

Normalization and Translation of XQuery

Norman May

SAP Research, CEC Karlsruhe

Vincenz-Prießnitz-Str. 1

76131 Karlsruhe

Germany

Guido Moerkotte

Database Research Group

University of Mannheim

68131 Mannheim,

Germany

ABSTRACT

Early approaches to XQuery processing proposed proprietary techniques to optimize and evaluate XQuery statements. In this chapter, we argue for an algebraic optimization and evaluation technique for XQuery as it allows us to benefit from experience gained with relational databases. An algebraic XQuery processing method requires a translation into an algebra representation. While many publications already exist on algebraic optimizations and evaluation techniques for XQuery, an assessment of translation techniques is required. Consequently, we give a comprehensive survey for translating XQuery into various query representations. We relate these approaches to the way normalization and translation is implemented in Natix and discuss these two steps in detail. In our experience, our translation method is a good basis for further optimizations and query evaluation.

Keywords: Database, Database Languages, Database Management Systems, Data Processing Software, Query Languages, XML

INTRODUCTION

As XQuery is used in an increasing number of applications, the execution time of these queries becomes more important for the acceptance of this query language. Especially for queries where potentially large amounts of XML are processed, strategies to reduce the query processing time need to be applied. The first XQuery processors often implemented a number of heuristics for this purpose. As XQuery becomes more popular, specific storage and index structures as well as specialized execution strategies were implemented.

Recently algebraic optimization techniques, as they are standard in relational databases, were used to build XQuery processors where optimizations are stated as algebraic equivalences, e.g. (Naughton et al., 2001; Jagadish et al., 2002; Fiebig et al., 2002; May et al., 2006; Ozcan et al., 2005; Nicola & van der Linden, 2005; Florescu et al., 2004; Boncz et al., 2006; Liu et al., 2005; Pal et al., 2005). This development is motivated by the ability (1) to apply optimizations known from relational databases and adopt them for XML processing, (2) to prove the correctness of query optimizations based on a formally defined query representation. Relational query optimizers today approach query optimization in a sequence of six steps:

1. **Scan and Parse the query statement** to analyze the lexical structure of the query.

2. **Normalization of the query, translation into an internal representation, type checking, and semantic analysis:** This phase checks the semantic correctness of the query. At the same time, additional information is attached to the parse tree, e.g. type information, references to schema information, or references to available statistics. As the parse tree is not the most convenient representation of the query to apply optimizations, it is translated into an internal query representation – usually an algebraic or calculus representation. The translation step may require some normalization to be applied before. The XQuery specification gives some rules for typing XQuery expressions (Draper et al., 2007), but more precise results can sometimes be obtained. In this chapter, we will focus on this optimization phase, in particular the normalization and translation step.
3. **First heuristic optimization phase:** In this phase the query optimizer applies heuristic optimizations. Some of these optimizations are hard to implement in a cost-based optimizer, e.g. predicate-move-around. Other optimizations applied in this phase prepare the query so that the search space of the cost-based optimizer is increased, e.g. query unnesting (May et al., 2004), (May et al., 2006) or view merging, and thus, often drastically improve the overall quality of the resulting query execution plan.
4. **Cost-based optimization phase:** Based on cardinality estimates and a cost-model for possible implementations of a query, the cost-based optimizer generates equivalent plan alternatives, called query execution plan (QEP). These alternatives differ in the order of the involved operators or their implementations. Among all alternatives examined in this phase, the most efficient one is chosen. Several metrics are used as efficiency criteria, e.g. resource consumption or expected query execution time.
5. **Second heuristic optimization phase:** This phase applies heuristic optimizations that are not considered in the previous phase, e.g. merging adjacent operators.
6. **Code generation:** This phase transforms the QEP into an executable form.

In this chapter, we first discuss how an XQuery query can be translated into an internal representation that is close to the well-known QGM-model used in IBM Starburst/DB2 (Haas et al., 1989). As several other query optimizers use a similar query representation, it is desirable to reuse their infrastructure to implement a query optimizer for XQuery. As (Grust et al., 2004) have pointed out, some care is required when a relational database is used to evaluate XQuery. Others discussed algebraic optimizations for XQuery based on native XML database management systems, e.g. (May et al., 2006; May, 2007; Re et al., 2006).

After surveying existing approaches to algebraic XQuery optimization and translation approaches, we introduce an algebra to represent XQuery statements that serve as input to algebraic optimizations. This algebra is defined over sequences of tuples as it is required to preserve the semantics of XQuery. Hence, it is not possible to directly apply algebraic optimizations known for SQL and relational databases to XQuery.

Then, we define the fragment of XQuery that is supported by our translation approach. This fragment covers a large fraction of the XQuery 1.0 language. After that, we introduce a number of normalization rules that prepare the XQuery statement for the translation step. We use rewrite rules that formally state the normalization rewrite. Such a formal notation is desirable because we can prove of correctness of these rewrites. In the core part of this chapter, we introduce the translation function that maps a normalized XQuery expression into the algebra as it is used in the Natix native XML database system. This algebraic expression can later be rewritten by cost-based algebraic optimizations.

We conclude this chapter with a summary of the main contributions of this chapter. Furthermore, we outline how the resulting algebraic expressions can be mapped to a calculus representation, and we discuss some open issues related to the translation and normalization of XQuery queries.

APPROACHES TO XQUERY NORMALIZATION AND TRANSLATION

In this section, we survey approaches to represent XQuery queries for the purposes of query optimization. There are a number of specifically tailored representations, e.g. (Kay, 2008). In contrast to these approaches formalisms based on algebras or calculus representations are advantageous for two reasons. First, these optimizations can be better expressed and implemented based on this internal query representation. Second, as a prerequisite for leveraging these optimization techniques, the XQuery statement needs to be translated into the desired internal representation. Early approaches to XML query processing focused on XPath which is a proper sublanguage of XQuery. Hence, we discuss approaches to XPath translation before we extend our scope to the XQuery language.

Query Representations for XQuery Optimization

XQuery optimizations typically transform a representation of the original XQuery statement into an equivalent form that supposedly is more efficient to evaluate. In this section we structure this variety of representations that were proposed as a logical algebra or calculus for XQuery or XPath.

Extensions to Relational Algebras. Extensions to relational algebras leverage the power and experiences of optimizing OQL and SQL by extending the logical algebras used for these languages. Relational algebras are based on sets (Maier, 1983). For SQL, this algebra was extended to support bag semantics (Dayal et al., 1982; Albert, 1991) or OQL (Cluet & Moerkotte, 1993; Steenhagen et al., 1994). Because the XQuery data model is based on sequences of items, algebras for XQuery need to handle both duplicates and order. Algebras proposed for order- and duplicate-aware data models (Slivinskas et al., 2002; Lerner & Shasha, 2003) and specifically for XQuery include (Beerli & Tzaban, 1999; Frasinca et al., 2002; May et al., 2004; Grust & Teubner, 2004).

