Towards Cost-based Query Optimization in Native XML Database Management Systems *

© Andreas M. Weiner Christian Mathis Theo Härder

Databases and Information System 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 the last few years, XML became a de-facto standard for the exchange of structured and semi-structured data. The database research community took this development into account by proposing native XML database management systems for efficient and transactional management of XML documents. One of the most important factors for success of such systems is a powerful query optimizer. Many researchers proposed 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 approaches proposed so far use some of these methods, we believe that they provide no sufficient input for real-world cost-based query optimization scenarios, because they only cover a small part of the overall query evaluation process. To provide adequate input for a cost-based XML query optimizer, we propose the *XML Query Graph Model* as a new internal representation enabling a smooth transition between XQuery language level and physical algebra operators. Furthermore, we introduce a set of rewrite rules for improving the execution of twig queries, e. g., by fusing two adjacent binary join operators to a complex n-way join operator. By presenting further rewrite rules, we make the most of existing joins and indexes—even before query transformation. Using these concepts, we are ready to sketch its integration into our upcoming cost-based XML query optimizer.

1 Introduction

Nowadays, XQuery is the language of choice for evaluating queries—ranging from trivial to complex—on XML documents. In recent years, the database community suggested many *Path Processing Operators (PPOs)* for efficiently evaluating structural relationships—such as `parent/child` or `ancestor/descendant` as defined by twig query patterns:

Definition 1 (Twig query pattern) *A twig query pattern (TQP) is a tree $QT = (V, E, \lambda, r)$ with a set of vertices V, a set of edges $E \subseteq V \times V$, a mapping $\lambda : E \rightarrow \{child, descendant\}$, and the root of the tree r. Every TQP contains at least one output node, i. e., one or more vertices are part of the query result.*

The algorithms for PPOs can be further partitioned into two classes: *Structural Joins (SJs)* [1, 32, 22] and *Holistic Twig Joins (HTJs)* [5, 15]. 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.

PPOs were optimized by various kinds of index structures enabling fast access to element, attribute, and text nodes. They even allow to obtain answers for complete TQPs. Basically, the XML indexing algorithms proposed so far can be partitioned into four classes: path indexes, element indexes, content indexes, and hybrid indexes. *Path indexes* [25, 10, 9, 17] are using structural summaries such as *Dataguides* [11] for providing efficient access to nodes satisfying structural relationships like `parent/child` or `ancestor/descendant`. *Element indexes* [5, 8, 15], which are indexing element nodes, serve for efficient input to SJ and HTJ operators. *Content indexes* [23, 19] provide efficient access to text or attribute value nodes. They can be implemented very efficiently using B*-trees or inverted lists. Finally, *hybrid indexes* [30, 33, 16, 18], 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* [14] or *TLC* [27], tuple-based algebras such as the *Natix Algebra (NAL)* [3] or NAL^{STJ} [21], and

* This work has been supported by the Rheinland-Pfalz cluster of excellence "Dependable adaptive systems and mathematical modelling" (see `http://www.dasmod.de`).

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems, Saint-Petersburg, Russia, 2008

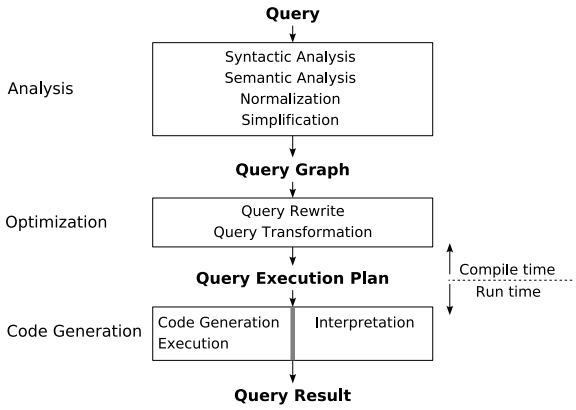


Figure 1: Overall query evaluation process

finally hybrid approaches like the proposal of Re et al. [29].

Whereas the database community has been doing research on query optimization of relational queries for 30 years, the field of XML query optimization is still in its infancy. We have learned from the experience on relational query optimization, that cost-based approaches outperform heuristic-based approaches in most cases. We believe that cost-based query optimization is also the method of choice for XML query languages like XQuery. To reach this goal, we have to define an overall process for query evaluation that considers all stages of the query evaluation lifecycle beginning with the translation of the query and ending with the provision of the final query result.

1.1 The Query Evaluation Process

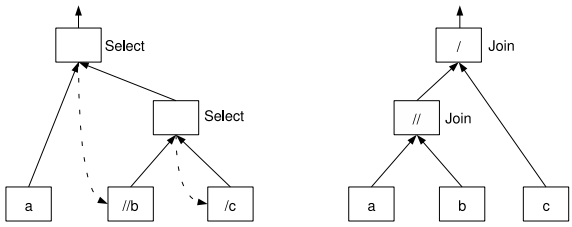
Figure 1 shows the three stages of the overall query evaluation process: *analysis*, *optimization*, and *code generation* [26]. During the analysis stage, the query is checked for syntactical and semantical correctness. These checks are followed by a normalization phase, where semantically equivalent 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 Evaluation Strategies for XML Queries

In general, there are two strategies for evaluating XQuery/XPath expressions: *node-at-a-time* and *set-at-a-time* processing.

