# XML Query Processing in XTC

Christian Mathis\* christian.mathis@sap.com SAP AG Walldorf, Germany

**Abstract:** In the past, the development of a declarative, set-based interface to access data in a DBMS was a key factor for the success of database systems. For XML, the lingua franca for declarative data access is XQuery. This paper summarizes the XQuery processing concepts that have been developed in the XTC system (the XML Transaction Coordinator)—a native XML database management system. We step through all query processing stages: from parsing over query normalization, type checking, query simplification, query rewriting, and plan generation to the execution.

## 1 Introduction

The eXtensible Markup Language (XML) was designed as a technique for document representation and data exchange. With the success of this meta language, the volume of data represented in XML grew steadily, resulting in large document collections. Keeping such collections serialized as text in files or as BLOBs in relational database management systems is clearly a bad idea. The process of parsing the relatively verbose XML representation upon access is too expensive. Furthermore, loading large XML instances into main memory is often not viable and multi-user access with updates cannot be efficiently supported without dedicated access mechanisms to document substructures. Therefore, in the last decade, tailored XML database management systems have been developed that can compactly encode XML documents, that enable the transfer of substructures of a document into main memory, and provide for ACID transactions. The XML Transaction Coordinator (XTC) [HH07] developed at the University of Kaiserslautern is a prototype of such an XML database management system (XDBMS). XTC is a so-called *native* XDBMS, because all its internal structures are tailored to XML storage and processing, in contrast to systems that map XML to relational tables for storage and processing. In the past, the development of a declarative, set-based interface to access data stored in a DBMS (e.g., SQL for relational systems) was a key factor for the success of database systems in general. For XML, the lingua franca for declarative data access is XQuery.

This paper summarizes the XML query processing concepts in native XDBMSs that have been developed in the author's doctoral thesis [Mat09]. It highlights all stages of the query

<sup>\*</sup>This work was conducted while the author was an employee at the Database and Information Systems Group (DBIS) at the University of Kaiserslautern.



Figure 1: Query evaluation in XTC

evaluation process: from parsing over query normalization, type checking, query simplification, query rewriting, and plan generation to the final execution. This approach to query processing resembles the "standard" query processing pipeline of relational query processors and, in fact, this work borrows quite some concepts. However, the semantic richness of the XML data model and the XQuery language requires new solutions at most stages and poses many interesting research problems. By building on the "standard" pipeline and standard techniques, the work from [Mat09] can be integrated in existing relational query processors, for example, to enable XML management in relational engines.

# 2 XML Query Processing on XTC—An Overview

Given a declarative query, the query processor has to generate a semantically equivalent, cost-optimal, procedural program, which consists of algorithms and database-specific access methods. In the following, we will sketch the process of XML query processing in XTC, from the external representation of a query in the XQuery language to the execution on the data store.

In the late 1980s and in the 1990s, the DB research community spent substantial efforts on the development of *extensible* query processors for database systems. The idea was to provide for a framework into which new concepts, such as new language constructs, new data models, or new processing algorithms could easily be integrated without the need to re-implement large portions of a query processor [Mit95, KD99]. Systems like EXO-DUS [GD87], VOLCANO [GM93, Gra94], and Starburst [MKL88, HFLP89, PHH92] are some well-known examples from that time. The query processor developed in [Mat09] stands in the tradition of these systems. Therefore, many concepts and terms could be

borrowed, and, although the XTC query processor was built from scratch, it can be seen as an extension in the sense of the idea of extensible query processing. To cope with complexity, query processing is generally split up into a number of stages. Each stage receives a query representation generated by some preceding stage (or given as input) and produces a further representation with a lower level of abstraction but enriched with more specific information on how the query has to be evaluated. Figure 1 depicts all query evaluation stages of the XTC query processor.

The process has a *logical abstraction layer* and a *physical abstraction layer*. The logical layer is completely system independent. The query representations and actions at this level can be reused to implement a query processor for another XML data source. The aim at this layer is 1) to find a procedural internal representation such that semantically equivalent (but syntactically different) queries are mapped onto the same representation, and 2) to rewrite the query in a way such that intermediate results are minimized. Such a representation layer below, because, in contrast to the declarative external query representation, a procedural internal representation about how the query can be evaluated. Furthermore, mapping semantically equivalent queries to the same internal representation makes the query processor robust.

At the physical layer, the query processor has to cope with low-level issues such as document storage layout, index structures, or processing algorithms to generate a program that operates on the database and efficiently computes the query result. In total, the query processor consists of the six components (see Figure 1): the *parser*, the *translator*, the *optimizer*, the *evaluator*, and the *metadata component* of the XTC system. Some of these components can share a sixth *infrastructure component*, which is not depicted in Figure 1. In the following, we give an overview over the various stages.