Tree Algebras. An alternative approach represents queries as pattern trees (Jagadish et al., 2002). XPath expressions are translated into a pattern tree. Computing the result of an XPath expression corresponds to finding all embeddings of the tree pattern in the XML tree instance. The use of tree algebras is motivated by the fact that one can formally reason about trees (Suciu, 2001). A particularly interesting result is that query containment is *coNP* complete once either two features //, [], * are combined with the child axis (Miklau & Suciu, 2002). For more restricted cases query containment is in *P*. As query containment is an important test for applicability of views to answer a query, these results affect our translation procedure discussed later in this chapter. On the other hand, tree algebras seem to lack the expressiveness needed to represent any query formulated in XQuery. Most tree algebras are restricted to a subset of axis steps and have difficulty in expressing advanced XQuery constructs such as node construction or type-based constructs (Hosoya & Pierce, 2000).

Calculus Representations. The third camp translates the XQuery query into a representation close to the query language level. This includes representations as query graph (Haas et al., 1989), (Jarke & Koch, 1984) or in comprehension calculus (Fegaras & Maier, 1995). Both the query graph model (Shanmugasundaram et al., 2001), (Ozcan et al., 2005) and the comprehension calculus (Fegaras et al., 2002) required extensions to support XQuery. Calculus representations do not define a strict execution strategy for an XQuery statement. As a consequence, optimizations such as unnesting rewrites are easier to implement because pattern matching needs to consider fewer cases. On the other hand, another translation step into a query execution plan (QEP) is needed.

Other Approaches. In the literature on XQuery optimization the distinction between logical and physical algebra is often blurred (Brantner et al., 2005; Re et al., 2006). The reason is that heuristics are used to directly derive an efficient QEP from the query. We prefer to clearly separate logical and physical algebra, as it is done e.g. in (Jagadish et al., 2002; Florescu et al., 2004; Liu et al., 2005; Boncz et al., 2006) where the logical algebra is not concerned with specific implementations of operators. These implementations are defined by the physical algebra. Overall, this separation shall lead to a cleaner architecture of the XQuery processor.

Normalization and Translation of XQuery

To enable a query statement to be optimized using algebraic optimizations, it needs to be translated into an algebra expression. Most systems prepare this translation with a normalization step. In this section, we give a historic account on these steps and relate them to the normalization and translation of XQuery.

Classic Techniques for Normalization and Translation of SQL. There are two main approaches to optimize a query. In the first one, the query is transformed into an internal representation that can be interpreted by the query evaluation system (Astrahan & Chamberlin, 1975; Wong & Youssefi, 1976). In this approach, called *interpretation*, there are only limited possibilities for optimizations.

Thus, the *translation* of a query into an internal representation is now the dominant technique in query processing. Relational algebra and relational calculus equivalences became prime targets for the translation of query languages because one can formally prove the equivalence of two expressions. It is then the task of query optimization to find equivalent expressions that can be evaluated more efficiently.

(Ceri & Gottlob, 1985) translate a SQL query into an algebraic expression in two steps: The first step transforms the SQL syntax into a restricted one establishing a normalized syntax. This simplifies the translation, the second step, in which the restricted SQL syntax is translated into the algebra. The authors argue that the resulting algebraic expression can be optimized and, thus, efficiency of the resulting algebraic expression is not an issue. Moreover, it is shown that their translation establishes a normal form because syntactically different queries are translated into the same algebraic expression.

Calculus representations are another representation into which SQL is translated (Negri et al., 1991; von Bültzingsloewen, 1987; Fegaras & Maier, 1995). (Fegaras & Maier, 1995) and (von Bültzingsloewen, 1987) use a normal form established during their translation as the basis for further optimizations. Optimizing nested queries either containing quantifiers or aggregate functions are the prime subjects of research here (Jarke & Koch, 1984; Bry, 1989; von Bültzingsloewen, 1987; Nakano, 1990; Fegaras & Maier 1995).

In Starburst and DB2, a query is translated into the Query Graph Model (QGM) (Haas et al., 1989). QGM is a query representation that is proprietary to the IBM database products that is based on the idea of the calculus representations above. For SQL and, as we will see later, also for XQuery, a mapping of query constructs to the QGM is defined (Ozcan et al., 2005), (Nicola & van der Linden, 2005). Unfortunately, only informal descriptions of this mapping are publicly available. Heuristic optimizations, such as the decorrelation of nested queries and the merging of QGM blocks, are performed on this representation.

Translation of XPath 1.0. Path expressions represent an important fragment of XQuery. Many features of the XPath 1.0 standard have become part of the XQuery specification. Thus, translation, optimization, and evaluation techniques proposed for XPath 1.0 should carry over to path expressions in XQuery. (Gottlob et al., 2002) observed that XPath expressions have an exponential worst-case run time when subexpressions are evaluated repeatedly. They propose to use memoization as execution strategy to avoid this redundant work. Along the same line, (Helmer et al., 2002) translate XPath location steps without positional predicates such that creating duplicates is avoided. These ideas were extended in (Hidders & Michiels, 2003), where redundant sort operations are removed when the result of the path expression will still be in document order. (Brantner et al., 2005) were the first to present a complete translation procedure for XPath 1.0 into algebraic expressions. A comprehensive translation of XPath into SQL statements is proposed in (Grust, 2002).

Normalization and Translation of XQuery. The idea of (Ceri & Gottlob, 1985) to normalize the full query syntax into a core language is also proposed in the XQuery specification (Draper et al., 2007). The formal semantics of XQuery is defined in terms of this core language. Thus, an interpretative view of evaluating XQuery is taken in the formal specification. While some implementations of XQuery implement these semantics literally, it was soon clear that efficient XQuery processing demands a query representation that is easy to optimize. The first proposal to normalize XQuery was proposed by (Manolescu et al., 2001). Their normalization rules prepare XQuery statements for the translation into SQL statements. Thus, all normalization rules work on the level of XQuery statements, remove nested FLOWR expressions, and establish some normal form that is not formally characterized. Extending previous work (Fegaras & Maier 1995), (Fegaras et al., 2002) translate XQuery statements into monoid

comprehensions. Rewrites establish a unique normal form to prepare subsequent optimizations. Monoid comprehensions allow for an elegant integration of different bulk types. Expressions in this calculus can be checked to preserve order or duplicates. Unfortunately, the cited work does not seem to exploit this fact.

The Timber system (Jagadish et al., 2002) follows a different approach. Queries are translated into pattern trees defined in the logical tree algebra TAX. Optimizations are defined as rewrites on this tree algebra. All pattern trees in TAX can be mapped to algebraic operators in the physical algebra.

These early proposals do not fully support the XQuery specification. Some of these efforts included a translation of XQuery into SQL (Krishnamurthy et al., 2003). However, the MonetDB/Pathfinder project covers a large subset of XQuery. In this system, XML documents are represented in a pre-/post order encoding that maps a unique identifier to each node in the document (Grust, 2002). This allows to construct SQL queries that retrieve all nodes that satisfy a path expression. Later, this translation was extended to larger fragments of XQuery (Grust et al., 2004; Grust & Teubner, 2004).