Node-at-a-time evaluation is inherent to the *XQuery Core Language* and follows a nested-loops-style evaluation approach which is similar to sub-selects in SQL. Even though it is not very efficient in most cases, it can be beneficial in low-selectivity scenarios. Figure 2(a) shows how a simple XPath expression `a//b/c` is evaluated using this strategy: For every `a` node, the evaluation context for the evaluation of `//b` is provided (dashed line). By iterating over all qualified `b` nodes the evaluation context for `/c` is furnished. Finally, every qualified `c` node is output.

On the other hand, set-at-a-time query evaluation is similar to relational merge joins. It is employed by almost all SJ and HTJ algorithms and is in most cases very efficient. Unfortunately, its employment is not always possible, because SJ and HTJ operators provide only limited support for XPath axes (in most cases only `child/descendant` axes are supported). Figure 2(b) shows how the XPath expression `a//b/c` is evaluated using this approach: First, the structural predicate `//` is evaluated between all `a` and `b` nodes. Afterwards, the result of the first join operator serves together with all `c` nodes as input for the second join operator which evaluates the structural predicate `/`.



(a) Node-at-a-time (b) Set-at-a-time

Figure 2: Evaluation strategies for XML queries

1.3 Problem Statement

Today, there is—to the best of our knowledge—no query processing proposal that completely covers the query evaluation process described in Section 1.1. At an abstract level, all proposals can be partitioned into two classes: advances in algebraic optimization of XML query languages and sophisticated algorithms for twig query processing. Algebraic optimization leads to sophisticated ideas on normalizing and simplifying XQuery expressions [7, 29, 13, 24, 4], 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. As a consequence, even the analysis stage of the overall query evaluation process is not covered completely. To provide adequate input for the optimization stage, this divergence 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, it 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 approach which reduces

the usage of PPOs for twig query evaluation to a minimum, provided there exist indexes that can deliver the results for components of—or even complete—TQPs.

The effective solution of these problems is preconditional for providing appropriate input for query optimization in general and for cost-based optimization scenarios in particular.

1.4 Our Contribution

The contribution of this paper can be outlined as follows: We introduce the *XML Query Graph Model (XQGM)* as a new internal representation which is tailor-made for XML query languages such as XQuery. By introducing a new logical n-way join operator which permits the evaluation of structural and positional predicates, the XQGM is flexible enough to mediate between node-at-a-time and set-at-a-time processing approaches as well as for finding TQPs as early as possible. Using the XQGM, we provide adequate input for the optimization stage.

To allow for a simpler detection of TQPs and their subsequent mapping to QEPs, we propose a set of rewrite rules which allow to fuse as much as possible adjacent binary SJ operators to a single complex n-way join operator.

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 index structures. Using these rules, we can exploit existing indexes as far as possible and employ join operators only where it is absolutely necessary.

Finally, using the concepts proposed before, we sketch the architecture of our upcoming query optimizer.

1.5 Related Work

Pirahesh et al. [28] 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.

Carey et al. [6] as well as Beyer et al. [2] provide an XML query interface for object-relational database systems. Compared to our approach, which relies on a native XML storage rather than an object-relational one, using their systems makes it very hard to exploit state-of-the-art evaluation algorithms for TQPs as well as the plethora of different indexing approaches described in Section 1.

The classic work of McHugh and Widom [23] on optimization of XML queries only focuses on optimizing path expressions using navigational access methods and lacks support for sophisticated indexing methods like CAS indexes.

Chen et al. [7] 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 execution plans. 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. [13] are focusing on finding tree patterns as early as possible during query optimization by concentrating on XPath expressions having only one output node. 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 [4]. In contrast, our internal representation can deal with *structural*, *value-based*, or *positional* predicates.

Re et al. [29] 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. [24] propose a set of rewrite rules for XQuery expressions which correspond to single XPath expressions having only one output node. Their rewrites are based on the formal semantics of XQuery 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 [21] presents NAL^{STJ} as an extension of the Natix Algebra [3], 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 [5] that, in most cases, TQPs can be evaluated more efficiently using HTJ rather than SJ operators.

Brantner et al. [4] introduce a set of rewrite rules for XQuery expressions. Their approach consists of two stages: During the normalization stage they prepare XQuery expressions as input for the second stage. During the second stage, they try to merge inner and outer XQuery FLWOR expressions into a single XQuery FLWOR expression.

2 The XML Query Graph Model

The *XML Query Graph Model (XQGM)* is inspired by Starburst's Query Graph Model [28]. 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.

