An Integrative Approach to Query Optimization in Native XML Database Management Systems

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

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

ABSTRACT

Even though an effective cost-based query optimizer is of utmost importance for the efficient evaluation of XQuery expressions in native XML database systems, such a component is currently out of sight, because former approaches do not pay attention to the latest advances in the area of physical operators (e. g., Holistic Twig Joins and advanced indexes) or just focus only on some of them.

To support the development of native XML query optimizers, we introduce an extensible cost-based optimization framework that integrates the cutting-edge XML query evaluation operators into a single system. Using the well-known plan generation techniques from the relational world and a novel set of plan equivalences—which allows for the generation of alternative query plans consisting of Structural Joins, Holistic Twig Joins, and numerous indexes (especially path indexes and content-and-structure indexes)—our optimizer can now benefit from the knowledge on native XML query evaluation to speed-up query execution significantly.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing—*XML, XQuery*

General Terms

Design, Experimentation, Performance, Theory

1. INTRODUCTION

Yet in the early days of XML database research, cost-based query optimization became an important issue in the context of *Lore* [16]. One of the big lessons learned from relational database systems is the inherent power of a declarative query language, which supports a user in describing *what* he is looking for, instead of *how* to get this information, in combination with an optimization infrastructure allowing for an efficient evaluation of such queries. Since the XML landscape has been shaped over the last decade, the world has changed a lot: new join operators and indexes have been introduced and XQuery has become the predominant XML query language. Even though XQuery is not completely declarative, it encompasses a large fragment of declarative language constructs. Using the novel join operators and indexes (Section 1.1), a plethora of possible query plans can be formed. Once again, by taking advantage of cost-based query optimization, we can provide reliable and efficient query plans for a large range of queries.

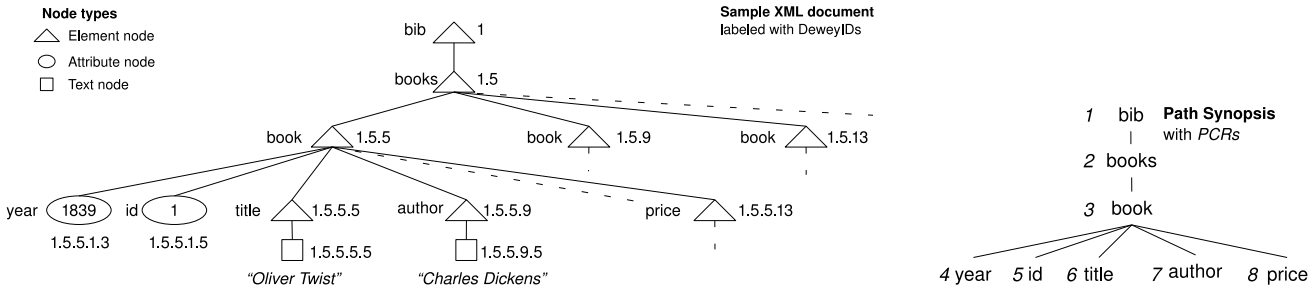
IDEAS10 2010, August 16-18, Montreal, QC [Canada]
Editor: Bipin C. DESAI

1.1 Background

Amongst others, *native XML Database Management Systems* (XDBMSs) provide a universal platform for efficiently storing, indexing, and querying XML documents of varying sizes, structures, and complexities. Structural relationships, e. g., `child` or `descendant`, play a dominant role in querying XML documents. Therefore, the research community proposed two different ways for evaluating such relationships: *Structural Joins* (SJs) [2] and *Holistic Twig Joins* (HTJs) [4]. The former class of operators evaluates structural relationships similar to relational sort-merge joins: To evaluate an XPath expression consisting of several location steps, each step is evaluated using a single SJ operator. By forming a cascade of SJ operators, this calls for classical join order optimization and access path selection. On the other hand, HTJ operators are n-way merge join operators that can evaluate path expressions—or even more complex patterns such as twigs—in a holistic manner. Here, only cost-based access path selection is performed.

Along with SJ and HTJ operators, several approaches for indexing XML documents were proposed. These methods can be partitioned into three equivalence classes: primary, secondary, and tertiary access paths. *Primary access paths* (PAPs) serve as input for navigational primitives as well as for SJ and HTJ operators. The most important representative of this class is a document index that indexes a document using its unique node labels as keys. Furthermore, a content index allows to access attribute and text content [16]. *Secondary access paths* (SAPs) allow for more efficient access to specific element nodes using element indexes [4]. They are mandatory for the efficient evaluation of structural predicates by SJ and HTJ operators. *Tertiary access paths* (TAPs) like path indexes [17] or content-and-structure (CAS) indexes [6] use a Dataguide [8] for providing efficient access to entire paths in a document.

In the XML world, path indexes and CAS indexes become *first-class citizens* for query evaluation, because they are as



(a) Sample XML document (b) Path synopsis

Figure 1: Sample XML document and corresponding path synopsis

powerful as SJ or HTJ operators. Using them to the full extent, allows to significantly reduce:

- *CPU cost*, because the time for deciding structural relationships by SJs or HTJs can now be saved
- *IO cost*, as only a single access path has to be scanned.

1.2 Problem Statement

To arrange the various SJ, HTJ, and access paths in such a way that the best possible plan is created, an optimizer's plan enumerator applies various equivalence rules for generating alternative plans. In recent years, only SJ reordering has been considered in cost-based XML query optimization settings [25]. Even though this approach is self-evident because it proved to be effective in relational query optimization, it does not pay attention to the various HTJ operators and indexes proposed in the literature. Today, we cannot be sure that SJ reordering is still the most effective strategy for XML query optimization. Therefore, the optimizer must be able to take advantage of both of them.

Although TAPs provide functionality comparable to materialized views in relational database systems and allow to quickly access all instances of a specific path in an XML document, recent optimization approaches do not consider them, too. As we have argued in Section 1.1, exploiting TAPs helps to reduce IO and CPU costs dramatically while making the overall QEP much simpler, because the number of operators is reduced significantly.

Currently, other authors are only focusing on cost-based XPath optimization. As XQuery is becoming the de-facto query language for XML, FLWOR expressions and other fancy language features add additional complexity and further increase the search space for optimizers tremendously.

1.3 Our Contribution

In summary, this paper contributes the following:

- We describe the *cost-based XQuery optimizer* implemented in the XML Transaction Coordinator (XTC)—a full-fledged native XDBMS.
- We provide a set of *plan-equivalence rules* that allow to take advantage of SJs, HTJs, PAPs, SAPs, and TAPs.
- We introduce a *cost model* that is used by our optimizer.
- We empirically evaluate the *reliability* of our cost model and assess the *effectiveness* of our optimization approach for XQuery/XPath queries.