In the following, as the standardization process of XQuery converged, the focus shifted to a more complete coverage of the XQuery specification. The XQuery engine of the BEA streaming XQuery engine (Florescu et al., 2004) translates XQuery expressions into an internal expression representation. While this representation shares the ideas of the relational algebra, it is specifically designed to represent XQuery expressions. Both normalization and optimization are carried out as rewrites on this query representation. A similar approach is taken by Galax (Re et al., 2006). This system implements the normalization of the XQuery specification literally. Afterwards, the resulting XQuery core expressions are translated into an extended algebra and optimized using algebraic rewrites.

Commercial relational database products also support XQuery to a varying extent. Microsoft SQL Server (Pal et al., 2005) and Oracle XML DB (Liu et al., 2005) support fragments of the XQuery specification. Queries are translated into algebraic expressions and, if possible, rewriting techniques of the relational optimizer are used for optimizations. To support XQuery in IBM DB2 (Ozcan et al., 2005; Nicola & van der Linden, 2005) the QGM query representation of DB2 was extended. The query representation used in Natix also maps the translated algebraic expression to an internal representation that is similar to the query graph model (Fiebig et al., 2002; May, 2007).

Figure 1 summarizes the approaches surveyed in this section.

Approach	XPath 1.0	XQuery 1.0	Query Representation
XML TaskForce (Gottlob et al., 2002)	(almost) Full	No	Context value table
XPC (Helmer et al., 2002; Brantner et al., 2005)	Full	No	Physical algebra
Natix (Fiebig et al., 2002; May, 2007)	Full	Partial	Calculus and physical algebra
(Fegaras et al., 2002)		Partial	Monoid comprehension calculus
Timber (Jagadish et al., 2002)	Partial	Partial	Tree algebra
BEA XQuery Processor (Florescu et al., 2004)	Full	Full	Combined logical and physical algebra
Pathfinder and MonetDB/XQuery (Grust et al., 2004; Grust & Teubner, 2004)	Full	Full	Logical algebra and either SQL (Pathfinder) or physical algebra (MonetDB)
MS SQL Server (Pal et al., 2005)	Partial	Partial	Logical and physical algebra
Oracle XML DB (Liu et al., 2005)	Partial	Partial	Logical and physical algebra
IBM DB2 (Ozcan et al., 2005; Nicola & van der Linden, 2005)	Full	Full	Calculus and physical algebra

Figure 1. Comparison of XML translation approaches.

NAL – THE NATIX A ALGEBRA FOR XQUERY OPTIMIZATION

In the remainder of this chapter, we refer to the logical algebra as defined below when we talk about an algebra or algebraic operator. The logical algebra, NAL, we introduce in this section defines the logical operations executed in a query. This set of operators is sufficient as a target for the translation into an internal query representation as a basis for query optimization.

In contrast to a physical algebra, a logical algebra does not imply any specific implementation. This allows us to investigate the equivalence of two algebraic expressions much more succinctly. Furthermore, query optimization can at least conceptually be split up into a phase with logical optimizations and physical optimization (Chaudhuri, 1998).

The definition of the algebraic operators in NAL requires some notation for our algebra. As noted above, our algebra extends the relational algebra by further operators needed to represent XQuery queries. Furthermore, it is defined over sequences of tuples as detailed below.

We denote sequences by $\langle \bullet \rangle$, the empty sequence by ϵ , and sequence concatenation by \oplus . Note that sequence concatenation is associative but not commutative. For a sequence e we use $\alpha(e)$ to select its first element and $\tau(e)$ to retrieve its tail. We equate sequences containing a single item and the item contained. This implicit conversion is demanded by the XQuery specification.

Tuples are constructed by using brackets $[\cdot]$ and concatenated by \circ . The set of attributes of a tuple t is denoted by $A(t)$. The projection of a tuple t on a set of attributes A is denoted by t_A . To access a single attribute B in a tuple, $B \in A(t)$, we use $t.B$. For all tuples t_1 and t_2 contained in a sequence of tuples, we demand $A(t_1) = A(t_2)$.

Given that, we can define the set of attributes $A(s)$ provided by a sequence s as the set of attributes of the contained tuples. Let e be an expression whose result is a tuple or a sequence of tuples. Then the set of attributes provided in the result of e is denoted by $A(e)$.

Binding an attribute a of some tuple to a value v is denoted by $[a:v]$. We call an attribute a in an expression e free if it occurs in e and is not bound to a value by e . That is, a value for a has to be provided by some other expression, e.g. an outer query block. We denote the set of free attributes of an expression e by $F(e)$. Note that attributes behave the same way as variables: they are bound to a value by some expression and referenced by another one. From now on, we will use the terms variable and attribute interchangeably.

For an expression e_1 possibly containing free variables, and a tuple e_2 , we denote by $e_1(e_2)$ the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by e_2 . Of course this requires $F(e_1) \subseteq A(e_2)$. For a set of attributes, we define the tuple constructor \perp_A such that it returns a tuple with attributes in A initialized to *NULL*. Thanks to the *NULL*-value, we can distinguish empty results from unknown values which is not possible in XQuery yet.

Using these notations, we introduce two elementary operations to construct sequences. The first is $\#$, which returns a singleton sequence consisting of the empty tuple, i.e. a tuple with no attributes. It is used in order to avoid special cases during the translation of XQuery. The second operation, denoted by $e[a]$, constructs a sequence of tuples with attribute a from a sequence of non-tuple values e . For each value c in e , a tuple is constructed containing a single attribute a whose value is c . More formally, we define $e[a] := \epsilon$ if e is empty, and $e[a] := [a : \alpha(e)] \oplus \tau(e)[a]$ else. We use this operation to map sequences of items in the XQuery data model into sequences of tuples in our data model.

We refer to an n-ary function, say f , with $f(e_1, \dots, e_n)$. Sometimes, we will omit the formal parameters in expressions. Then the actual parameters of f must be bound by the enclosing expression. We denote the identity function by id and concatenation of functions or operators by \circ .

For result construction we define a function with signature $C(\text{type}, \text{name}, \text{content})$. It constructs a node of the requested node type, with given tag name, and content. We use the arguments *elem*, *attr*, etc. to identify the node type. To support computed constructors, the name and content may reference previously

bound variables. Not every argument is meaningful for every node type. But for the sake of simplicity, we ignore this fact.

Based on the notation, we are now able to define the algebraic operators in NAL; Figure 2 summarizes their definitions. For space reasons we cannot discuss these operators in detail here, and thus we refer to (May, 2007; May et al., 2006) for a detailed discussion of this algebra and possible implementations of the involved operators.