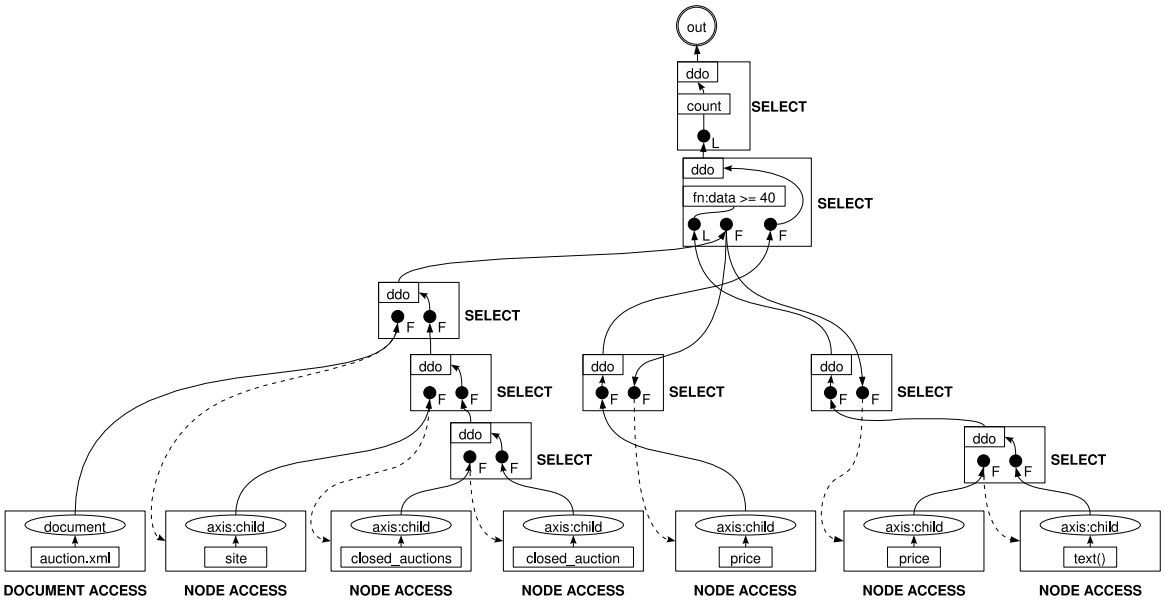


Figure 3: Query graph after translation

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 approve 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 tuple sequences, transforms them according to its inherent evaluation strategy, and finally delivers a tuple sequence as output. The following list shows the operators we use in the context of the XQGM:

- *SELECT*: Selects 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 [21]. There exist various subtypes for the evaluation of semi-, anti-, and left-outer joins.
- *ACCESS*: 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.
- *GROUP BY*: Groups tuple sequences according to a specific group predicate. It is also accountable for calculating group-wise aggregate functions.
- *UNION, INTSCT, DIFF*: These operators calculate from ordered tuple sequences order-preserving and duplicate-eliminating union, intersection, and difference.
- *SORT*: Performs additional sorting and duplicate elimination on tuple sequences.

All TOs inherit from a *Generic Tuple Operator (GTO)* that provides basic functionality needed to create query graphs. They are permitted to add additional features

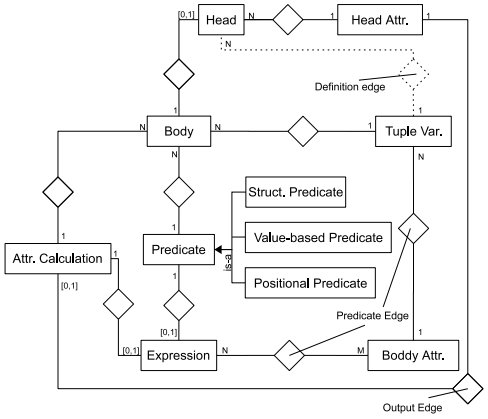


Figure 4: The constituents of the GTO

such as new predicate types, e. g., for evaluating structural relationships. Figure 4 shows how the constituents of the GTO relate to each other [26].

All TOs except the access operators consist of a *Head* and a *Body* (Figure 5). 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

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

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.

Throughout the rest of this paper, we use the following example—Q5 of the XMark benchmark queries [31]—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 3 shows the corresponding query graph built during the analysis step described in Figure 1. 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 [21]—which we omit due to space restrictions—are used to introduce a logical structural join operator to the internal representation. Figure 6 shows the query graph 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.

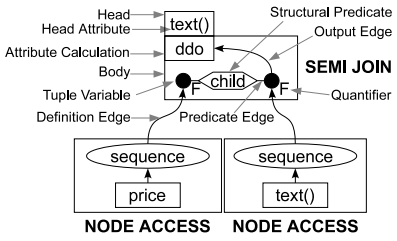


Figure 5: A simple query graph

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. 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: Node Access (NA) and Document Access (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 5 provides all `price` element nodes found in the document.

The join operator is responsible for the evaluation of structural joins and is introduced during the unnesting process as described by Mathis [21]. 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 is accountable for the evaluation of value-based and positional predicates which can be assigned to F- or L-quantified tuple variables. Additionally, a select operator can evaluate functions, e. g., `fn:count`.

