform an intersection on the outgoing tuples of the newly created join operators.

Rewrite rule 5. **IF** (t is n-way Join) $\wedge$ (t describes TQP that can make use of at least two indexes) $\wedge$ (t contains tuple variables v0, v1, and v2) $\wedge$ (v0 forms the twig pattern's root node) $\wedge$ (v1 and v2 are child nodes of v0) $\wedge$ (v1 and v2 are root nodes of subtrees) $\wedge$ (for each path from v0 over v1 resp. v2 at least one index matches) **THEN**({CreateJoinOp(tl), CreateJoinOp(tr), MoveSubtree(t, tl, v1), MoveSubtree(t, tr, v2), InsertTupleVar(t, v3), InsertTupleVar(t, v4), InsertPredicate(t, self, v3, v4), UpdateHead(t), UpdateHead(tl), UpdateHead(tr), InsertDefEdge(v3, tl), InsertDefEdge(t4, tr)});

By applying rewrite rules 4 and 5 recursively on the QG, we obtain a QG that uses as much as possible existing path indexes or CAS indexes for query evaluation. To show the application of rewrite rules 4, we assume that there exist two different indexes which we can use for query evaluation: I_1(`//closed_auctions/closed_auction`) and I_2(`//price[String]`). The right subtree of the QG shown in Figure 5 can be directly mapped to an index access operator for index I_2. On the other hand, index I_1 cannot be used for query evaluation, yet. By applying rewrite rule 4, we get a QG where index I_1 can now be used for query evaluation.

4. CONCLUSIONS

In this paper, we proposed an extension to Starburst's Query Graph Model—namely the *XML Query Graph Model (XQGM)*—that serves as our internal representation for XML queries and permits a smooth transition between QPs and QEPs. To find TQPs as early as possible in unnested QGs, we introduced a set of rewrite rules which were driven by our rewrite philosophy: *Whenever possible, a query should be converted to a single join operator.* Using these rules, a query optimizer can evaluate TQPs using HTJ or appropriate CAS or path indexes. To make the most out of all existing indexes—especially those that can only answer parts of a TQP—we introduced further rules for join decomposition. Using these rules in combination with our knowledge on existing indexes allows for a separation of paths in TQPs, that can be answered using indexes, from those, that have to be evaluated using PPOs. In the future, we will integrate our rewrite rules in the query optimizer of *XTC (XML Transaction Coordinator)* [6] which is our prototype of a native XML database management system. To allow for a cost-based query optimization, we will develop an appropriate cost model that will drive the decision process for the application of the rewrite rules.

Acknowledgments

We thank Sebastian Bächle and the anonymous reviewers for their valuable comments on this paper.

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*, pages 141–154, 2002.

[2] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. ICDE*, pages 705–716, 2005.

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

[4] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. VLDB Conference*, pages 237–248, 2003.

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

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

[7] J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. How To Recognize Different Kinds of Tree Patterns from Quite a Long Way Away. In *Proc. Plan-X*, pages 14–24, 2007.

[8] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL, LNCS 2397*, pages 149–164, 2001.

[9] C. Mathis. Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility. *Informatik – Forschung und Entwicklung*, 21(3–4):147–164, 2007.

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

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

[12] P. Michiels, G. A. Mihaila, and J. Siméon. Put a Tree Pattern in Your Algebra. In *Proc. ICDE*, pages 246–255, 2007.

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

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

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

[16] C. Re, J. Siméon, and M. F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. ICDE*, page 14, 2006.

[17] 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.

[18] A. M. Weiner, C. Mathis, and T. Härder. Associativity Rules for Native XML Databases—Appendix. `http://www.xtc-project.de`, 2008.