1.4 Related Work

In this section, we give a brief assessment of relevant publications on relational cost-based query optimization, optimization frameworks, and XML query optimization.

Cost-Based Query Optimization in RDBMSs. Selinger et al. [22] proposed the first cost-based query optimizer, which was part of *System R*—the prototype of the first relational database system. The optimizer was capable of optimizing simple and linear SPJ (select, project, and join) queries. The authors introduced a simple cost model based on weighted IO and CPU costs and used statistics on the number of data pages consumed by relations to bind the cost model's variables to concrete values. Their dynamic programming algorithm initially selects optimal operator fittings for access paths. Thereafter, an optimal join order is determined based on a local optimality assumption. To early prune the search space, not all possible enumerations are taken into account. Instead, they only focus on interesting join orders, i. e., orders that do not require additional introductions of Cartesian products.

Graefe and DeWitt [10] presented the *EXODUS Optimizer Generator*. This system is not tailored to a specific data model and supports the specification of algebraic transformations as rules. Together with a concrete data model, these rules serve as input for an optimizer generator, which creates a tailor-made query optimizer.

The *Starburst project* [19] contributed important concepts to the emerging research field of query optimization. Among other things, the authors present the so-called Query Graph Model (QGM)—an extended relational algebra with a strong emphasis on structural relationships between language constructs. Beyond that, they introduced the concepts of rule-based query optimizers that can be easily modified by adding new rules for query transformation and query translation. Hence, this approach greatly improves the extensibility of such systems.

The *Volcano project* [11] as well as the *Cascades project* [9] are heirs of the EXODUS project. The authors distinguish between transformation rules, which serve for algebra-to-algebra transformations, and implementation rules describing the mapping from logical algebra expressions to operator trees. In contrast to the System R approach [22], they use a top-down query optimization algorithm that first takes a bird's eye view on QEPs.

Query Optimization Frameworks. Lanzelotte and Valduriez [14] contributed an extensible framework for query optimization that models the search space independent of a particular search strategy. Using this approach, developers can build highly-extensible plan enumeration frameworks.

Kabra and DeWitt [13] proposed *OPT++* as an object-oriented approach for extensible query optimization. By combining an extensible search component with an extensible logical and physical algebra representation, they lift the work of Lanzelotte and Valduriez to the object-oriented level.

XML Query Optimization. The classic work of McHugh and Widom [16] on the optimization of XML queries only targets at the isolated and strictly limited problem of optimizing path expressions using navigational access paths and lacks support for SJ and HTJ operators.

Wu et al. [25] proposed five novel dynamic programming algorithms for structural join reordering. Their approach is orthogonal to our work, i. e., it can be employed to choose the best join order in SJ-only scenarios. Compared to our work, they use only a very simple cost model for driving the join-reordering process and do not consider the combination of SJ and HTJ operators as well as different index-based access operators.

Zhang et al. [26] introduced several statistical learning techniques for XML cost modeling. In contrast to our work, which will follow a static cost modeling approach, they demonstrate how to describe the cost of a navigational access operator. Unfortunately, they do not cover set-oriented SJ and HTJ operators.

Balmin et al. [3] sketch the development of a hybrid cost-based optimizer for SQL and XQuery being part of *DB2 XML*. Compared to our approach, they evaluate every path expression using an HTJ operator and cannot decide on a fine-granular level whether to use SJ operators or not.

Che et al. [5] describe an XPath optimization framework that performs so-called deterministic (algebraic) optimization. They provide no cost-based optimization framework and do not use a full-fledged DBMS as testbed.

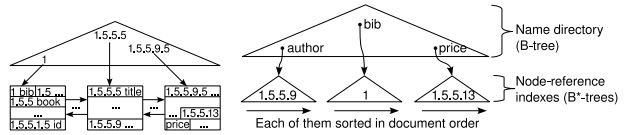
Georgiadis et al. [7] describe a first approach to cost-based XPath optimization. In contrast to our proposal, which supports the optimization of XQueries, too, they do not consider HTJs and advanced access paths like path indexes or CAS indexes.

2. PRELIMINARIES

Our XQuery optimization framework is part of the *XML Transaction Coordinator* (XTC)[1], which is our prototype of a native XDBMS providing full transaction support and differing APIs for accessing XML data (e. g., DOM, Sax, and XQuery). To understand the cost formulae (Section 3.2) of the various access paths, we first describe their physical representation in our system.

To enable efficient evaluation of structural predicates, a node labeling scheme is required, that assigns each node in an XML document a unique identifier that enables to decide for two given nodes the XPath axis they are related to—without further document access. As a prefix-based labeling scheme, *DeweyIDs* [18] qualify very well for this job. For example, Figure 1(a) shows an XML document that is labeled with DeweyIDs. By just looking at the DeweyIDs, we can see that `book` (label *1.5.5*) is a child node of `books` (label *1.5*), because they share the same prefix and their level information (total number of odd figures separated by ".") differs only by one.

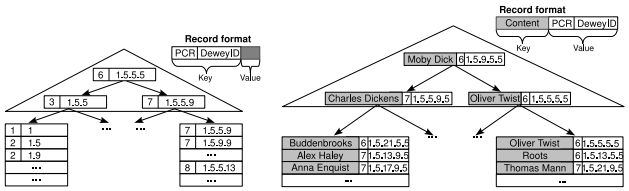
Figure 1(b) depicts the corresponding *Path Synopsis* (PS) for the document shown in Figure 1(a). A PS is a structural summary, which is actually an extended Data Guide [8], that is annotated with so-called *Path Class References* (PCRs). Every PCR refers to a unique path in the document. For example, PCR *3* refers to all `/bib/books/book` paths. Furthermore, the PS provides statistical information on the number of instances of each PCR. Figure 2(a) shows the *document index*. The document index is our PAP, i. e., our default access path, that simply indexes the complete document using the DeweyIDs as keys. Using a shared scan over it, inputs for SJ, HTJ, and navigational primitives can be provided. Figure 2(b) illustrates a SAP providing efficient access to specific element nodes. The *element index* is a two-stage index where element names (e. g., `book`) serve as keys in the *name directory*. Each entry points to a *node-reference index* that contains all nodes in document order having the same element name.