<p>Scan Singleton</p> $\# := \langle [] \rangle$
<p>Selection</p> $\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else} \end{cases}$
<p>Tid</p> $tid_a(e) := tid_a(e,1) \text{ where}$ $tid_a(e,n) := \alpha(e) \circ [a:n] \oplus tid_a(\tau(e),n+1)$
<p>Projection</p> $\Pi_A(e) := \alpha(e) _A \oplus \Pi_A(\tau(e))$
<p>Tid-Duplicate Elimination</p> $\Pi_A^{tid_b}(e) := \begin{cases} \alpha(e) _A \oplus \Pi_A^{tid_b}(\tau(e)) & \text{if } \alpha(e).b \notin \Pi_b(\tau(e)) \\ \Pi_A^{tid_b}(\tau(e)) & \text{else} \end{cases}$
<p>Map</p> $X_{a.e_2}(e_1) := \alpha(e_1) \circ [a:e_2(\alpha(e_1))] \oplus X_{a.e_2}(\tau(e_1))$
<p>Product</p> $e_1 \bar{\times} e_2 := \begin{cases} \varepsilon & \text{if } e_2 = \varepsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \bar{\times} \tau(e_2)) & \text{else} \end{cases}$ <p style="text-align: center;">where e_1 is a singleton</p>
<p>Cross Product</p> $e_1 \times e_2 := (\alpha(e_1) \bar{\times} e_2) \oplus (\tau(e_1) \times e_2)$
<p>Join</p> $e_1 \times_p e_2 := \sigma_p(e_1 \times e_2)$
<p>D-Join</p> $e_1 < e_2 > := \alpha(e_1) \bar{\times} e_2(\alpha(e_1)) \oplus \tau(e_1) < e_2 >$
<p>Semijoin</p>

$e_1 \propto_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \propto_p e_2) & \text{if } \exists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \propto_p e_2 & \text{else} \end{cases}$
<p>Antijoin</p> $e_1 \triangleright_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \triangleright_p e_2) & \text{if } \exists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \triangleright_p e_2 & \text{else} \end{cases}$
<p>Left Outer Join</p> $e_1 \nabla_p^{g:e} e_2 := \begin{cases} (\alpha(e_1) \nabla_p e_2) \oplus (\tau(e_1) \nabla_p^{g:e} e_2) & \text{if } (\alpha(e_1) \nabla_p e_2) \neq \varepsilon \\ (\alpha(e_1) \circ \perp_{A(e_2) \setminus \{g\}} \circ [g:e]) \oplus (\tau(e_1) \nabla_p^{g:e} e_2) & \text{else} \end{cases}$
<p>Union</p> $e_1 \hat{\cup} e_2 := e_1 \oplus e_2$
<p>Intersection</p> $e_1 \hat{\cap} e_2 := e_1 \propto_{A(e_1)=A(e_2)} e_2$
<p>Difference</p> $e_1 \hat{\wedge} e_2 := e_1 \triangleright_{A(e_1)=A(e_2)} e_2$
<p>Unnest</p> $\mu_{A:g}(e) := (\alpha(e) \times (\Pi_{A:A(g)}(\alpha(e).g))) \oplus \mu_{A:g}(\tau(e))$
<p>Unnest Map</p> $Y_{A:e_2}(e_1) := \Pi_{\bar{a}}(\mu_{A:\hat{a}}(\chi_{\hat{a}:e_2}(e_1)))$
<p>Binary Grouping</p> $e_1 \Gamma_{g:A_1 \theta A_2; f} e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1))] \oplus (\tau(e_1) \Gamma_{g:A_1 \theta A_2; f} e_2) \quad \text{where}$ $G(x) := f(\sigma_{x _{A_1} \theta A_2}(e_2))$
<p>Unary Grouping</p> $\Gamma_{g;\theta A; f}(e) := \Pi_{A:A'}(\Pi_{A':A}^D(\Pi_A(e)) \Gamma_{g:A' \theta A; f} e)$

Figure 2. NAL algebra.

NORMALIZATION OF XQUERY

Prior to the translation of an XQuery statement – or rather transforming its parse tree into the logical algebra – a normalization step is employed to normalize the representation of the query statement. The XQuery specification explicitly defines a core language of XQuery that is usually much more verbose than the equivalent original statement. But it limits the number of cases to consider, e.g. for describing XQuery's formal semantics or its translation into an optimizer-internal representation. A similar approach is usually taken for SQL (Ceri & Gottlob, 1985).

The rewrites applied to a query during normalization, however, should obey to a number of requirements:

- **Soundness:** Each transformation must preserve the semantics of the given query.

- **Completeness.** Ideally, every query construct should be handled by the normalization and translation step. As XQuery is a query language with many features, we cannot treat every language construct yet. Instead, we concentrate on the XQuery fragment defined below.
- **Uniqueness:** The normalization and translation of equivalent queries should result in the same representation of the query, i.e. a normal form. The application order of rewrite rules should not matter, and it should be ensured that the normal form is reached by the normalization algorithm if the normal form exists.
- **Effectiveness** The normal form should be reached in a finite number of transformations, preferably in a few transformation steps. We will point out how we achieve this.
- **Optimizability.** The normal form should be a good starting point for optimizations applied later during query optimization.

In our presentation of the normalization step, we will first present the XQuery fragment that is currently supported by our approach. We then informally discuss properties of the normalized query. After that we introduce the rewrite rules used to normalize XQuery statements and analyze how they contribute to achieving the desired properties of a normalized query statement. We then illustrate our approach based on a number of examples and conclude this section by enumerating some restrictions of our method.

Supported XQuery Fragment

In this chapter, we focus on a subset of XQuery that is expressive enough to formulate complex queries, e.g. nested queries. However, the translation and optimization approach we cover here is general enough to support the missing features. Figure 3 presents the subset of the XQuery grammar we currently support. It is a variant of LixQuery grammar (Hidders et al., 2004).

In this grammar, we denote terminals with `terminal` and non-terminals with `nonterminal`. Some terminals contain complex regular expressions of tokens. We refer to (Hidders et al., 2004) for their definition and simply use angle brackets instead, i.e. `<complex token>`. For simplicity, we use a very restrictive set of functions which we all treat as special built-in functions. In particular, we ignore user-defined or recursive functions.

In the grammar, we only give the productions for computed constructors. Since our example queries use direct constructors, we need to normalize them into computed constructors as defined in (Draper et al., 2007). We will use this normalization step in this chapter without repeating the associated rewrites.

We have added `flwrExpr` to be able to express queries more succinctly and `quantExpr` because we want to express quantifiers explicitly. Furthermore, we distinguish between general comparison – having existential semantics – and value comparison. All these extensions to LixQuery are syntactic sugar, but are often used in practice. As we will see later, their treatment has several implications on normalization, translation, and optimization of XQuery.