# 3 Parsing, Normalization, Static Typing, and Simplification

In the first stage, XQuery expressions need to be analyzed by a parser and to be converted into an *abstract syntax tree* (AST). In XTC, the XQuery grammar specified by the W3C Recommendation [BCF<sup>+</sup>04] is given to a parser generator to create the XQuery parser. In the next stage, the query translator transforms a given AST into an internal representation for the query optimizer. The translator has four stages: *normalization, static typing, simplification,* and *XQGM transformation.* Normalization and static typing are defined in the XQuery Formal Semantics Recommendation [CFS07]. Normalization transforms an XQuery expression to an equivalent expression in the XQuery Core Language, which is a subset of the original XQuery language. Static type checking derives the type of all subexpressions in the query and checks for static typing errors. The derived type annotations of all subexpressions can be used for optimization and restructuring.

Simplification aims at the removal of subexpressions with no effect on the query result. Such redundant constructs are sometimes introduced by programs that automatically generate queries, by view expansion, by users who do so accidentally, or by normalization.



Figure 2: Abstract syntax tree for XMark query Q5

Simplification is implemented using the *infrastructure component* of the query processor. This component interprets a query representation (in this case the AST) as a tree and employs a rule-inference engine to apply tree transformations that are specified by *restructuring rules*. A rule has a *pattern* and a *transformation instruction*. When a rule matches the tree representation, the transformation instruction is applied to rewrite the tree at that position. Because the infrastructure component is just an implementation aspect, it will not be introduced in detail.

To illustrate these steps, let us consider the following query that emanates from the XMark benchmark [SWK $^+02$ ] (Query 5) and returns the number of *price* elements that have a content larger than or equal to "40":

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

The abstract syntax tree produced by the parser for this query consists of roughly 40 nodes. For the sake of brevity, Figure 2 does not contain all these nodes, but only a fragment of the complete AST. As you can see, the representation is quite straightforward. Every particle from the XQuery grammar corresponds to a node in the AST.

Normalization translates the AST produced by the parser into a rewritten AST with the same semantics, but with a reduced set of language constructs. As a result, normalization removes syntactic sugar. The normalized version of the above query has the following form<sup>1</sup>:

<sup>1</sup>Note, this representation is simplified to facilitate comprehension. Function *ddo* stands for *fn:distinct-doc-order*, and—against the W3C recommendation—the constructs to produce positional information are omitted.

```
return child::closed_auction)))
where fn:data(ddo(
    for $fs:dot in $i
    return
        ddo(for $fs:dot in child::price
            return child::text()))) >= fn:data(40)
return
    ddo(for $fs:dot in $i
    return child::price))
```

You can observe that the normalized variant of the query does not contain any path expressions, only axis steps (e.g., child::site). Path expressions are rewritten to *for* clauses. The normalization process injects *ddo* and *fn:data* functions to ensure duplicate-free intermediate results (*ddo*) and atomic values for comparisons (*fn:data*).

Static typing infers the type of all subexpressions in a normalized query. For example, in the query above, the static type of the integer literal "40" is trivially *integer*. The surrounding *fn:data* function also delivers type *integer*, which is then used in the comparison. The comparison, in turn, is of type *Boolean*, and so on.

Even in our small example, you can observe that the normalization process is defined in a rather defensive manner, i. e., it injects certain functions blindly, even when they are not necessarily required. For example, the injected *fn:data* function around the integer literal "40" does not have an effect and can be safely omitted. A further example is the *ddo* function that is always injected, even when the intermediate result will always be in distinct document order. Besides normalization, users might write XQuery expressions with redundant or unnecessary subexpressions. Simplification aims at removing this kind of redundancy. An equivalent query for the above one might look like the following:

```
let $auction := doc(auction.xml)
return count(
    for Si in
      for $fs:dot in $auction
       return
           for $fs:dot in child::site
           return
               for $fs:dot in child::closed_auctions
              return child::closed auction
    where fn:data(
           for Sfs:dot in Si
           return
             for $fs:dot in child::price
             return child::text()) >= 40
    return
       for $fs:dot in $i
       return child::price)
```

The *ddo* functions are not necessary and the *fn:data* function around the integer literal can be removed<sup>2</sup>. Currently, the XQuery processor can detect simplification opportunities in various situations (see [Mat09]). Note, however, that the simplification logic aiming at removing *ddo* functions is not yet integrated (although this topic has already been discussed in the literature [FHM<sup>+</sup>05]). Since XQuery is a quite flexible and freely composable language, many more situations than those handled in this work allowing for simplifications might exist. This work does however not dwell further.

<sup>&</sup>lt;sup>2</sup>This is actually possible, because static typing revealed that the argument of the *fn:data* function is already an atomic value and therefore does not need atomization.



Figure 3: An example query represented in XQGM

# 4 The XML Query Graph Model

The last translation stage is the XQGM transformation. In this stage, the query translator transforms the AST into an instance of the XML query graph model. This is necessary, because the AST is not an appropriate format for query optimization as it lacks procedurality, i. e., it does not reveal data flow and control flow to evaluate the query. A better-suited internal query representation is the *query graph model* (QGM) introduced in the relational Starburst system [HFLP89, PHH92]. Although the QGM was designed for a relational engine, it provides enough flexibility to embed new language constructs like, for example, SQL recursion. In this work, we reused the QGM to support XML query processing. The resulting internal representation is called XQGM for *XML query graph model*. The initial XQGM instance for our sample query is depicted in Figure 4. All logical and physical plans presented in this and the following chapters are generated by a plan visualization tool developed in [MWHH08].