(a) Document index (b) Element index

Figure 2: Primary and secondary access path

Tertiary access paths provide advanced query processing capabilities. They can be considered as materialized views on paths in an XML document. Our system provides a path index and an additional content-and-structure (CAS) index, which both can have their values clustered by PCRs or DeweyIDs.



(a) Path index (b) CAS index

Figure 3: Tertiary access paths

Figure 3(a) shows a *path index* for our sample document. Here, the tuple (PCR, DeweyID) serves as key in the index. For example, if we want to evaluate the path expression `/bib//book/title`, we can use the knowledge on the document's structure provided by the PS. Thus, we can infer that we will find all nodes satisfying the path expression by scanning the path index for all records having PCR *6*.

Content-and-structure (CAS) indexes support paths that end on content nodes (attribute or text content). Here, the content node's value serves as key in the index. The indexed value is formed by a tuple (PCR, DeweyID) or (DeweyID, PCR), depending on the clustering. Figure 3(b)

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

illustrates a CAS index (with PCR clustering) that is defined for the content nodes of the path `/bib/books/book/title` and `/bib/books/book/author`, respectively. As we have already discussed in Section 1.1, CAS indexes are a potent means for supporting the efficient evaluation of point or range predicates on content nodes. They are extremely helpful, because, in contrast to structural relationships, which can be decided by just looking at the DeweyIDs of the involved nodes, the evaluation of value-based predicates would otherwise require more expensive accesses to a content index or—even worse—to the document index.

3. OPTIMIZATION FRAMEWORK

The sequence-based *XML Query Graph Model* (XQGM), which is an extension to the seminal Query Graph Model (QGM) [19], serves as solid foundation for our query optimization approach. The XQGM is equivalent to the XQuery Core Language (normalized version of XQuery without syntactic sugar) and supports the most important concepts of XQuery 1.0 such as FLWOR expressions and node construction.

After normalizing an XQuery expression, several algebraic rewrites (e. g., query unnesting) are applied to get rid of the node-at-a-time evaluation (especially all the nested `for`-loops for the evaluation of structural relationships) inherent to the XQuery Core Language. Furthermore, `text()` accesses are pushed up as much as possible. After query rewrite, we get an XQGM representation where almost all operators follow the classical set-at-a-time processing paradigm, because, now, all structural relationships are evaluated using logical SJs. We can pass on this representation to the query optimizer that selects the best join order, optimal operator implementations, and the cheapest access paths.

Due to the lack of space, we cannot give a full introduction to the XQGM. Instead, we only briefly sketch the XQGM representation for an XQuery expression[2].

In Figure 9 (Appendix), you can see a slightly simplified version of XMark benchmark query Q11. Each XQGM operator receives and produces (nested) sequences of DeweyIDs enriched with PCRs and probable content nodes. In addition to the Select operator provided by the classical QGM for the evaluation of value-based joins, the XQGM provides a logical SJ operator that joins two sequences of node IDs based on their structural relationships, which can be decided using their corresponding DeweyIDs. The SJ operators receive their initial inputs from Access operators that provide access to the node labels of element or attribute nodes. The Group By operators group their inputs by `doc("auction.xml")//person`. The Merge operator filters out all `doc("auction.xml")//person` subtrees that do not have `profile/@income` and `name` subtrees, respectively. Furthermore, for each `$p/name` that satisfies the merge predicate, the `text()` expression is evaluated.

The projection specification (`PROJ_SPEC`) describes which tuple sequences are passed on to the subsequent operator. Each operator is connected to a tuple variable (filled circles) that can have three different quantifiers: for (`F`), let (`L`), and exists (`E`). In this example, only `F` and `L` are used, which express the corresponding XQuery language constructs. For `F`-quantified tuple variables, a full iteration over their input is performed, whereas `L`-quantified tuple variables just bind the complete input sequences for predicate evaluation.

The top-most Select operator (below the output node) materializes the query result. The second Select operator evaluates a value-based join (as given in the `where` clause). Solid lines describe the direct data flow, whereas the dotted line refers to the provisioning of the current evaluation context. Here, the Select operator's left tuple variable receives the current evaluation context. For each sequence item provided by the context, the value-based join predicate is evaluated for each tuple received via the right tuple variable that provides all tuples that satisfy the following expression: `doc("auction.xml")//initial`.

This section is organized as follows: In Section 3.1, we describe the equivalence rules that enable the plan enumerator to generate alternative plans. Next, the cost model that allows the plan enumerator to prune expensive plans is presented in Section 3.2.

3.1 Plan Equivalences

At the beginning of query optimization, for every XQGM instance, a corresponding plan (which normally consists of subplans) is derived. A plan encompasses all *static properties* of the corresponding XQGM instance, e. g., structural predicates, projection specifications, or orderings that have to be preserved. In addition to that, each plan contains *dynamic properties*, e. g., required sorting on inputs, cost estimates, and the currently assigned physical operator. In contrast to static properties, dynamic properties may change during every state transition.

For a more convenient formulation of the plan equivalences, we use the following nomenclature: We denominate a plan and its properties by $\mathbf{P}[p_1, \ldots, p_n]$, whereas $\mathbf{P}$ is the type of the plan and the properties $p_1 \ldots p_n$ are enclosed in squared brackets. For our equivalence rules, we distinguish between the following types of plans:

$\mathbf{P}_i$	A plan of an arbitrary type
$\mathbf{A}_j$	Access plan (scan over a PAP or SAP)
$\mathbf{SJ}_t$	Structural Join of type t (omitted if irrelevant)
HTJ	Holistic Twig Join
TAP	Scan over a TAP

Every plan may have several of the following properties:

I	Currently assigned physical operator
P	Predicate (e. g., XPath axis)
D	Input sequences where duplicates must be removed
O	Sequences to be projected out

Consecutively, we use the following notation to describe the plan equivalences: $\mathbf{P}_1 \equiv_c \mathbf{P}_2$. This reads: plan $\mathbf{P}_1$ is equivalent to plan $\mathbf{P}_2$ iff the condition c is satisfied. To not compromise readability, we only give informal definitions of c.

Implementation Exchange. Let O_p be the set of all physical operators available to the optimizer. The implementation ($p_1 \in O_p$) of plan $\mathbf{P}$ can be changed iff there is another implementation $p_2 \in O_p$ that implements $\mathbf{P}$, too: $\mathbf{P}[\mathrm{I} : p_1] \equiv_c \mathbf{P}[\mathrm{I} : p_2]$

Structural Join Associativity. In contrast to the relational world, where a single join associativity rule is sufficient, the

[2]A formal definition of the syntax and semantics of the XQGM is provided by Mathis [15].