Let us have a look at our running example which is shown in Figure 6: 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 [5].

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` or `descendant` axes and which are connected over an F-quantified definition edge.

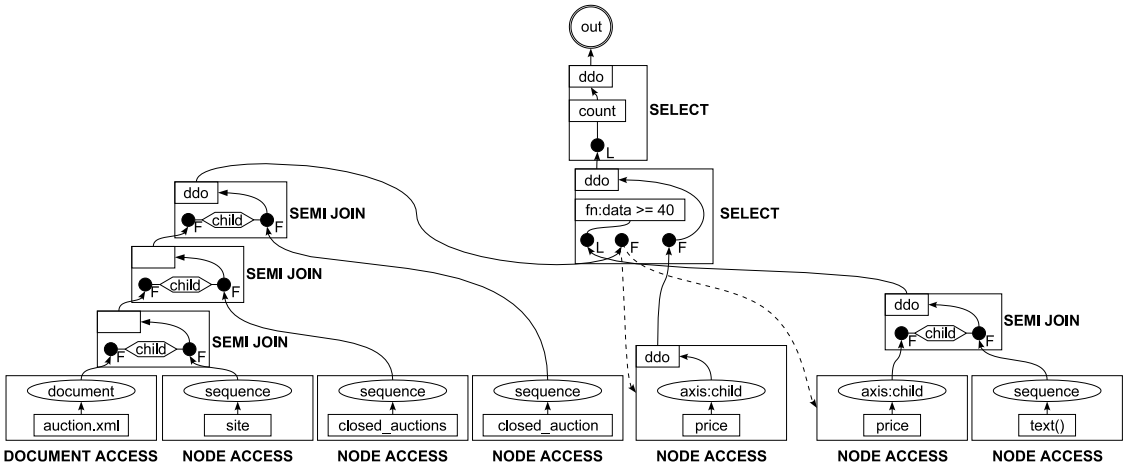


Figure 6: Query graph after unnesting

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.

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

Example 1 Figure 7 shows the query graph presented in Figure 6 after applying rewrite rule 1 which results in a complex 4-way join operator.

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 Fusion of Select and Node Access Operators

Rewrite rule 3 permits to merge a select and an adjacent NA operator by moving the predicates from the select operator to the body of the NA operator. This rewrite is beneficial, because an NA operator can be directly mapped to a physical element or CAS index access operator.

The correctness of this rewrite is preserved by the fact that the early evaluation of the selection predicate by the NA operator does not change the query semantics compared to filtering the output of an NA operator by the select operator.

Rewrite rule 3 **IF** *((t is Select with F-quantified tuple variable v) ∧ (v references b) ∧ (b is Node Access))* **THEN** *({CopyBody(t,b), CopyHead(t,b), CopyPredicateEdge(t,b), UpdatePredicateEdge(v), CopyOutputEdge(t,b), Delete(t)});*

3.4 Commutativity Rule

In the relational world, commutativity is an important property of binary join operators which approves to exchange the left and the right join partner. Rewrite rule 4 defines how we can partially 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 et al. [22], 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 4 **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)});*

3.5 Associativity Rules

An associativity rule empowers a relational plan generator to traverse 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

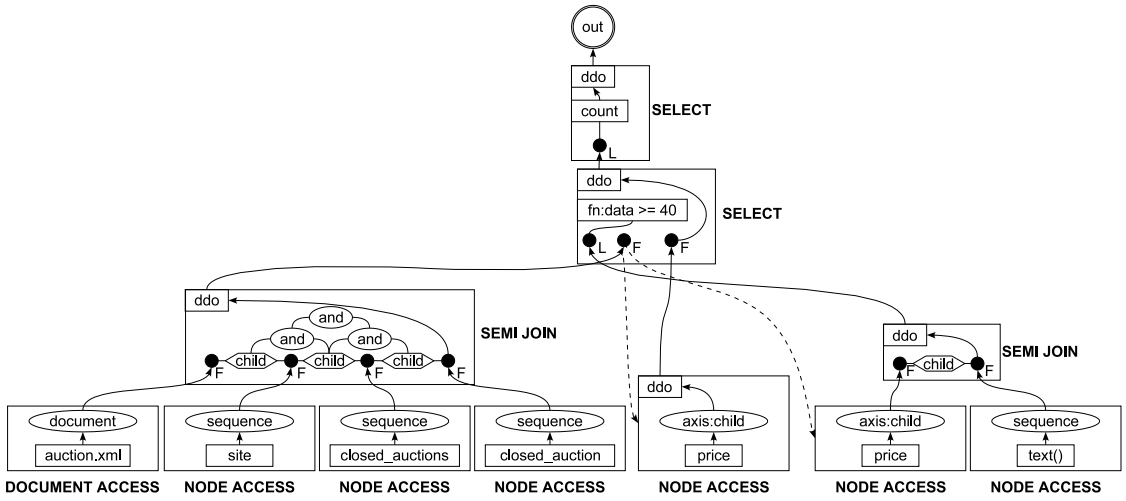


Figure 7: Query graph after join fusion

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. On the other hand, a semi-join TO can create duplicates on its output. We can partition binary semi-join operators into two different equivalence classes depending on the emergence of duplicates: (1) semi-join operators where only tuples of one incoming tuple sequence can contain duplicates after join evaluation (join operators that evaluate `parent/child` or `previous-/following-sibling` axes), and (2) semi-join operators where both incoming tuple sequences can contain duplicates after join evaluation (join operators that evaluate `ancestor/descendant` or `previous/following` axes).