The syntax and semantics of XQGM can be found in [Mat09]. Here, we only give a brief introduction by example. Consider the query and its corresponding XQGM instance in Figure 3: An XQGM instance is an operator graph or a box-and-arrow diagram. Every box is a logical operator which produces data (most operators also consume data). The data produced flows along the arrows. All operators have a *name* describing the functionality of the operator and a unique *identifier* that follows the name in braces, e.g., "SELECT (2)". In the following, we use a lower-case font to refer to operator names. The graphical elements inside an operator specify how the operator processes input data

and how it computes results. For example, select (2) consists of four so-called *tuple variables* (depicted as circles) controlling the input data flow and creating a tuple stream, a *predicate* describing the selection expression on the tuple stream, a *sort specification* to modify the order of the tuple stream, and a *projection specification* defining how the output shall be computed. Tuple variables carry a quantifier (e.g., "F", for *for* quantification, and "L" for *let* quantification; see below) and a unique identifier to facilitate their distinction separated by a colon. The data model, based on which the semantics of XQGM is defined is similar to the XML data model (XDM) [FMM<sup>+</sup>04]. The major difference is that the XQGM data model allows tuples with nested tuple sequences.

To illustrate the semantics of XQGM, we step through the query execution of our sample query shown in Figure 4:

- Let us start with the control flow: The query processor calls the topmost select (1) operator, which, in turn, calls the next select (2) operator below to produce some output. Select (2) has three tuple variables, one of which carries an "F" specifying *for*-quantification semantics. The other tuple variables carry an "L" for *let*-quantification semantics. Tuple variables receive the output generated by their subgraphs. They define how this output is assembled into a stream of tuples. How this actually works will be sketched below. For now, we just proceed with the subexpression under tuple variable F:6. Select (3) is called and, in turn, access (5).
- Every operator calls its dependent sub-operators and awaits data for further processing. Access (5) is the first operator that actually produces data. It is a document access operator delivering the *virtual root node* [FMM<sup>+</sup>04] of the "auction.xml" document. This node is passed to the select (3) operator which binds it to tuple variable F:0 and calls select (6) to produce a result for tuple variable L:5.
- Select (6) in turn calls access (7), which is a navigational access operator. This type of access operator needs a *context node* as input from which the navigation starts. The context node is delivered by a *correlated input edge*, depicted as a dotted arrow. Tuple variable F:0 provides this input by passing the currently bound virtual root node to access (7). The result of the navigation on the child axis and the subsequent name test is a single *site* node. This node is passed to select (6) which binds it on tuple variable F:1 and calls select (8) to produce results for tuple variable L:4.
- Select (8) calls access (9) which delivers the *closed\_auctions* element (exactly one in every XMark document) using the current node at tuple variable F:1 as correlated input. The *closed\_auctions* element serves as correlated input for access (10) which returns all *closed\_auction* elements below. These elements are passed to tuple variable L:3 which collects them all, puts them into a sequence, and binds this sequence as the current value (which is actually the semantics of the *let* quantification).
- The sequence is then passed to the projection specification, which applies the *ddo* function. A tuple variable may either be referenced via a correlated edge (dotted arrow) or by a so-called *tuple variable reference* depicted as a rhomb. The *ddo* function is also applied in select (6) and select (3) passing the sequence of *closed\_auction* elements to tuple variable F:6.



Figure 4: XMark query Q5 represented in XQGM

• So far, every for-quantified tuple variable received only a single node as input. For single nodes, the semantics of for and let are the same. This time, however, tuple variable F:6 receives a sequence of possibly more than one node. While *let* passes these nodes as a whole as described above, for iterates over the sequence items, just like the corresponding constructs in the XQuery language. You can further notice that the subtrees below tuple variables L:11 and L:14 depend on the current node at tuple variable F:6, because these subgraphs have a correlated input edge starting at F:6. This means that for every node at F:6, the dependent subtrees are evaluated and their result sequences are bound to the corresponding tuple variables. In the following, we will call L:11 and L:14 dependent tuple vari*ables*, whereas F:6 is called *independent*. The subtrees below independent tuple variables have to be evaluated first, because they provide the input for the subtrees below dependent tuple variables. Essentially, the subtree below F:6 evaluates doc("auction.xml")/site/open\_auctions/open\_auction. For every open\_auction, the expression below L:11 evaluates the relative path price/ text() and L:14 the relative path price.

• Inside select (3), the predicate is evaluated for every *open\_auction* element. If the predicate evaluates to *true*, the current value at tuple variable L:14 is read by the projection specification and passed as an intermediate result to select (1). In turn, select (1) collects all these intermediate sequences in another sequence on which the *count* function is evaluated to obtain the final result.

The reader familiar with the dynamic evaluation phase specified in the Formal Semantics has noticed that the evaluation model defined there and in XQGM is essentially the same, i. e., an initial XQGM instance acts as specified in the Formal Semantics. This is meaningful, because it ensures correctness. In a way, XQGM is a graphical representation for normalized XQuery expressions. A large fraction of XQuery can be captured solely by XQGM's select and access operators. We will not formally introduce the syntax and semantics of XQGM in this paper. The details can be found in [Mat09].