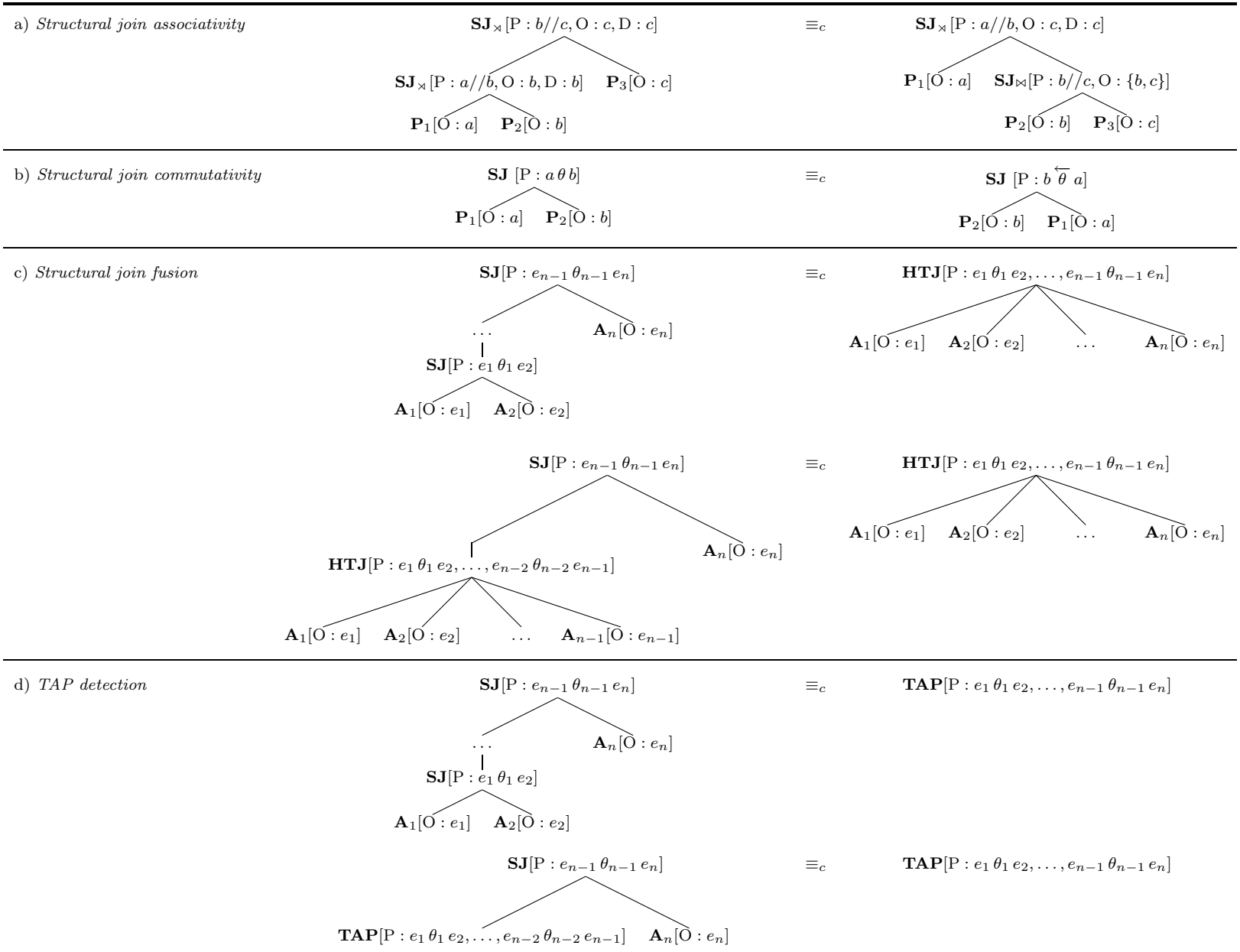


Figure 4: Structure-modifying plan equivalences

Operator	IO Cost	CPU cost
a) *Document index scan*	$\big[h(i) + \text{PCard}(i) - 1\big] \cdot \mathit{PageFetchCost}$	$\text{TCard}(i) \cdot \text{EvalCost}(p)$
b) *Element index scan*	$\big[h(i_n) + h(i_r) + \text{PCard}(i_r) - 1\big] \cdot \mathit{PageFetchCost}$	$\text{TCard}(i_r) \cdot \text{EvalCost}(p)$
c) *Path index scan*	$\big[h(i) + \lceil \text{PathSel}_\text{i}(e) \cdot \text{PCard}(i) \rceil - 1\big] \cdot \mathit{PageFetchCost}$	$\text{PathCard}(e) \cdot \text{EvalCost}(p)$
d) *CAS index scan*	$\big[h(i) + \lceil \text{PathSel}_\text{i}(e) \cdot \text{Sel}(p) \cdot \text{PCard}(i) \rceil - 1\big] \cdot \mathit{PageFetchCost}$	$\text{PathCard}(e) \cdot \text{Sel}(p) \cdot \text{EvalCost}(p)$
e) *StackTree*	$\text{Cost}_\text{IO}(\mathit{left}) + \text{Cost}_\text{IO}(\mathit{right})$	$\text{Cost}_\text{CPU}(\mathit{left}) + \text{Cost}_\text{CPU}(\mathit{right}) +$ $\text{TCard}(\mathit{left}) \cdot \text{EvalCost}(p)$
f) *NavTree*	$\text{Cost}_\text{IO}(\mathit{left}) + \text{TCard}(\mathit{left}) \cdot \text{Cost}_\text{IO}(\mathit{right})$	$\text{Cost}_\text{CPU}(\mathit{left}) + \text{TCard}(\mathit{left}) \cdot$ $\text{Cost}_\text{CPU}(\mathit{right}) \cdot \text{EvalCost}(p)$
g) *Extended TwigOpt*	$\text{Cost}_\text{IO}(\mathit{left}) + \text{Cost}_\text{IO}(\mathit{right})$	$\text{Cost}_\text{CPU}(\mathit{left}) + \text{Cost}_\text{CPU}(\mathit{right}) +$ $\text{TCard}(\mathit{left}) \cdot \text{EvalCost}(p)$

Table 1: Cost formulae

XML world makes things more complicated (orders must be preserved, early duplicate elimination). Therefore, we provide a set of *Structural Join Associativity* rules for different combinations of axes and output nodes [23]. For your convenience, we repeat the join associativity rule for two SJ operators evaluating the `//` axis and having the output node c. For example, this rule could be applied to reorder the following XPath expression `a//b//c`. In Figure 4 a, the corresponding plan equivalence is sketched. Here, arbitrary plans P_1, P_2, and P_3 provide the sorted input sequences a, b, and c, respectively. On the left-hand side, two right semijoins ($\rtimes$) are used to calculate the structural relationships. Because in both cases, the `//` axis is evaluated, potential duplicates (on b and c, respectively) are eliminated early. On the right-hand side, a full join between the sequences of plan P_2 and plan P_3 is performed. Next, the result is joined with sequence a.