Note that we have simplified the grammar. For example, our grammar does not explicitly enforce any precedence rules for binary operators as it is done in the XQuery specification (Boag et al., 2007). Nevertheless they are still left associative.

```

mainModule ::= expr <EOF>
Expr       ::= singleExpr | exprSeq
exprSeq    ::= singleExpr ( "," singleExpr )*
singleExpr ::= flwrExpr | quantExpr | andExpr
builtIn    ::= ( "doc(" singleExpr ")"
              | "name(" singleExpr ")"
              | "string(" singleExpr ")"
              | "integer(" singleExpr ")"
              | "contains(" singleExpr "," singleExpr ")"
              | "true()" | "false()"

```

```

| "not(" singleExpr ")"
| "count(" singleExpr ")"
| "distinct-values(" singleExpr ")"
flwrExpr ::= (forExpr | letExpr)+ whereClause? "return" singleExpr
rangeExpr ::= var "in" singleExpr
bindExpr ::= var ":@" singleExpr
forExpr ::= "for" rangeExpr ("," rangeExpr)*
letExpr ::= "let" bindExpr ("," bindExpr)*
whereClause ::= "where" singleExpr
quantExpr ::= ("some" | "every") rangeExpr ("," rangeExpr)* "satisfies" ExprSingle
andExpr ::= compExpr ( ("or" | "and") compExpr )?
compExpr ::= addExpr ( (genComp | valComp | nodeComp) addExpr )?
genComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="
valComp ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
nodeComp ::= "<<" | ">>" | "is"
addExpr ::= multExpr ( ("+" | "-") multExpr )*
multExpr ::= union ( ("*" | "div" | "idiv" | "mod") union )*
union ::= path ( ("|" | "union" | "intersect" | "except") union )*
path ::= filter ( ("/" | "//") path )*
filter ::= step ( "[" singleExpr "]" )*
step ::= "." | ".." | QName | "@" QName | "*" | "@*" | "text()" | primaryExpr
primaryExpr ::= builtIn | QName | constr | var | literal | empSeq | "(" expr ")"
literal ::= string | integer
string ::= <String>
integer ::= ( (<Digits> | "+" <Digits> ) | ("-" <Digits> ) )
var ::= "$" QName
empSeq ::= "("
constr ::= "element" "{" singleExpr }" "{" expr }"
| "attribute" "{" singleExpr }" "{" expr }"
| "text" "{" singleExpr }"
| "document" "{" singleExpr }"
QName ::= <NCName> | (<NCName> ":" <NCName>)

```

Figure 3. Supported XQuery fragment.

A Notation for Normalization Rules

Conceptually, the normalization and translation rules we present here match patterns of the textual XQuery representation and transform them, given the bindings of the matched pattern. In this section, we will denote pattern matching with regular expressions on the grammar presented above. We refer to terminal symbols with `terminal` and to non-terminals with `nonterminal`. During normalization, some rules introduce new variable names using the expression `<$v = newVar(>`. Thereby, we create a new variable name to which we can refer by `$v`. We will also use `fun` to refer to arbitrary functions including `builtIn`, `andExpr`, `compExpr`, `addExpr`, `multExpr`, and `constr`. In the case of constructors, these arguments refer to the computed node name and the computed content.

As an important preparation step to the translation, we normalize XQuery expressions on the query level. More precisely, all normalization steps work on the abstract syntax tree created by the XQuery parser. Our normalization rewrites transform the XQuery statement into a normal form. The normalized query is easier to translate into our algebra because we have to consider fewer query patterns. Moreover, normalization facilitates common subexpression elimination because we introduce new variables that are

bound to complex expressions. In this query representation, it is much easier to detect common subexpressions.

There are many relationships between path expressions embedded into XQuery expressions and equivalent expressions in XQuery (Draper et al., 2007), (Michiels et al., 2006). For example, the transformation of XQuery into the XQuery core breaks location steps into nested FLWOR expressions (Draper et al., 2007). We use several of these techniques and, hence, reuse normalization rules presented there. But we will tailor normalization for our needs. In particular, we will break XPath expressions apart only when a location step contains a filter expression. The reverse step, detecting tree patterns, has been discussed in (Michiels et al., 2006). Our motivation for doing so is that especially for simple path expressions many optimizations are known, e.g. (Amer-Yahia et al., 2001; Helmer et al., 2002; Hidders & Michiels, 2003; Balmin et al., 2004). Several of these optimizations are only tractable or applicable for simple path expressions.

FLWR Expressions

Objectives. The objective of normalizing FLWR expressions consists in obtaining a uniform representation for different formulations of the FLWR expression. As a result, the subsequent steps of query compilation are simplified, most importantly the translation step and several optimizations. For example, during normalization we reduce the number of query patterns which have to be handled during query translation. Our normalization rewrites separate the query into three parts:

- **The binding part** consists of **for** and the **let** clauses. It gathers all queried data, computes intermediate results, and binds them to variables.
- **The modifying part** alters the tuple stream, either by changing the order of items as specified in the **order by** clause or by filtering out items in the **where** clause.
- **The result construction part** consists of the **return** clause, which solely refers to bound variables.

for rangeExpr₁ (, rangeExpr)+ → for rangeExpr₁ (N-1)

let bindExpr₁ (, bindExpr)+ → for rangeExpr (, rangeExpr)*
let bindExpr₁ (N-2)

(some|every) rangeExpr₁ (, rangeExpr)+ → let bindExpr (, bindExpr)*
satisfies exprSingle (N-3)

(some|every) rangeExpr₁ (, rangeExpr)+ → (some|every) rangeExpr₁
satisfies (N-3)

(forExpr|letExpr)+ → (forExpr|letExpr)+ (N-4)

where singleExpr₁ → let < \$p = newVar() > := singleExpr₁
return singleExpr₂ where \$p

(forExpr|letExpr)+ → (forExpr|letExpr)+ (N-4)

where singleExpr₁ → (some|every) rangeExpr₁ (, rangeExpr)*
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (some|every) rangeExpr₁ (, rangeExpr)*
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

where singleExpr₁ → (forExpr|letExpr)+
return singleExpr₂ let < \$v = newVar() > := singleExpr

```

let < $v_2 = newV ar() > := data($v_1)
satisfies
some < $v_3 = newV ar() > in singleExpr2
let < $v_4 = newV ar() > := data($v_3)
satisfies $v_2 = valComp $v_4

```

Figure 4. Normalization of FLWOR expressions.

Normalization Rules. In Figure 4, the rewriting rules for normalizing FLWR expressions are summarized. We now discuss the idea behind each normalization rule.

N-1 and N-2: We split **for** or **let** clauses that bind multiple variables into individual clauses. Note that, in contrast to the grammar productions for the **forExpr** and **letExpr**, the occurrence indicator in both rules is + instead of *. This is necessary for the correctness of the rewrite because it makes sure that the list of for or let clauses contains at least two clauses. After the exhaustive application of this rewrite, each **forExpr** or **letExpr** binds at most one variable.

N-3: We split quantified expressions that bind multiple variables into individual quantified expressions. Note that, in contrast to the grammar production for the **quantExpr**, the occurrence indicator of the **RangeExpr** in this rule is + instead of *. As for the previous rewrites it is necessary for the correctness of the rewrite. After the exhaustive application of this rewrite, each **quantExpr** contains at most one **RangeExpr**.