# 5 Algebraic Rewriting

Because the semantics of an initial XQGM instance generated by the query translator adheres to Formal Semantics Recommendation, the above sketched evaluation model heavily relies on nested subexpressions and *node-at-a-time* navigational methods. This model it is often far from being optimal.

Therefore, besides classical algebraic optimizations such as selection push-down and select fusion to minimize intermediate results and the number of operators required, the algebraic rewriting stage tries to unnest queries as far as possible to enable bulk or setat-a-time processing. Unnesting substitutes correlated subexpressions by joins, i.e., by bulk operators. Like simplification, algebraic rewriting is also implemented using the infrastructure component. The XQGM instance is interpreted as a tree structure on which the generic rule engine executes rule-based transformations. The result of the algebraic rewriting stage is an unnested and pre-optimized XQGM instance. At this point, the physical optimization of the query begins and system-specific issues come into play. Before we discuss plan generation, we like to summarize the algebraic rewriting rules developed for the XTC query processor. [Mat09] contains all rules with the description of the rule pattern, its preconditions, and the transformation instructions:

- *Removal of external tuple variables*: Every variable reference in XQuery results in a tuple variable reference in XQGM. Some of these references are unnecessary and are removed by this rule.
- *Removal of descendant-or-self steps*: Due to normalization, a double-slash operation as in doc ("auction.xml") //item in XQuery results in a *descendant-or-self::node()* navigation in XQGM. Sometimes, this navigation step can be replaced by a *descendant* step, which is achieved by this rule.
- *Range-query detection*: Many XDBMSs provide index structures to evaluate contentbased predicates. Those predicates can be point queries or range queries. Rangebased predicates are specified in XQuery with the help of comparison operators and

Example	Twig	Document	Result	Tuple Result
a) plain		(a1)	$(a_1)$ $(a_1)$	
Plain twig nodes, order in document is not significant, all nodes produce output.		() () () () () () () () () () () () () (		[a <sub>1</sub> , b <sub>1</sub> , c <sub>1</sub> , d <sub>1</sub> ] [a <sub>1</sub> , b <sub>2</sub> , c <sub>2</sub> , d <sub>2</sub> ]
b) output ordering		0		
Plain twig nodes, order in document is not significant, all nodes produce output.	ۿ؋ۿ؋ڹ			$  \begin{bmatrix} a_1, b_1, c_1 \\ [a_1, b_1, c_2 ] \\ [a_1, b_1, c_3 ] \\ [a_2, b_2, c_1 ] \\ [a_2, b_2, c_2 ]  \end{bmatrix} $
c) boolean	$\sim$		and result	
Plain twig nodes and and/or twig node, all nodes produce output.	and/or C		(a) (b) (c) (c) (c) (c) (c) (c) (c) (c) (c) (c	$\begin{matrix} [a_1, b_1, c_1, d_1] \\ [a_1, b_2, c_2, 0] \\ [a_1, b_3, 0, d_2] \end{matrix}$
d) optional		0		
Plain twig nodes and optional edge (dashed): subwig rooted at 'b' is optional.	) ب ن ن			[a <sub>1</sub> , b <sub>1</sub> , c <sub>1</sub> , d <sub>1</sub> ] [a <sub>1</sub> , 0, 0, 0] [a <sub>1</sub> , 0, 0, 0]
e) grouping	<u> </u>	<u> </u>	$\bigcirc$	
Plain twig nodes and grouping node (double circle); subtwigs rooted at 'c' and 'd' grouped into result of 'b'.				[a <sub>1</sub> , b <sub>1</sub> , <c<sub>1, c<sub>2</sub>&gt;,<d<sub>1, d<sub>2</sub>&gt;]</d<sub></c<sub>
f) output	Contractor (1990)			
Plain twig nodes and grouping node (double circle); optional edge and output expression; only 'b' produces output				[ <x>{d<sub>1</sub>}{d<sub>2</sub>}</x> ]
g) output filter	test: fn:data([d]) = "XML"			
Plain twig nodes and grouping node (double circle); optional edge and output filter (predicate)			(đ.)	[d <sub>1</sub> ]
h) positional	red: cp = 1			
Plain twig nodes and one output node; two nodes generating positional information and two positional predicates	a) ppred: cp >		6)	[b <sub>1</sub> ]

Figure 5: Twig matching examples

the Boolean *and*. The range-query detection rule finds such range predicates and converts them into an XQGM range predicate, which is easier to evaluate and to map to the above mentioned index structures.