Structural Join Commutativity. Figure 4 b, shows the *Structural Join Commutativity* rule. This rule allows to exchange the left and the right join partner of an SJ by just replacing the XPath axis θ by its reverse axis $\overleftarrow{\theta}$. Please note, this is only possible if θ is not the `attribute` axis.

Structural Join Fusion. Let $e := e_1\,\theta_1\,e_2, \ldots, e_{n-1}\,\theta_{n-1}\,e_n$ be a path expression with $\theta_1, \ldots, \theta_n \in \{/, //\}$. By applying *Structural Join Fusion*, which is illustrated in Figure 4 c, on a cascade of SJs that evaluates e, we can iteratively replace it by a single HTJ. It is worth noting that the query optimizer is not forced to make a binary decision between an exclusive evaluation of e using SJs *or* HTJs. Moreover, the query optimizer can decide on a fine-granular level, i. e., for every path step $e_i\,\theta_i\,e_{i+1}$, whether it is cheaper to integrate it into the previously formed HTJ or not. Hence, the creation of hybrid plans consisting of SJs *and* HTJs is possible, too.

TAP Detection. Until now, the plan equivalences of Figure 4 a–c are only able to create alternative plans consisting of SJs, HTJs, PAPs, and SAPs. Using them, we can already span a fairly large search space providing plenty of opportunities for optimization. Anyhow, we can now play our hand by considering TAPs. Using the *TAP Detection* rule, the plan enumerator can generate further alternative plans by replacing a cascade of SJ operators (as depicted in Figure 4 d) or a HTJ operator by a single scan over an appropriate CAS or path index. For a path expression $e := e_1\,\theta_1\,e_2, \ldots, e_{n-1}\,\theta_{n-1}\,e_n$ with $\theta_1, \ldots, \theta_n \in \{/, //\}$ and its corresponding PCR p_e, the query optimizer uses the PS to find out which indexes qualify as alternative access paths.

3.2 Cost Model

In this section, we introduce a subset of our cost model (Table 1 a–g) that is relevant for understanding our query optimization approach. The evaluation cost of an operator is described by its IO and CPU costs. We only consider the CPU costs, if alternative plans have equal IO costs. The IO cost is estimated using the total number of pages that must be loaded into a cold database buffer. The total number of items, which must be processed, serves as estimate for the CPU cost.

PageFetchCost	Cost for fetching a page from disk and loading it into the DB buffer
$\text{TCard}(x)$	Total # elements in x.
$\text{PCard}(x)$	Total # pages consumed by x
$\text{PathCard}(e)$	Total # instances of path expression e
$\text{Sel}(p)$	Selectivity of value-based predicate p
$\text{PathSel}_\text{i}(e)$	$= \frac{\text{PathCard}(e)}{\text{TCard}(i)}$
$h(i)$	Height of index i
$\text{EvalCost}(p)$	Evaluation cost of optional predicate p
$\text{Cost}_{x \in \{\text{IO,CPU}\}}(y)$	Cost x of child operator y

Table 2: Constant and functions

Table 2 shows the constant and functions used for the description of the various cost formulae. We currently use the EXsum framework [1] for cardinality estimation. In general, our approach can work together with arbitrary estimation frameworks. In particular, we chose EXsum, because it can provide reliable estimates even for rarely used axes like `parent` or `ancestor`.

Primary and Secondary Access Paths. In Figure 2(a), we illustrated the physical representation of the document index. Table 1 a shows the corresponding cost formula. For retrieving all element nodes with same name from document index i, we have to perform a full index scan. Foremost, we have to access the first leaf page resulting in $h(i)$ page fetches. Next, we need to scan all leaf pages consumed by the index $(\text{PCard}(d))$[3].

Compared to a document index scan, where the complete index must be read to get all element nodes having a particular name, the element index (Figure 2(b)) allows for fine-granular access to specific element nodes. In order to reach the first entry of a node-reference index i_r, we have to find the corresponding pointer in the name directory $h(i_n)$ and find the first leaf page $(h(i_r))$. Table 1 b shows the complete formula.

Tertiary Access Paths. Figure 3(a) shows a path index i with PCR clustering for three different paths (PCRs 3, 6, and 7). Let us assume path expression e selects all paths having PCR 6. Furthermore, we postulate that all records have the same length. Then we can estimate the fraction of leave pages we actually have to touch using the $\text{PathSel}_\text{i}(e)$ (Table 1 c).

For a CAS index, which allows for the evaluation of point as well as range predicates, Table 1 d shows the corresponding formula. For path indexes as well as for CAS indexes that are DeweyID-clustered, $\text{PathSel}_\text{i}(e) = 1.0$ always holds, because their entries are now clustered according to their order of appearance in the document rather than their containment w. r. t. a specific path class.

Evaluation of Structural Predicates. In the context of this work, the query optimizer can choose between two implementations for an SJ operator: *StackTree* [2] and *NavTree*. StackTree is a classical binary SJ operator. The NavTree is equivalent to a relational nested-loops join. The cost formulae for both operators are shown in Table 1 e and Table 1 f, respectively.

Besides binary join operators, our optimization framework provides an implementation for plans gained using Structural Join Fusion (4 c). The *Extended TwigOpt* operator—which adds, amongst others, grouping, evaluation of positional predicates, and support for negative predicates to TwigOptimal [12]—can be employed to evaluate path expressions or even more complicated structures like twigs in a holistic manner. Even though Extended TwigOpt is an n-way join operator, the cost formulae shown in Table 1 g depicts the binary case that can be easily generalized. The alert reader has already recognized that the cost formulae for StackTree and Extended TwigOpt are identical. They differ in the value provided by the function $\text{EvalCost}(p)$ that returns the evaluation cost for the structural predicate p. The authors showed in [24] how to define this function in such a way that whenever Extended TwigOpt outperforms StackTree, $\text{EvalCost}(p)$ is lower or higher, respectively. Using this approach, the query optimizer can decide for every binary structural relationship, whether it is cheaper to perform Structural Join Fusion or not.