N-4: When the **where** clause of a FLWR expression contains a complex expression, we introduce a new **letExpr** and bind the computation of this complex expression to a new variable $\$p$. For this rewrite, we consider $\text{singleExpr}_1 \in \{\text{flwrExpr}, \text{builtIn}, (\text{Expr})\}$ as complex expressions but leave comparisons and quantified expressions as they are. We replace the old complex expression by a reference to the new variable $\$p$. After the exhaustive application of this rewrite the **where** clause contains only references to variables, quantified expressions, or comparison operators.

N-5: We move every complex expression in the range predicate of a quantified expression into a new **letExpr**. These **letExpr** are a convenient extension to simplify detection of common subexpressions and during translation. For this rewrite, we consider $\text{singleExpr} \in \{\text{flwrExpr}, \text{builtIn}, (\text{Expr})\}$ as complex expressions.

N-6: Similar to rule N-4, we move a complex expression from the **return** clause of a FLWR expression into a new **letExpr**. For this rewrite, we consider $\text{singleExpr} \in \{\text{flwrExpr}, \text{builtIn}, (\text{Expr}), \text{constr}, \text{exprSeq}\}$ as complex expressions. The exhaustive application of this rewrite leaves only a single variable reference in the **return** clause.

N-7: This rule replaces complex expressions inside a sequence of expressions by variables which are bound to the result of the replaced complex expression. We consider $\text{singleExpr} \in \{\text{flwrExpr}, \text{builtIn}, (\text{Expr}), \text{constr}, \text{exprSeq}\}$ as complex expressions.

N-8: This rule replaces complex expressions as function arguments by variable references which are bound to the result of the replaced complex expression. We consider $\text{singleExpr} \in \{\text{flwrExpr}, \text{builtIn}, (\text{Expr}), \text{constr}, \text{exprSeq}\}$ as complex expressions. We also treat built-in functions, constructors, arithmetic expressions, or comparisons as functions and refer to them by **fun**.

N-9: This rule turns general comparisons denoted by **genComp** into value comparisons denoted by **valComp**. The original general comparison is replaced by a quantified expression with the corresponding value comparison. The mapping of general comparisons into value comparisons is summarized in the table below (Draper et al., 2007). Note that we introduce the proper type conversion while typing both arguments and, hence, do not introduce them here.

genComp	valComp
=	eq
!=	ne
<	lt

<=	le
>	gt
>=	ge

Let us make sure that the rules in Figure 4 achieve our goals. First, notice that neither in the **where** clause nor in the **return** clause any of the rewrites introduces complex expressions. Second, notice that the rewrites introduce complex expressions only in new **let** clauses. They possibly create new **for** or **let** clauses containing complex expressions. The exhaustive application of these rewrites eventually results in the normal form discussed at the beginning of this section. Since we only move around complex expressions but do not create ones, we reach this normal form in as many steps as there are complex expressions.

XPath Expressions

When normalizing XPath expressions, our main goal consists in restructuring them such that they are easier to optimize. We attempt this by breaking branching path expressions into simple path expressions. This gives us two important opportunities for optimization:

(1) Predicates become visible. This enables us to detect join predicates, to move them into the **where** clause, and to unnest nested XPath expressions. (2) We assume that indices or materialized views are available rather for simple path expressions than for complex path expressions. Additionally, the problem of matching view definitions to path expressions in the user query becomes tractable when we extract simple path expressions from complex path expressions.

However, we have to be careful to preserve the semantics of path expressions.

1. In particular, we need to preserve document order, and we need to handle duplicates and position-based functions correctly. Currently, we do not rewrite the XPath expression when its evaluation depends on document order.
2. XPath expressions can contain predicates that correlate the selected node in the current path expression to nodes in another path expression.
3. XPath expressions may contain nested expressions that are interpreted as nested queries.

$$\text{path}_{\$c} (//) \text{ step } [\text{singleExpr}] \quad \rightarrow \quad \text{for } \langle \$v = \text{newVar}() \rangle \text{ in } \$c / \text{path } (//) \text{ step} \quad (\text{N-10})$$

$$\text{path}_{\$c} (//) \text{ step } ([\text{singleExpr}]^+ \text{ path}_2 \quad \rightarrow \quad \text{let } \langle \$v_1 = \text{newVar}() \rangle \text{ :=} \quad (\text{N-11})$$

$$\quad \quad \quad \$c / \text{path}_1 (//) \text{ step } ([\text{singleExpr}]^+)$$

$$\quad \quad \quad \text{for } \langle \$v_2 = \text{newVar}() \rangle \text{ in } \$v_1 / \text{path}_2$$

$$\quad \quad \quad \text{return } \$v_2$$

Figure 5. Normalization of XPath expressions.

Our normalization rewrites are summarized in Figure 5. They introduce new variables that store the intermediate results of the path expressions. We expect this to be beneficial for factorization of common subexpressions. When we add these variables into the current scope, we have to avoid name clashes.

N-10: This rewrite moves an XPath predicate into the **where** clause of a FLWR expression when the path expression is inside a **for** clause. Note that we ignore several intricate issues here: (1) the result of the XPath predicate is the effective boolean value of expression `exprSingle`. The computation done for the predicate might depend on actual types returned at runtime. (2) Positional predicates are another source of difficulty we ignore here. (3) Document order must be correct, e.g. when the last axis step before a positional predicate computes a reverse axis.

N-11: This rewrite allows us to break XPath expressions into pieces. Note that in both rewrites we use the variable `$C` to explicitly refer to the set of context nodes. We also expand abbreviated syntax in path expressions into the corresponding unabbreviated form (Draper et al., 2007), i.e.

1. We treat occurrences of `@NodeTest` as attribute axis, i.e. `attribute::NodeTest`.
2. We treat occurrences of `..` as parent axis, i.e. `parent::node()`.
3. We expand each occurrence of `//` in a relative location path to `/descendant-or-self::node()`. When the axis step afterwards contains a node test but no positional predicate, we can even replace `//NameTest` by `/descendant::NameTest`, which is more efficient to evaluate.
4. We rewrite absolute location paths so that they explicitly use function `fn::root`, i.e. `fn:root(self::node())` treat as `document-node()`.
5. When the axis name is omitted from an axis step, the default axis is `child` unless the axis step contains an attribute test or schema attribute test. Hence, we expand these path expressions by a `child` step including the node test.