- *Select fusion*: Some rewriting rules leave select operations in a state, where only a simple operation, like applying the *distinct-doc-order* function, is executed. In these cases, the select fusion rule merges the select operation with its input operation.
- *Predicate push-down*: Predicate push-down is a standard rewriting strategy from relational query engines. It can also be implemented in XQGM for XQuery. Due to the existential semantics of general XQuery comparisons (see [CFS07]), predicate push-down is a little bit more complicated to implement.
- *Query unnesting*: The normalization phase introduces a nested sub-expression, whenever a variable is referenced. This is also reflected in the XQGM instance of the query. Especially for navigation axes, this approach leads to node-at-a-time evaluation, i. e., for every input node, the navigation axis is evaluated as a sub-expression. A similar situation arises in SQL queries with nested sub-queries. In almost all situations, these queries are unnested by the SQL processor and are replaced by a joinbased equivalent. This approach is also viable in XQGM. Here, however, we do not introduce value-based joins, but *structural joins*. For structural joins, many efficient implementations have been proposed in the literature (e. g., [AkPJ+02, CVZ+02, MHH06, MH06]). After the query has been unnested, all these algorithms can be applied. Besides from the discussion of query unnesting in [Mat09], an algebraic approach to query unnesting can also be found in [Mat07].
- Twig detection: Algorithms for twig pattern matching have been heavily researched in the past [BKS02, CLL05, FJSY05, CLT<sup>+</sup>06]. Twig matching algorithms can be used to evaluate branching path expressions that often occur in XQuery. To support twigs, XQGM specifies a dedicated twig operator. This operator has a so-called twig specification that can express twigs with various interesting properties that support idioms frequently occurring in XQuery expressions. Figure 5 exemplifies the semantics of the twig specification. The result is represented as graphical subtrees and as nested tuples (a data type of the XML algebra [Mat09] based on which XQGM is defined). Essentially a twig can return all nodes that match (Figure 5a). We can enforce that the result adheres to the document order [CFS07] (Figure 5b). The twig specification can define Boolean predicates (Figure 5c) and optional sub-patterns (Figure 5d). Some queries implicitly group results. Therefore, the double circle in Figure 5e signals that the matches below shall be grouped (in the tuple result, groups are represented by sequences in angle brackets). Furthermore, the XQGM twig specification allows to embed output expressions and filter predicates, for example, to generate new XML elements based on the matched results (Figure 5f) or to check content-based predicates (Figure 5g). Finally, even positional predicates can be specified (Figure 5h). The twig detection rule is responsible to find substructures in an XOGM instance that can be evaluated by a twig operation with the expressiveness of the sketched twig specification.

To illustrate the rewriting stage, Figure 6 shows the results on our running example. First you can observe that the query does not contain any nested subexpressions, it has been completely unnested (note, the dotted lines inside twig (28) have a different semantics).



Figure 6: Rewritten XMark query Q5

The access operators are not navigation-based anymore, but access *all* nodes that match a certain node test (e.g., all *price* nodes). The nodes in the twig specification are connected to the corresponding input tuple variables by dotted lines. In the specification, C stands for a child relationship, D for descendant, and @ for attribute. The select (1) operator has the same function as before. It collects the *price* sequences generated by the twig operator, adds them to a sequence, and applies the *count* function. Note, the completeness of the unnesting and twig detection rule has not been formally shown in [Mat09], i.e., we do not know, whether all twig queries can be unnested and whether all twigs can be found. Therefore, we classify the approach as best effort. Nevertheless, we note that all XMark queries [SWK + 02] could be unnested and all twigs were discovered.

## 6 Plan Generation

Given the result of the rewriting stage, the query processor now has to assemble a query execution plan (QEP), i. e., it has to map the logical operators onto algorithms and document access methods. These algorithms can roughly be grouped into 1. navigational, join-based, and index-based methods for path matching, and 2. into all remaining algorithms that are necessary to evaluate selections, projections, grouping, value-based joins,

etc. The algorithms of the first group, which are also called *path processing operators* (PPOs), play a major role in this work, because PPOs access the document (in contrast to the operators in the second group, which merely operate on the intermediate results delivered by path operators). Document access can be expensive, therefore, these operators need special attention. The set of all physical operators is called *physical algebra* (PAL). This term was introduced in [GM93] and shall help to distinguish operators from the physical level (algorithms) from operators on the logical level (XQGM). We give a brief introduction to the physical algebra in the next section.

Given an XQGM instance, plan generation is implemented in two stages, the first one of which also relies on the rule engine of the infrastructure component. Here, the rules describe logical-to-physical mappings or XQGM-to-Plan transformations (similar to [KD99]). Whenever a rule matches, a description of the physical operators implementing the matched XQGM operator is created and attached to the matched logical operator. Considering the relationship between a logical XQGM operator and operators from the physical algebra, the 1:1, 1:n, n:1, and n:m cardinalities apply: Sometimes there is only one physical alternative for a logical operator (1:1), sometimes there are more than one alternatives (1:n), and sometimes a group of logical operators is implemented by a (group of) physical operator(s) (n:1 or n:m). In the second stage, the plan generator iterates over the XQGM instance and builds different QEPs from the physical alternatives it finds. Often, the optimizer can create a large variety of structurally different but logically equivalent QEPs for a single XQGM instance.

From all the different QEPs, the query processor now has to decide, which of them is the cheapest in terms of processing costs. The answer to this question depends on a large variety of parameters, such as the optimization goal (e.g., response time, throughput, main-memory usage, etc.), the structural layout of the document, value distributions in the document, the current system state (I/O-bound or CPU-bound), and so on. The applicability of certain physical operators depends on the physical layout of the database, i.e., on document storage and indexing. At the time [Mat09] was written, cost-based query optimization was under development. Therefore, in [Mat09], the author restricted the plan generator to the following: 1. The plan generator should be able to generate every possible plan in the search space, and 2. the plan generator be able to find a good plan based on simple heuristics. We will come back to this point at the end of the next section.