Example 2 Let a denote the left join partner, b denote the right join partner of a binary structural semi-join j that evaluates the `child` axis. If j only produces a tuples satisfying the structural predicate, then duplicate elimination has to be performed, because every node can have multiple child nodes. In contrast, if j only delivers b tuples to consuming operators, then duplicate elimination is not needed, because every node has at most one parent node. If j would evaluate a `descendant` axis, then duplicate elimination could be necessary in both cases, because every node can have multiple descendants and multiple ancestors.

As mentioned before, to provide a complete set of associativity rules, all combinations of axes have to be considered. Additionally, different output nodes need to be taken into account. A node is called an *output node* if its tuple sequence contributes to the query result or is processed in a subsequent TO. Due to space constraints, we cannot discuss all associativity rules provided in our framework, except for one rule[2].

Figure 8 shows the associativity rule for one output node and two adjacent semi-join operators that evaluate the `descendant` axis[3]. To support a more fine-granular treatment of sorting and duplicate elimination, we replace a call to the `ddo` function, which only eliminates duplicates, by `D`.

We assume that the output of each join operator is implicitly sorted by the node that is used by a subsequent TO or that contributes to the final result. On the left hand side of Figure 8, a structural full-join is performed between tuples of TO A and B which needs no additional sorting or duplicate elimination. The following semi-join operator requires duplicate elimination and sorting for two reasons: (1) it has only incoming edges, (2) each tuple of the incoming tuple sequence can have multiple descendant c nodes. On the right side, a structural join is performed first between TO B and TO C. Because this structural relationship is evaluated using a semi-join, we need additional duplicate elimination, because every b node can have multiple c descendants. The following semi-join operator requires duplicate elimination for the same reason, but it needs no additional sorting because of our implicit sorting assumption.

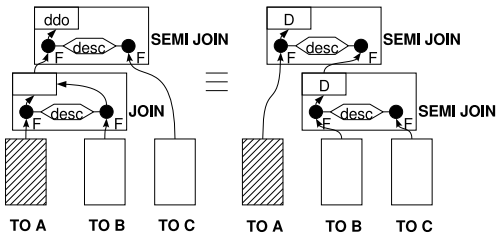


Figure 8: Associativity rule for two `descendant` axes and one output node

[2]The complete set of associativity rules can be found in Weiner et al. [34].

[3]This query graph corresponds to the following XPath expression: `a[.//b//c]`.

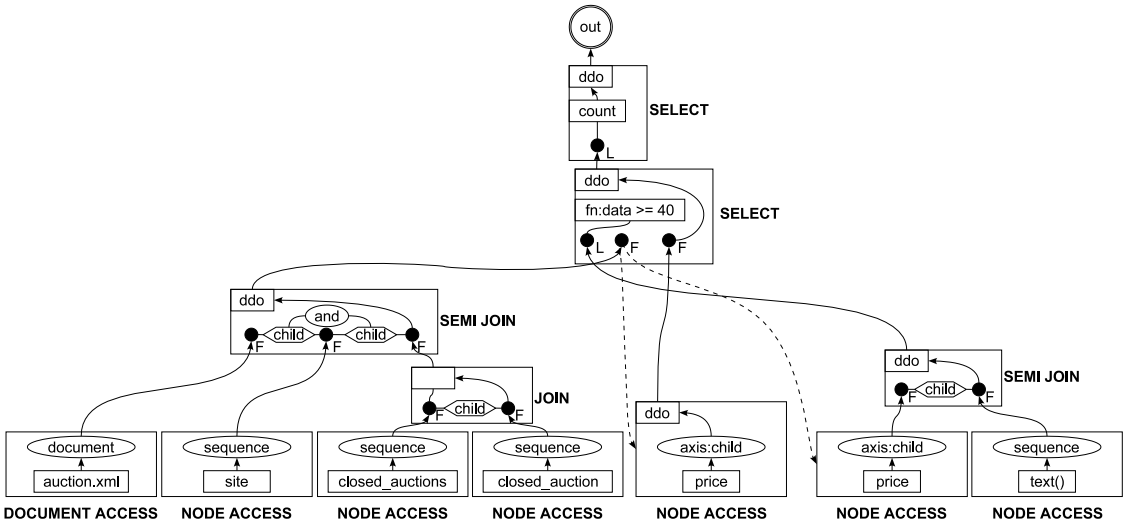


Figure 9: Query graph after join decomposition

3.6 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[4]. 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 5 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 5 **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 query's root node. To get rid of this overlap, we use rewrite rule 6 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 6 **IF** *(t is n-way Join) ∧ (t describes TQP that can make use of at least two indexes) ∧ (t contains tuple variables v0, v1, and v2) ∧ (v0 forms the twig pattern's root node) ∧ (v1 and v2 are child nodes of v0) ∧ (v1 and v2 are root nodes of subtrees) ∧ (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 5 and 6 recursively on the query graph, we obtain a resulting query graph that uses as much as possible existing path indexes or CAS indexes for query evaluation.