[3]We always decrease the total number of pages fetches by one, because we assume that the first page has already been loaded into the buffer in the previous step.

4. PLAN ENUMERATION ALGORITHM

In the context of this work, we use a bottom-up optimization strategy that is similar to the plan generation approach of System R. Algorithm 1 sketches the plan generation algorithm. Plan generation starts at the leaf nodes of the query graph (Access Operators). For every Access Operator, the getSuccessors functions returns all possible access paths. Using the prune function, only the cheapest alternative is retained. Next, for every inner node (*currentPlan*) of the query graph (e. g., SJ or HTJ) the getSuccessors function creates all valid permutations (interesting SJ orders) of the subtrees residing on the *Goal* stack. For every combination, all matching plan equivalences (Table 4) are applied to *currentPlan*. The result is a set of equivalent plans differing in their implementation or arrangement of operators. The prune function estimates the costs of every plan $s \in \mathit{Successors}$ and returns only the cheapest one (s') that is marked as goal state and pushed onto the *Open* stack for the next iteration. All other states $s \in \mathit{Successors} - \{s'\}$ are dismissed. Finally, the algorithm terminates when the root of the query graph is reached and *Open* is empty.

In Section 5, you can convince yourself that our plan generation algorithm traverses even large search spaces (e. g., some XMark benchmark queries have up to tens of thousands alternatives) at moderate cost.

Algorithm 1: Search algorithm

```
Input: A stack Open with an initial plan
Output: The stack Goal containing the cheapest plan
        according to the cost model
1  Goal ← ∅;
2  while Open.size() > 0) do
3  |  currentPlan ← Open.pop();
4  |  if ¬ isGoal(currentPlan) then
5  |  |  Successors ← ∅;
6  |  |  Successors ← getSuccessors(currentPlan);
7  |  |  Successors ← prune(Successors);
8  |  |  foreach s ∈ Successors do
9  |  |  |  Open.push(s);
10 |  |  end
11 |  else
12 |  |  Goal ← Goal ∪ {currentPlan};
13 |  end
14 end
15 return Goal;
```

5. EMPIRICAL EVALUATION

Access path	*PageFetchCost* [ms]
Document Index Scan	0.78
Element Index Scan	2.48
Path Index Scan	2.45
CAS Index Scan	1.85

Table 3: Optimizer settings

Our experiments were done on an Intel XEON quad core (3350) computer (2.66 GHz CPUs, 4 GB of main memory, 500 GB of external memory) running Linux with kernel version 2.6.14. Our native XDBMS server—implemented using Java version 1.6.0_07—was configured with a page size of

16 KB and a buffer size of 256 16-KB frames. The experimental results reflect the average values of five executions on a cold database buffer. Table 3 shows the assignments to the constant *PageFetchCost* for the different access paths. All concepts described before, are implemented in the cost-based query optimizer of XTC. Because our optimization framework is rule-based, we can easily tailor it to our needs.

Access path	Query	f	Est. IO	Act. IO
Doc. index	Full scan	2.0	9,225	9,133
	Full scan	10.0	46,260	46,451
Element index	`//text`	2.0	329.84	320.20
	`//listitem`	10.0	828.32	894.80
	`//bidder`	10.0	748.96	721.80
Path index	Path p_1	10.0	115.15	125.80
	Path p_2	10.0	303.80	293.40
CAS index	Full, `income`	10.0	303.40	307.60
	Point, `income`	10.0	47.68	44.60
	Range, `income`	10.0	199.89	219.60

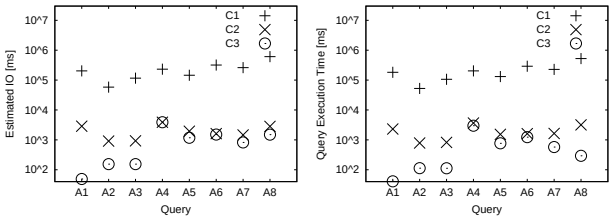
Table 4: Est. IO vs. actual IO on access paths

In our first experiment, we verify the correctness of our cost formulae (Table 1 a–d). Table 4 shows the experimental results for different access patterns on XMark documents with different scaling factors ($f = 1.0 \approx 100$ MB). For the document index, we performed full scans over the complete index. For the element index, we scanned three node-reference indexes of varying sizes. Furthermore, we defined path indexes on p_1=`/site/closed_auctions/closed_auction` and p_2=`/site/people/person`. Besides a full scan on the content of all `income` attributes, we also performed a point query (`income` $= 9,876.0$), and a range query ($20,000.0 \leq$ `income` $\leq 80,000.0$). On average, we get an error of less than 4%. Though our cost model is fairly simple, these results are promising.

		C1	C2	C3
Access path	Primary	✓	✓	✓
	Secondary		✓	✓
	Tertiary			✓
Plan equivalence	Impl. exchange	✓	✓	✓
	SJ associativity	✓	✓	✓
	SJ commutativity	✓	✓	✓
	Join fusion	✓	✓	✓
	TAP detection			✓

Table 5: Optimizer configurations

If not otherwise stated, for the remaining experiments, we query an XMark document with $f = 6.0$ (approx. 600 MB). To generate different plans, we use our bottom-up plan enumerator and three different optimizer configurations (Table 5). All of them can apply all plan equivalences except for the TAP detection. Configuration C1 can only use the document index as access path. In configuration C2 and C3, the optimizer can choose between the document index and element indexes. TAP detection is only allowed for configuration C3, because, here, path indexes and CAS indexes may be used in addition to the document index or element index. All TAPs were created using XTC's auto-indexing mechanism [21]. The results reflect the optimal plan according to the cost model, which was formed by using the available access paths and plan equivalences.



(a) Estimated IO (b) Evaluation time

Figure 5: PathMark-A results; $f = 6.0$

The second experiment tests our cost model on the PathMark-A queries[4] (simple XPath expressions). These queries are all XPath queries consisting of several steps, but do not contain accesses to `text()` nodes. Here, selecting the best access path is crucial. If you have a look at Figure 5, you can see that the estimated IO is a good indicator for the overall query execution time. Please note, even though the optimizer chose the best SJ order or used an appropriate HTJ operator, the estimated IO as well as the execution time is worse. Using C2 accelerates query evaluation up to 99% (compared to C1). For C3, the cost model recommended the use of path indexes for all queries. As the execution timings indicate (Figure 5(b)), this was a good decision, hence, the execution time could be significantly reduced further.