# 7 The Physical Algebra

The physical algebra contains all query evaluation algorithms. Of particular importance are those algorithms that access the document or some index to match path patterns. Because XQuery heavily depends on efficient path pattern matching, we focus our discussion on path procession operators (PPOs). We distinguish *navigational*, *join-based*, and *indexbased* PPOs. The first group of operators is also the most expressive one; every path expression in a query can be evaluated by navigating the document. Compared to join-based and index-based methods, they are, however, often enough the group of operators with the slowest performance. Hence, navigational primitives are a "fall-back solution", when no operators of the other two groups can be applied to evaluate a certain path expression.



Figure 7: Sample document with path-based node labels

Join-based operators stream through the document or over an index that contains references to all elements and evaluate path expressions by matching structural relationships among the streamed nodes. Compared to navigational methods, they often provide for better performance. However, their use is restricted to certain XPath axes. The two most prominent representatives for this group are *structural joins* (STJ) and *holistic twig joins* (HTJ). Especially holistic twig joins have gained much attention in the literature and many variations of the original algorithm [BKS02] have been presented. Most of these variations aimed at optimizing the algorithm's structure matching phase and at increasing its performance. Below, we will sketch an HTJ algorithm that has been designed to work hand in hand with indexes and the other algebra operators.

The last group of operators provides index access. We will illustrate how path queries extracting inner elements (such as //cd[id="cd\_101"]) can be answered with path indexes and how index-based operators can be "married" with join-based operators. Note, index-based operators have yet again a reduced expressiveness compared to join-based operators, because join-based operators can match arbitrary branching path patterns and index-based operators can only match linear paths. In the following, we will give an overview over the PPOs in the physical algebra, starting with navigational PPOs.

#### 7.1 Navigational PPOs

Let us assume, a document like the one depicted in Figure 7 is stored in XTC's document store [Mat09]. Upon storage, the nodes are numbered using path-based node labels (also known as DeweyIDs or OrdPaths). These labels have certain salient features: they allow to compute all ancestor labels, their lexicographical order represents the document order, and they leave gaps to insert new nodes without altering existing labels. Given a node label, the document allows to retrieve other nodes, for example, all children, the complete subtree, the next/previous child, the parent node, and so on. These access primitives are used by navigational PPOs.

The *axis-step navigation operator* receives an axis, an XQuery node test, and a reference to the operation that provides its input node stream. For each node in the input, the algorithm queries the document store to evaluate the axis expression and applies the node test. Nodes that fulfill the test are passed on to the next operator. With this algorithm, all XPath axes and node tests can be evaluated. Note, this algorithm can be applied to evaluate all (but the left-most) access operations in Figure 4.

A problem with the axis-step navigation operator is that it might do work twice and that it might return duplicates in the wrong order (depending on the query and the document). As an example, consider XQuery expression d/descendant::w, where d is bound to sequence S. S is a series of context nodes that serve as starting points for the navigation. Let S contain two nodes u and v, where v is a descendant of u. Suppose that in the document, a node w exists, which is a descendant of both u and v. On the evaluation of axis step d/descendant::w, the above algorithms would return w twice (because they are evaluated both on u and v). However, XQuery semantics demands that the result of an axis step is duplicate free. Therefore, *distinct-doc-order* functions are embedded during normalization to sort the result and remove duplicates. The *multi-node navigation operator* avoids sorting and duplicates by analyzing the input sequence and only navigating from nodes that produce a duplicate-free result in document order. The details of this and the following algorithms are omitted in this paper. We refer to [Mat09].

### 7.2 Join-Based PPOs

Navigational PPOs require some input node(s), which serve(s) as a starting point for the navigation operation over the document. In contrast, join-based PPOs do not directly access the document. They operate on *two or more* node streams and are capable of finding path matches in these streams. How the node streams are created does not matter. They could be the result of a document scan, an index scan (see below), or they could be the result of other operators. XTC implements variations of two well-known algorithms: the structural join (STJ) and the holistic twig join (HTJ) algorithm.

The structural join in XTC is a merge-join algorithm, which is an extension of the original StackTree operator [AkPJ<sup>+</sup>02]. As an example, consider the two tuple streams containing *track* elements and *title* elements (see Figure 7). The *track* stream has the following node labels: 1.3.11.3, 1.3.11.5, 1.3.11.7, and so on. The *title* stream has labels 1.3.3, 1.3.11.3.3, 1.3.11.5.3, and so on. Suppose we want to find all *title* children below all *track* elements. Because our node labels encode this information and the node streams are ordered, we can apply the merge-based StackTree algorithm. In our example, the first *title* element does not find a join partner.