Example Query

In this section, we apply our normalization rules to a concrete query to demonstrate their effectiveness in establishing our normal form. Starting with a query that uses a quantified expression we get existentially quantified expressions, and, thereby we make implicit computations explicit. Second, we want to rewrite the query such that it is more convenient to optimize. In particular, both types quantified expressions, but also implicit grouping is formulated with nested queries in XQuery 1.0. The resulting normalized query can later be unnested using techniques presented, e.g. in (May et al., 2004; May et al., 2006), realizing performance improvements in orders of magnitude.

```
for $t1 in doc("bib.xml")//book/ title
where $t1 = doc("reviews.xml")// entry / title
return $t1
```

Normalization is simple because we only need to turn the general comparison into a quantified expression using rewrite N-9. This rewrite introduces function data to apply atomization to the result of both range expressions.

```
for $t1 in doc("bib.xml" )//book/ title
where some $v1 in $t1
  let $v2 := data ($t1)
  satisfies
    some $v3 in doc("reviews.xml" )// entry / title
      let $v4 := data ($v3)
      satisfies $v2 eq $v4
return $t1
```

We now have established the desired form:

1. All data retrieval is done in the **for** or **let** clauses.
2. Implicit computations (e.g. the existential nature of general comparison) have become explicit.
3. The **return** clause only contains variable references.
4. Function calls, except function `fn:distinct-values`, do not contain complex expressions as arguments.

Restrictions

Several of our normalization rewrites are only valid under the assumption that certain information of the involved subexpressions will not be observed in the remainder of the query. This information includes node identity, local namespace declarations, and non-determinism of XQuery expressions. Besides our

normalizations, these issues rule out many other optimizations. But for many queries they do not cause any problems, and hence our normalizations will be valuable in many cases.

Node Construction and Node Identity. In some cases, common subexpressions cannot be factorized (Boag et al., 2007). For example:

```
( <a/>, <a/> )
```

is not the same as

```
let $x := <a/>
return ( $x, $x )
```

because the first expression constructs two distinct XML element nodes, whereas the second returns two identical XML nodes. This problem occurs in all rewrites that introduce new **let** clauses containing expressions with constructors. Since most operations do not exploit node identity, this problem is rarely an issue. In most cases, node construction is only done to construct the final result which is returned to the user.

Namespaces. When moving expressions, we need to be careful because element constructors might introduce new namespaces. When we move expressions out of these scopes, e.g. by introducing a new let expression, we violate these scoping rules as shown in the following example taken from (Florescu & Kossmann, 2004):

```
declare namespace ns="uri1"

for $x in fn:doc("uri ") / ns :a
where $x/ns:b eq 3
return
  <result xmlns:ns="uri2">
    { for $x in fn:doc("uri ") / ns :a
      return $x/ns:b }
  </ result>
```

When we apply our normalization rewrites as usual, the FLWOR expression bound to variable \$v2 is evaluated using namespace uri1 instead of uri2.

```
declare namespace ns="uri1"

for $x in fn:doc("uri ") / ns :a
where $x/ns:b eq 3
let $v2 := ( for $x in fn:doc("uri ") / ns:a
             return $x/ns:b )
let $v1 := <result xmlns:ns="uri2"> { $v2 } </ result>
return $v1
```

Thus, the rewrites might change the namespace declarations that are defined in the current evaluation context. In principle, one could establish the proper namespace declarations, but in this work we will ignore the problem of namespaces.

Ordering Mode. The result of the following expression is not deterministic. Depending on the order in which the values in the input sequence are applied to the predicate list, the result of this expression can either be an error or the value 3.

```
unordered{
  ("foo", "bar", 3)[ floor (.) < 5][1]
}
```

Hence, one must be careful when inferring unorderedness in subexpressions of queries, e.g.

```
some $i in ("foo", "bar", 3)[ floor (.) < 5][1]
satisfies true
```

Again, we will ignore these issues in our optimizations and assume deterministic results. We refer to (Grust et al., 2007) for a further discussion on this topic.

TRANSLATION OF XQUERY INTO THE NATIX ALGEBRA

The result of normalization, discussed in the previous section, will now turn out to be a convenient starting point for the translation of XQuery queries into our algebra. Let us therefore summarize the structure of normalized queries as they are produced during normalization.

First, path expressions are broken up into simple path expressions. Consequently, we only need to treat simple path expressions without nested path expressions or predicates in our translation function. Second, path expressions are only located in the **for** clause and the **let** clause. This assures uniform results after translation for different formulations of the same query. Third, nested query blocks are explicitly marked by FLWOR expressions or quantified expressions. Fourth, correlation between query blocks is explicitly handled in the **where** clause. Nested query blocks become subject to unnesting in later steps of the optimization process.

The binary T function for FLWOR expressions:

$$T(Q, A) := \begin{cases} T(REST, [tid_p(Y_{xT_T(e)}(A))]) & \text{if } Q = \text{for } \$x [\text{at } \$p] \text{ in } e \text{ REST or} \\ & \text{if } Q = \$x \text{ in } e \text{ REST} \\ T(REST, X_{xT_T(e)}(A)) & \text{if } Q = \text{let } \$x := e \text{ REST and } e \text{ is sequence-valued} \\ T(REST, X_{xT_I(e)}(A)) & \text{if } Q = \text{let } \$x := e \text{ REST and } e \text{ returns a single item} \\ T(REST, \sigma_{T_I(p)}(A)) & \text{if } Q = \text{where } p \text{ REST} \\ T(REST, \text{Sort}_{x_1 \dots x_n}(A)) & \text{if } Q = \text{order by } \$x_1 \dots \$x_n \text{ REST} \\ \Pi_e(A) & \text{if } Q = \text{return } \$e \\ A & \text{if } Q \text{ is empty string} \end{cases}$$

The unary functions T_T and T_I for other expressions:

$$T_T(Q) := \begin{cases} \text{translation of Brantner et al.} & \text{if } Q \text{ is a simple path expression} \\ \Pi^D(T_T(e)) & \text{if } Q = \text{distinct-values}(e) \\ T(Q, \#) & \text{if } Q \text{ is a FLWOR expression} \\ T_I(Q)[x] & \text{if } Q \text{ returns (a sequence of) items} \end{cases}$$

$$T_I(Q) := \begin{cases} \exists t \in T_T(R) : T_I(P) & \text{if } Q = \text{some } R \text{ satisfies } P \\ \forall t \in T_T(R) : T_I(P) & \text{if } Q = \text{every } R \text{ satisfies } P \\ f(T_I(e_1), \dots, T_I(e_n)) & \text{if } Q = f(e_1, \dots, e_n) \\ v & \text{if } Q \text{ is a variable reference to variable } \$v \\ c & \text{if } Q \text{ is constant } c \end{cases}$$

Figure 6. Translation of XQuery FLWOR expressions into the algebra.

Translation Function

Based on the properties mentioned above, we specify the translation procedure by means of three mutually recursive procedures T (see Figure 6). For a given query Q , $T_I(Q)$ translates Q into our algebra. The binary function $T(Q, A)$ is responsible for translating a FLWOR expression Q into the algebra. The first argument of this function is the (remainder of) the query to be translated, and the second argument is the algebraic expression constructed so far. The result of each translation step is a tree of algebraic operators which produce sequences of tuples. For each clause of the FLWOR expression, we give the corresponding translation rule. For non-FLWOR expressions, we use two different unary translation functions. Function $T_I(Q)$ translates a subexpression Q into a function with a simple return type in the XQuery data model, while function $T_T(Q)$ returns an algebraic expression which produces sequences of tuples. Notice that we rely on the translation presented by (Brantner et al., 2005) to translate simple path expressions. However, in contrast to that proposal, we do not fix the implementation of the location steps during translation. This decision is made during cost-based optimization instead. As a consequence, a wider range of optimizations is considered, e.g. using an index.