Name	Definition
B1	`doc('auction.xml')//asia/item[location= 'Germany']`
B2	`doc('auction.xml')//asia/item[location > 'C' and location <= 'G']`
B3	`doc('auction.xml')//text//*[keyword >= 'c' and keyword <= 'd']`
B4	`doc('auction.xml')//profile[@income > 40000] [age <= 19]`

Table 6: XPath queries with value-based predicates

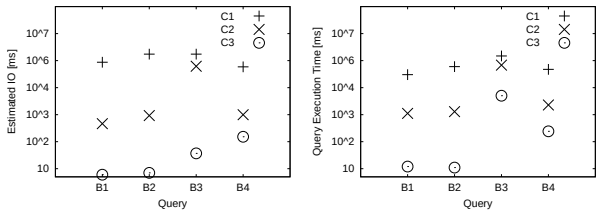
Hitherto, we only had a look at queries without value-based predicates that require additional accesses to the document, because they cannot be decided by just looking at their DeweyIDs. As discussed in Section 2, CAS indexes can help to overcome this problem. Table 6 shows the queries we used for this experiment. Here, the optimizer has the opportunity to exploit CAS indexes for the evaluation of a point predicate (B1), range predicates on element values (B2 and B3), and, finally, range predicates on attribute and element values (B4).

In Figure 6, you can see that the execution time for our query set is substantially reduced in configuration C3, because the optimizer uses CAS indexes. Though for query B3, the optimizer underestimated the IO costs—because the uniform distribution assumption did not hold—its decision proved to be the most efficient choice after execution.

In the previous experiments, you could convince yourself that our optimization approach is beneficial for XPath

[4]See: `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html`

queries. XPath is a subset of XQuery, thus, our optimization approach can also help to speed up the "access parts" of XQueries. We tested our optimization approach on the 20 different XMark benchmark queries [20], which make exhaustive use of XQuery features.



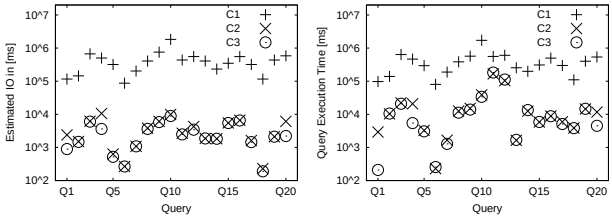
(a) Estimated IO (b) Evaluation time

Figure 6: Value-based queries; $f = 6.0$

For example, query Q19 is defined as follows:

```
let $auction := doc("auction.xml") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending
empty greatest
return
<item name="{$k}">
 {$b/location/text()}
</item>
```

Figure 7 shows the results we obtained. Only with configurations C2 and C3, acceptable execution times can be provided. In most cases, C3 found only slightly faster plans than C2. This does not happen because of wrong optimization decisions but, mainly, due to two reasons: (1) most paths are not very selective and (2) most plans consist of blocking operators that are absolutely necessary for query evaluation (e. g., tuple grouping or tuple unnesting) in our system.

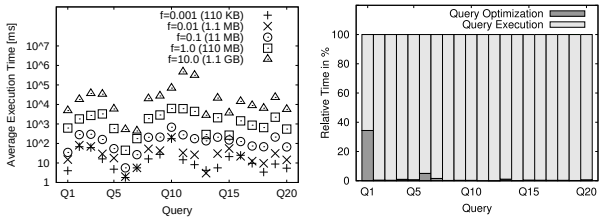


(a) Estimated IO (b) Evaluation time

Figure 7: XMark benchmark queries; $f = 6.0$

Figure 8(a) shows the scalability of the optimizer (configuration C3) for the XMark benchmark queries, which were executed on XMark documents whose sizes are ranging from 110 KB to 1.1 GB. Query Q11 and Q12 are very complex, include non-selective joins, and produce very large intermediate results that scale quadratically with the document size. Hence, the execution time of optimal plans increases quadratically, too. The execution times of the remaining queries scale linearly with the document size. For small documents (size $\leq$ 10 MB), the average scale factor is even at most 6.85, i. e., an increase of the document size by factor 10 results only in a 6.85 times longer execution time. For the largest document in our experiment (1.1 GB), we still get an average scale factor of 10.5 for all queries except of Q11 and Q12.

Finally, Figure 8(b) shows how query optimization (from query parsing to the generation of the physical plan) relates to the overall execution time in configuration C3. On average, 97.62% of the time is spent for query execution and only 1.54% of the time was consumed by query optimization. If no optimization is performed, the resulting plans are at least as worse as the ones gained using configuration C1 (Figure 7). Therefore, spending a small amount of the overall evaluation time on query optimization results in plans that are on average almost two order of magnitude faster than their unoptimized counterparts.



(a) Scalability (b) Optimization phases vs. execution time; $f = 6.0$

Figure 8: Scalability and optimization time

6. CONCLUSIONS AND FUTURE WORK

With this paper, we provide the basis for cost-based XQuery optimization in native XDBMSs. As shown in Section 5, our cost-based optimization approach substantially reduces IO costs for a large range of queries.

As our experiments revealed, efficient query evaluation on large documents is almost impossible without using SAPs and TAPs. In all our experiments, even join-order optimized plans using PAPs, were up to almost four orders of magnitude slower than the overall best plans. Therefore, it is a good heuristics to create at least SAPs on the complete document in order to gain acceptable results. Compared to TAPs, SAPs are heavy-weight data structures. Thus, update costs can be high if subtrees are inserted into the document that contain almost only structural parts. Our experiments showed that making high usage of TAPs is never a bad choice. Compared to SAPs, they are ultra light-weight. For example, only the content nodes are stored in a CAS index and the path information can be immediately derived using a PCR look-up on the path synopsis. Hence, each access to it reduces IO as well as CPU costs, because only a single access path is scanned and no structural relationships must be decided. Schmidt and Härder [21] showed that TAPs can be automatically created using a background job at acceptable costs. In our experience, it proved to be a good heuristics to always create CAS indexes on attribute nodes to make the evaluation of value-based predicates fast.

Some years ago, there was a grand debate on SJs vs. HTJs. According to our experience with our optimization framework, this argument ends in a draw. Having a look at the XMark benchmark queries—which are far more realistic than the queries used to compare the performance of SJs and HTJs—shows that both operators are thwarted by several

blocking operators, e. g., sorting and duplicate elimination, that are necessary to correctly evaluate XQueries and that dictate their "heartbeat".

For the XMark benchmark queries, the optimized plans for almost all queries scale linearly with the document size.