Example 3 To show the application of rewrite rules 5, 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 query graph shown in Figure 7 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 5, we get the query graph shown in Figure 9. As a consequence, index I_1 can now be used for query evaluation.

Considering the transformations illustrated in Figures 6 to 9, we can see how the rewritings push the QG towards a better starting point for query transformation by finding TQPs and exploiting existing indexes as early as possible. After query unnesting shown in Figure 6, the sequence of three semi-join operators was transformed into a single 4-way join operator (see Figure 7). Figure 8 sketched how associativity rules can be defined that help to enhance the search space for a cost-based query optimizer. Finally, Figure 9 exemplified how the knowledge on existing indexes helps to rewrite the QG w. r. t. a simpler mapping to index access operators during query transformation.

4 Setting-up an Infrastructure for Cost-based XML Query Optimization

After having introduced the XQGM as our internal representation for XQuery expressions and defined various rules for query rewrite, we are now ready to discuss the

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

infrastructure for cost-based XML query optimization which takes a rewritten QG as input and transforms it into a—hopefully—near-optimal QEP.

4.1 The Big Picture

Figure 10 shows the overall architecture of an XML query optimizer which will be integrated into our prototype of a native XML database management system called the *XML Transaction Coordinator (XTC)*[5]. The optimizer consists of five major components: The *Generic Pattern Matcher (GPM)* provides means for extensible rule-based pattern matching on arbitrary graph-structured trees. Using the GPM, a developer just describes the pattern to be matched in a declarative manner. As a consequence, the developer has only to implement the transformation code manually.

The *Rewriter* applies the rewrite rules described in Section 3 to the QG and uses the GPM for pattern matching.

The *Plan Generator* is responsible for the transformation of QGs to QEPs. Depending on the plan generation strategy employed, a heuristics-based or a cost-based optimization approach is chosen. If no cost model is present, a heuristics-based approach has to be followed by using a static set of transformation rules. On the other hand, using a cost model and statistical information, a cost-based query optimization can be applied. No matter which plan generation approach is used, we also use the GPM for pattern matching to allow for a generation of alternative plans.

One of the main ingredients of a cost-based query optimizer is the *Cost Model*. It uses the statistical information provided by the *Statistics* component for cost estimation. In contrast to classic cost-based query optimization, the cost model is not solely used during query transformation. If we step back and have a look at the overall query evaluation process, the following question arises even during the analysis stage: *Is it beneficial to perform a query unnesting or not?*—in other words, shall we follow a set-at-a-time rather than a node-at-a-time processing approach. This is only one example where costing information is needed even before query transformation.

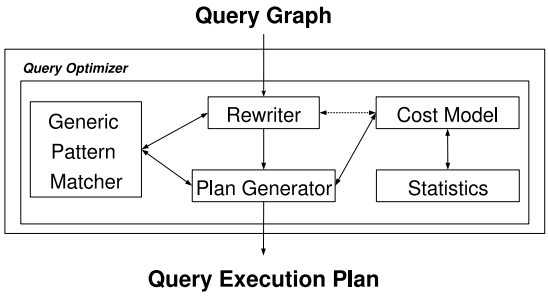


Figure 10: The architecture of the optimizer

4.2 Managing the search space

We have learned from classic relational cost-based optimization, that the search space—consisting of all semantically equivalent QEPs—for a plan generator gets tremendously large. If we consider cost-based optimization of XML queries, the situation gets even worse due to many different alternative evaluation methods for SJs and HTJs as well as different index-based access methods. To enable a query optimizer to keep as much as possible promising QEPs in main-memory, we use a kind of mesh data structure as proposed by Graefe and DeWitt [12]. Figure 11 shows how a new graph is stored in the mesh. On the left-hand side, a graph *Graph* and its corresponding mesh representation *Mesh* is shown. After applying query rewrite by fusion two adjacent operators to a new one, *Graph* is transformed into *Graph'*. The transformation causes an update of the mesh. To allow for space-efficient management of a large amount of different plans, only the newly added nodes are physically stored in the mesh. All other nodes are reused and referenced by virtual edges (dotted lines). The filled circles are associated with an equivalence class that encompasses all semantically equivalent subtrees. For example, the top-most operator in *Mesh'* contains two equivalence classes, where the first class has one representative and the second class contains two alternatives.