We extended the original StackTree operator by some features for the integration into our physical algebra. Our StackTree variant supports semi-joins, full joins, and outer joins. The join implementation returns the result in distinct document order (in case of a semi-join), or in *inner/outer* and *outer/inner* output ordering (in case of a full join or an outer join). The evaluation axis can be one of the following: *child*, *attribute*, *descendant*, *parent*, *ancestor* (and the *-or-self* variations of *descendant* and *ancestor*). The reverse variants of the algorithms are implemented by exploiting join commutativity.

The twig join operator is a complex merge-join operation that can be seen as an extension of the structural join. In contrast to structural joins, the algorithm can consume more than two node streams, in which it matches complex branching path patterns known as twigs. Our notion of a twig has been introduced as the twig specification in Section 5 and Figure 5. To the best of our knowledge, [Mat09] published the first algorithm that provides all features that can be expressed in our twig specification. Such an algorithm is desirable, because the higher its expressiveness, the more operations can be embedded into the twig algorithm, thus, resulting in a smaller number of operators in the final plan and fewer intermediate results. Furthermore, evaluating as many operations as possible can avoid intermediate-result materialization.

We picked a promising approach as the basis for our implementation, namely, the TwigOpt algorithm proposed by [FJSY05]. The algorithm operates on a set of node streams, one for each node in the twig. For example, in the physical representation of the XQGM instance shown in Figure 6, each twig input produces such a stream. The streams can be generated by a document scan, an index scan, or by other operators. They have to return the nodes in document order. The twig algorithm accesses a stream through a cursor. The cursor state can be modified using methods setToFirst(), getCurrent(), and forwardTo(). The first method initializes the cursor to the first node. The second method returns the node at the current location of the cursor, and the forwardTo() method advances the cursor. Based on the state of the cursors, the TwigOpt algorithm identifies the cursor that can skip the largest fraction of its input stream. This cursor is advanced as far as possible. After each move, the cursor positions are checked for a twig match. In case of a match, some output according to the twig specification is produced.

#### 7.3 Index-Based PPOs

XTC supports a variety of indexes. The document store itself is an index that allows to retrieve nodes by their labels. All elements with a certain name can be indexed by the so-called *element index*. Element-index scans can produce node streams for the STJ and HTJ algorithms. For value-based point and range predicates, XTC can index text nodes in the *content index*. Because path expression occur frequently in XQuery, *path indexes* can be created. A path index is specified by a linear (i. e., non-branching) path pattern, for example I(//cd/title). All nodes that fulfill this pattern are stored in the index. Combining path indexes with content indexes results in the so-called *content-and-structure* (CAS) index, which is also defined by a linear path pattern. For all these indexes, appropriate access operators exist in the physical algebra.



Figure 8: The path synopsis

In the following, we want to highlight a specialty of XTC, namely, the integration of pathindex and CAS-index scans with the holistic twig join operator. Both index types are built with the help of the *path synopsis* (PS). A PS is the structural summary of all paths in the document. In XTC, the PS is kept in main memory for fast access. Figure 8 shows the PS of our sample document from Figure 7. Every node or attribute in the PS is labeled with a unique integer called *path-class reference* (PCR). Given the PS, a PCR, and a node label, we can reconstruct the entire path to the root without accessing the document (the labels can be computed from the given node label and the element names can be retrieved from the path synopsis). For example, PCR 11 (see Figure 8) and node label 1.3.11.3.3 (see Figure 7) let us reconstruct the following ancestor nodes: *title* (1.3.11.3.3), *tracks* (1.3.11.3), *tracks* (1.3.11), *cd* (1.3), and *recordStore* (1).

Because our path-based indexes store PCRs together with node labels, scan operations on these index types return PCR-label pairs. By applying ancestor reconstruction to such an index scan, we can compute the node streams that are required as input to the holistic twig join operator without document access. This is accomplished with the help of an algorithm called *ancestor tuple builder* (ATB). Figure 9 gives a schematic overview over the interaction between the holistic twig join and the ancestor tuple builder: Let us assume that a linear path pattern of a twig specification is covered by a path index (the darker shaded nodes in Figure 9). Their cursors ( $C_1$  to  $C_3$ ) forward the HTJ's cursor requests to the ATB-input algorithm, which returns the necessary nodes based on an index-scan cursor ( $C_I$ ). Nodes for the ancestor can call open() to open the node stream and processTo(), which advances the computation to a given node label (similar to the other twig cursors).

#### 7.4 Heuristics for Plan Generation

As stated in Section 6, the plan generator shall enable the generation of all plans in the search space (see [Mat09] for the details), and shall be able to assemble a good plan. To reach the second goal, we conducted a number of experiments based on the XMark benchmark [SWK<sup>+</sup>02]. Figure 10 highlights some results. All 20 XMark queries were evaluated either fully implemented by navigational operators (see Section 7.1), by join-based operators (see Section 7.2), or based on a set of existing indexes (see Section 7.3). As



Figure 9: Integrating index scans with the twig join algorithm

you can observe, in this benchmark, join-based query evaluation is almost always a better strategy than navigation. Especially on queries with long paths, index-based evaluation is advantageous. From these and from other experimental results reported in numerous papers about indexing, structural joins, and holistic twig joins, we drew a set of rules to parameterize the plan generator. We chose the following simple heuristics: The plan generator 1. always unnests the query to enable join-based query processing, 2. favors join-based processing over navigational processing, if an element index exists, 3. favors twig joins over structural joins, and 4. gives indexes the following precedence (from high to low priority): CAS index, content index, path index, element index, and document index.