Our future work will focus on the optimization of the remaining XQuery language constructs (e. g., value-based joins), the integration of further physical algebra operators, and the development of a refined cost model that models CPU cost more precisely. Furthermore, we will integrate cost-based decisions even in earlier stages of the query evaluation process, e. g., during the XQuery-to-XQGM mapping.

7. REFERENCES

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

[2] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, pages 141–154, 2002.

[3] A. Balmin et al. Cost-based Optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[4] N. Bruno et al. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD*, pages 310–321, 2002.

[5] D. Che, K. Aberer, and M. T. Özsu. Query optimization in xml structured-document databases. *VLDB Journal*, 15(3):263–289, 2006.

[6] B. F. Cooper et al. A Fast Index for Semistructured Data. In *Proc. VLDB*, pages 341–350, 2001.

[7] H. Georgiadis et al. Cost Based Plan Selection for XPath. In *Proc. SIGMOD*, pages 603–614, 2009.

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

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

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

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

[12] H. Jiang et al. Holistic Twig Joins on Indexed XML Documents. In *Proc. VLDB*, pages 273–284, 2003.

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

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

[15] C. Mathis. *Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems*. PhD thesis, Univ. of Kaiserslautern, 2009.

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

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

[18] P. E. O'Neil et al. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD*, pages 903–908, 2004.

[19] H. Pirahesh et al. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD*, pages 39–48, 1992.

[20] A. Schmidt et al. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, pages 974–985, 2002.

[21] K. Schmidt and T. Härder. On The Use of Query-Driven XML Auto-Indexing. In *Proc. ICDE SMDB Workshop*, 2010.

[22] P. G. Selinger et al. Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD*, pages 23–34, 1979.

[23] A. M. Weiner et al. Rules for Query Rewrite in Native XML Databases. In *Proc. EDBT DataX Workshop*, pages 21–26, 2008.

[24] A. M. Weiner and T. Härder. Using Structural Joins and Holistic Twig Joins for Native XML Query Optimization. In *Proc. ADBIS*, LNCS 5739, pages 149–163, 2009.

[25] Y. Wu et al. Structural Join Order Selection for XML Query Optimization. In *Proc. ICDE*, pages 443–454, 2003.

[26] N. Zhang et al. Statistical Learning Techniques for Costing XML Queries. In *Proc. VLDB*, pages 289–300, 2005.

Simplified XMark Benchmark Query Q11

```
let $auction := doc("auction.xml") return
 for $p in $auction//person
 let $l :=
  for $i in $auction//initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
 return <items name="{$p/name/text()}">
  {count($l)}
  </items>
```

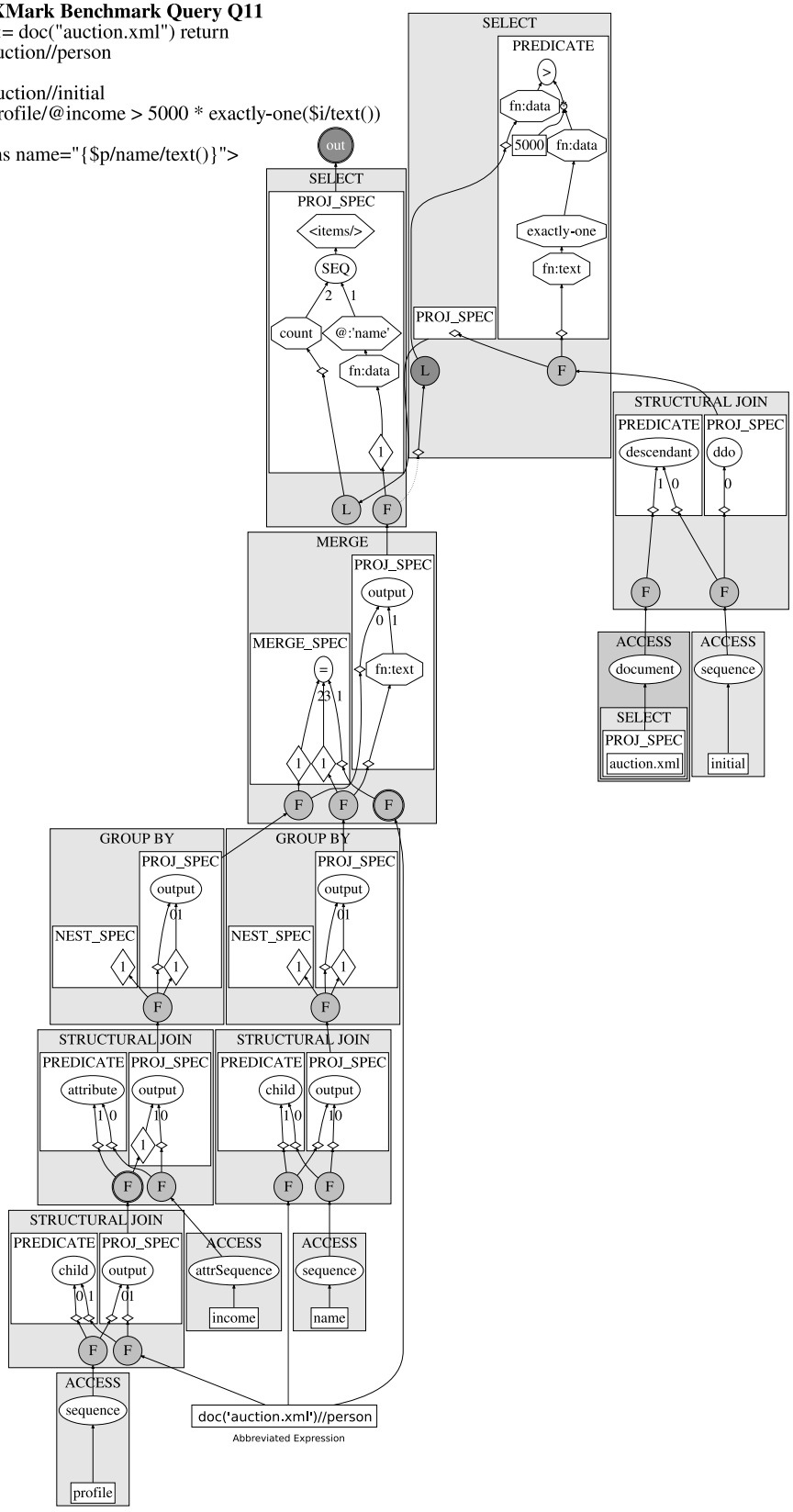


Figure 9: XQGM example