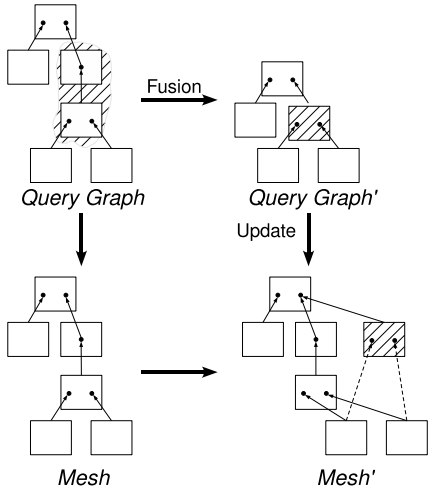


Figure 11: Updating the mesh

4.3 Choosing a Good Initial Plan

To reduce the time for finding a near-optimal query plan, we employ a two-phase query optimization approach: First we choose a good initial QEP using a heuristics-based approach. This QEP can either be immediately passed on to the code generation component or sent to a cost-based query optimizer.

4.3.1 Mapping of access operators

The XQGM access operators are providing input for consuming operators. In general, we can distinguish between two access methods: document scan and index access. A document access operator—e. g., the left-most access operator in Figure 6—is always mapped onto a document scan, because it only accesses the document's root node. For all other access operators, an element index scan—if present—is the access method of choice.

[5]Project website: `http://www.xtc-project.de`

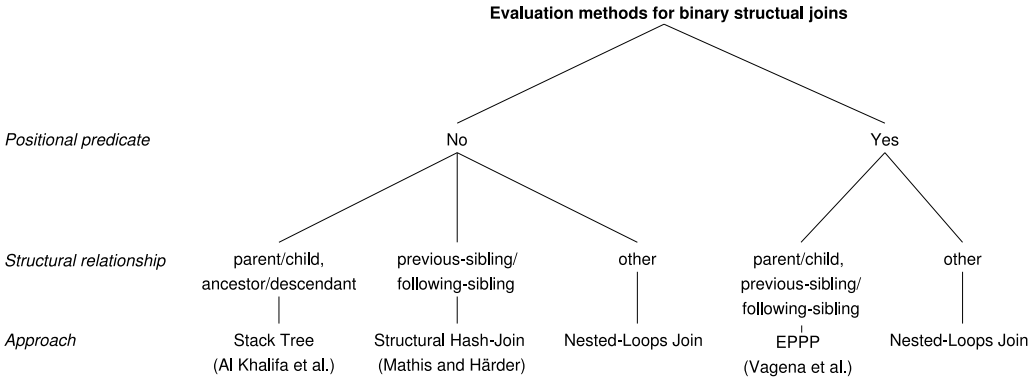


Figure 12: Decision tree for mapping join operators

4.3.2 Mapping of binary join operators

As outlined in Section 1, there exists a huge variety of different SJ and HTJ algorithms. Using the rewrite rules of Section 3, we can reduce the number of binary structural join operators to a minimum. If there are still binary structural joins, we use the decision tree shown in Figure 12 to choose the evaluation method. If positional predicates are missing, the join operator has three options depending on the structural relationship it has to evaluate: The *StackTree* algorithm by Al-Khalifa et al. [1] is used, when the join operator evaluates a `parent/child` or `ancestor/descendant` axis. For the evaluation of the `previous-sibling/following-sibling` axis, we use one of the structural hash-join algorithms proposed by Mathis and Härder [22]. In all other cases, we use a classic nested-loops join approach. On the other hand, if the binary structural join operator evaluates a positional predicate, we have two alternative evaluation methods: Vagena et al. [32] proposed *EPPP* which will be used for the evaluation of `parent/child` and `previous-sibling/following-sibling` structural relationships. For all other combinations, we also choose a nested-loops join algorithm.

4.3.3 Mapping of complex n-way join operators

Complex n-way join operators are created during query rewrite by fusing two adjacent binary structural joins. If there is no path or CAS index that can be used for the evaluation of a complex n-way join, we use the *TwigStack* algorithm proposed by Bruno et al. [5].

4.3.4 Making the most of available indexes

In Section 3.6 we showed how to prepare QGs to make the most of available indexes. During query transformation, we can benefit from join decomposition. Whenever a join operator—no matter whether binary or n-nary—and its subtree(s) can be evaluated using an available CAS or path index, we immediately map it to an index access operator.

4.4 Handling of Statistical Information by Profiles

During query transformation, we use a profile-based approach for managing statistical information as proposed by Mannino et al. [20]. Every QG is associated with a corresponding *profile hierarchy* which is formed by two different node types: *base profile (BP)* and *intermediate profile (IP)*. Figure 13 shows a QG and its associated profile hierarchy. Every QG leaf node (access operator) has its own BP which contains statistical information about the chosen physical access method. For example, an index scan contains information about the height of the search tree and the number of its leaf pages. Base profiles are periodically updated by the database management system and provide the foundation for cost estimation.

On the other hand, IPs are associated with QG inner nodes and do not provide accurate statistical information. Instead, they use the statistical information provided by their child nodes (BPs or BIs) as input for cost estimation. This is not done automatically by the database management system. Instead, a database administrator has to run a statistics collection tool similar to DB2's `runstats` manually to update the intermediate profiles. Using cost formulas which strongly depend on the assigned physical operator, they allow for an estimation of I/O and CPU costs.