## 8 Conclusion

XML data processing has been an actively researched topic in the last decade, which lead to XML support in all major commercial database systems. This paper summarized the author's research on XML query processing, which was conducted during the authors engagement in the XTC project at the University of Kaiserslautern. The hierarchical data model and the semantically rich XQuery language required new approaches to data storage, indexing, query processing algorithms, and query rewriting. To bring these new ap-



Figure 10: A comparison between physical operators on the XMark query set

proaches together, XTC leans on the classical relational query-processing pipeline and extends the well-known relational Query Graph Model for query representation. For a prototypical system, XTC has a quite extensive physical algebra including a rich set of different index types and navigational, join-based, and index-based query processing algorithms. This makes XTC ideal as a test bed for future research.

## Acknowledgements

This work would not have been possible without the help of the XTC team: I like to thank Michael Haustein (the founder), Karsten Schmidt, Sebastain Bächle, Yi Ou, Leonardo Ribeiro, Aguiar Moraes Filho, Andreas Weiner, Stefan Hühner, and Caeser Ralf Franz Hoppen. I thank Theo Härder for his guidance and inspiration.

## References

[AkPJ <sup>+</sup> 02]	Shurug Al-khalifa, Jignesh M. Patel, H. V. Jagadish, Divesh Srivastava, Nick Koudas, and Yuqing Wu. Structural joins: A Primitive for Efficient XML Query Pattern Matching. In <i>Proc. ICDE</i> , pages 141–152, 2002.
[BCF <sup>+</sup> 04]	Scott Boag, Donald Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Ro- bie, and Jérome Siméon. XQuery 1.0: An XML Query Language. W3C Recommen- dation, 2004. http://www.w3.org/TR/xquery/.
[BKS02]	Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In <i>Proc. SIGMOD</i> , pages 310–321, 2002.
[CFS07]	B. Choi, M. Fernández, and J. Siméon. The XQuery Formal Semantics: A Foundation for Implementation and Optimization. W3C Recommendation, January 2007. http://www.w3.org/TR/xquery-semantics/.
[CLL05]	Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In <i>Proc. SIGMOD</i> , pages 455–466, 2005.
[CLT <sup>+</sup> 06]	Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig2Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents. In <i>Proc. VLDB</i> , pages 283–294, 2006.
[CVZ <sup>+</sup> 02]	Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In <i>Proc. VLDB</i> , pages 263–274, 2002.
[FHM <sup>+</sup> 05]	M. Fernández, J. Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Optimizing Sorting and Duplicate Elimination. In <i>Proc. DEXA</i> , pages 554–563, 2005.
[FJSY05]	Marcus Fontoura, Vanja Josifovski, Eugene J. Shekita, and Beverly Yang. Optimizing Cursor Movement in Holistic Twig Joins. In <i>Proc. CIKM</i> , pages 784–791, 2005.

- [FMM<sup>+</sup>04] M. Fernández, A. Malhotra, J. March, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Recommendation, 2004. http://www.w3.org/ TR/xpath-datamodel/.
- [GD87] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. In *Proc. SIGMOD*, pages 160–172, 1987.
- [GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. ICDE*, pages 209–218, 1993.
- [Gra94] Goetz Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.
- [HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible Query Processing in Starburst. In *Proc. SIGMOD*, pages 377–388, 1989.
- [HH07] Michael P. Haustein and Theo Härder. An Efficient Infrastructure for Native transactional XML Processing. *Data and Knowledge Engineering*, 61(3):500–523, 2007.
- [KD99] Navin Kabra and David J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal*, 8(1):55–78, 1999.
- [Mat07] Christian Mathis. Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility. *Computer Science Research and Development*, 21(3):147–164, 2007.
- [Mat09] Christian Mathis. Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems. Doctoral Thesis, University of Kaiserslautern, July 2009.
- [MH06] Christian Mathis and Theo Härder. Hash-Based Structural Join Algorithms. In *Proc. EDBT Workshops*, pages 136–149, 2006.
- [MHH06] Christian Mathis, Theo Härder, and Michael Haustein. Locking-Aware Structural Join Operators for XML Query Processing. In *Proc. SIGMOD*, pages 467–478, 2006.
- [Mit95] Berhnhard Mitschang. Anfrageverarbeitung in Datenbanksystemen (Entwurfs- und Implementierungskonzepte). Vieweg, 1995. German only.
- [MKL88] Guy M. Lohman Mavis K. Lee, Johann Christoph Freytag. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proc. VLDB*, pages 18–229, 1988.
- [MWHH08] Christian Mathis, Andreas Weiner, Theo Härder, and Caesar Ralf Franz Hoppen. XTCcmp: XQuery Compilation on XTC. In *Proc. VLDB*, pages 1400–1403, 2008.
- [PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. *SIGMOD Record*, 21(2):39–48, 1992.
- [SWK<sup>+</sup>02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, pages 974–985, 2002.

Industrieprogramm