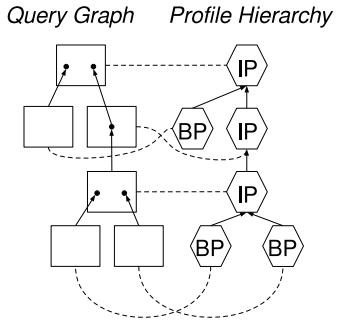


Figure 13: Query graph with associated profile hierarchy

5 Conclusions and Future Work

In this paper, we smoothed the way for cost-based XML query optimization. First, we introduced the XML Query Graph Model that serves as our internal representation for XML queries and permits an easier transition between QPs and QEPs. Second, we introduced a set of rewrite rules which allow to fuse as much as possible adjacent binary structural join operators to a single n-way join operator. Using these rules, a query optimizer can evaluate TQPs using HTJ or appropriate CAS or path indexes more easily. 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. Third, we sketched the architecture of our query optimizer that will be integrated in our prototype of a native XML database management system. Using the concepts described in this paper, we are already able to provide a heuristics-based query optimizer.

In the future, we will focus on the definition of an appropriate cost model. Choosing the right formulas for cost estimation is far from trivial. Compared to relational join algorithms—e. g., nested-loops join or merge join—SJ and HTJ algorithms are very complex in terms of how they process data. We expect that this leads to complex cost formulas whose quality can only be verified by extensive empirical experiments. Currently, we do not know which plan generation strategy, e. g., dynamic programming or simulated annealing, works best. For that reason, our plan generator will be flexible enough to switch between different strategies to allow for a comparison w. r. t. execution time, quality of the chosen plan and resource consumption.

Acknowledgments

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

References

[1] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, pages 141–154, 2002.

[2] Kevin S. Beyer, Roberta Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, Fatma Özcan, Hamid Pirahesh, Norman Seemann, Tuong C. Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *Proc. SIGMOD Conference*, pages 347–358, 2005.

[3] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. ICDE*, pages 705–716, 2005.

[4] Matthias Brantner, Carl-Christian Kanne, and Guido Moerkotte. Let a Single FLWOR Bloom. In *Proc. XSym, LNCS 4704*, pages 46–61, 2007.

[5] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD Conference*, pages 310–321, 2002.

[6] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *Proc. WebDB*, pages 105–110, 2000.

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

[8] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. VLDB Conference*, pages 263–274, 2002.

[9] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. In *Proc. SIGMOD Conference*, pages 121–132, 2002.

[10] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. VLDB Conference*, pages 341–350, 2001.

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

[12] Goetz Graefe and David J. DeWitt. The Exodus Optimizer Generator. *SIGMOD Rec.*, 16(3):160–172, 1987.

[13] Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. How To Recognize Different Kinds of Tree Patterns from Quite a Long Way Away. In *Proc. Plan-X*, pages 14–24, 2007.

[14] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL, LNCS 2397*, pages 149–164, 2001.

[15] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic Twig Joins on Indexed XML Documents. In *Proc. VLDB Conference*, pages 273–284, 2003.

[16] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *Proc. SIGMOD Conference*, pages 779–790, 2004.

[17] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proc. ICDE*, pages 129–140, 2002.

[18] Hua-Gang Li, S. Alireza Aghili, Divyakant Agrawal, and Amr El Abbadi. FLUX: Content and Structure Matching of XPath Queries with Range Predicates. In *Proc. XSym*, pages 61–76, 2006.

[19] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. VLDB Conference*, pages 361–370, 2001.

[20] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical Profile Estimation in Database Systems. *ACM Comput. Surv.*, 20(3):191–221, 1988.

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

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

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

[24] Philippe Michiels, George A. Mihaila, and Jérôme Siméon. Put a Tree Pattern in Your Algebra. In *Proc. ICDE*, pages 246–255, 2007.

[25] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proc. ICDT*, pages 277–295, 1999.

[26] Bernhard Mitschang. Anfrageverarbeitung in Datenbanksystemen (German only). Vieweg, 1995.

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

[28] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD Conference*, pages 39–48, 1992.

[29] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. ICDE*, page 14, 2006.

[30] Flavio Rizzolo and Alberto O. Mendelzon. Indexing XML Data with ToXin. In *Proc. WebDB*, pages 49–54, 2001.

[31] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conference*, pages 974–985, 2002.

[32] Zografoula Vagena, Nick Koudas, Divesh Srivastava, and Vassilis J. Tsotras. Efficient Handling of Positional Predicates Within XML Query Processing. In *Proc. XSym, LNCS 3671*, pages 68–83, 2005.

[33] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proc. SIGMOD Conference*, pages 110–121, 2003.

[34] Andreas M. Weiner, Christian Mathis, and Theo Härder. Associativity Rules for Native XML Databases. Internal Report, AG DBIS, `http://wwwlgis.informatik.uni-kl.de/`, 2008.