Since a FLWOR expression can occur within simple expressions and vice versa, these functions are mutually recursive. In the translation rule for the **let** clause we explicitly select the translation function to use: if the expression bound in the **let** clause is sequence-valued, this sequence is turned into a sequence of tuples. Otherwise, we use the translation function that returns single items.

Example Query

Let us consider the quantified query we have discussed above. Below, we repeat the result of normalization:

```
let $d := doc("bib.xml")
let $v1 := $d //book
for $t in $v1/ title
where some $v2 in $v1/author
    let $v3 := fn: data ($v2)
    satisfies
        some $v4 in $d //book/ editor
            let $v5 := data ($v4)
            satisfies $v3 eq $v5
return $t
```

We begin with the first **let** clause of the FLWOR expression. The translation results in:

$$X_{d:T(\text{doc}(\text{"bib.xml"}))}(\#)$$

After translating the function call in the subscript, we encounter another **let** clause.

$$X_{v1:T(\$d//book)}(X_{d:X_{d:\text{doc}(\text{"bib.xml"})}}(\#))$$

We continue with the **for** clause which is mapped to an unnestmap operator by the translation function.

$$Y_{r:T}(\$v1/title)(X_{v1:Y_{bd}/book}(\#)(X_{d:X_d.doc("bib.xml")}(\#)))$$

The **where** clause is translated into a selection operator. We have to translate the predicate recursively.

$$\sigma_{T(\dots)}(Y_{r:Y_{trv1}/title}(\#)(X_{v1:Y_{bd}/book}(\#)(X_{d:X_d.doc("bib.xml")}(\#))))$$

We continue with the first quantified expression.

$$\sigma_{\exists x \in T(\dots):T(\dots)}(Y_{r:Y_{trv1}/title}(\#)(X_{v1:Y_{bd}/book}(\#)(X_{d:X_d.doc("bib.xml")}(\#))))$$

To avoid clutter, we will refer to the result of translating the range expression of the first quantified expression by e_1 and to the result of translating the range predicate of this existential quantifier by e_2 . Thus, we get:

$$\sigma_{\exists x \in e_1:e_2}(Y_{r:Y_{trv1}/title}(\#)(X_{v1:Y_{bd}/book}(\#)(X_{d:X_d.doc("bib.xml")}(\#))))$$

The recursive translation of the range expression is similar to the translation of the **for** clause and **let** clause. The translation of the second quantified expression is also similar to the translation of the first quantifier:

$$e_1 := X_{v2:fn:data(v1)}(Y_{v1:Y_{av1}/author}(\#))$$

$$e_2 := \exists y \in X_{v4:fn:data(v3)}(Y_{v3:Y_{cc}/editor(Y_{c,d}/book)}(\#)) : v2 = v4$$

The last translation step consists of translating the **return** clause which introduces a projection.

$$\Pi_I(\sigma_{\exists x \in e_1:e_2}(Y_{r:Y_{trv1}/title}(\#)(X_{v1:Y_{bd}/book}(\#)(X_{d:X_d.doc("bib.xml")}(\#))))))$$

Clearly, the translation is a simple mapping of the normalized XQuery expression into our algebra. The resulting algebraic expression contains nested algebraic expressions – in this example existential quantifiers. In (May et al., 2004; May et al., 2006), we have demonstrated that after unnesting such algebraic expressions the query can be evaluated much more efficiently because the cost-based query optimizer has more choices to evaluate the query. For other algebraic optimizations we can expect similar positive effects.

Mapping to Calculus Representation

In the previous sections, we have presented our normalization steps as rewrites on the abstract syntax tree of the parsed XQuery query. We have also defined a translation function that maps XQuery constructs into our algebra. In Natix we integrate normalization, translation, and factorization of common subexpressions. We also assign a type to all translated constructs and annotate them with cost and cardinality information, see (May, 2007) for details.

The translation presented in the previous section yields a canonical operator tree as it is usually presented in database text books (Garcia-Molina et al., 2001). However, detecting patterns on such an algebraic expression is difficult and inefficient because the argument relationship is directly encoded into the algebraic expression. For many rewrites the exact argument relationship is not important. Such rewrites are more difficult to implement on algebra trees because pattern matching must consider more combinations of argument relationships. For this reason we do not translate the parsed query directly into an algebraic expression. Instead, our internal query representation unifies features of calculus and algebra. It is similar to the query graph model (Haas et al., 1989), (Shanmugasundaram et al., 2001). After the translation step, all steps of the Natix optimizer work on a common query representation. During cost-based optimization, it is turned into a representation closer to an algebraic expression annotated with implementation hints.

Our solution relies on the idea of blocks. Each block is able to capture the semantics of a FLOWR expression. It captures the order of expressions in a FLWOR expression, and at the same time it allows for efficient pattern matching.

Notation	Description
Π_A	the attributes A specified in the final projection
$P = \langle p_1 \dots p_k \rangle$	the producers P
$p = l_1 \wedge l_2 \wedge \dots \wedge l_m$	a conjunctive predicate
$C = \langle c_1 \dots c_n \rangle$	expressions c_i whose result is bound to a variable
$U = \langle u_1 \dots u_o \rangle$	sequence-valued expressions u_i whose result must be iterated over
$G = \langle g_1, \dots, g_p \rangle$	grouping attributes

Figure 7. Components of a block.

As Figure 7 shows each block contains a list of producers, P , similar to generators in a calculus expression, a list of AlgChi operators, C , each of which encapsulates the computation of an expression, a list of AlgUnnest operators, U , each of which represents the computation of a sequence-valued function whose result is immediately flattened, and a pointer to the parent block. Additionally it contains a projection list.

To illustrate our idea we first assume a FLWOR expression without **let** or **order by** clauses and with path expressions whose predicates are all moved into the **where** clause if possible. Then the semantics of a simple block is defined as the algebraic expression

$$\Pi_A(\sigma_p(Y_{p_n}(\dots(Y_{p_2}(Y_{p_1}(\#)))))).$$

Thus, the variable in the **return** clause constitutes the projection of the block. The **where** clause is represented by a selection operator, and the **for** clauses are implemented by a sequence of unnest map operators. When the producers p_i can be evaluated independently, we can turn the unnest map operators into cross products.

Remember that we also denote the concatenation of the application of algebraic operators with \circ . The semantics of the block is defined by the algebraic expression

$$\begin{aligned} & \Pi_A^{(i_1)} \circ \\ & \sigma_{l_1}^{(i_2)} \circ \sigma_{l_2}^{(i_3)} \circ \dots \circ \sigma_{l_k}^{(i_j)} \circ \\ & \quad X_{cn}^{(i_{j+1})} \circ X_{cn}^{(i_{j+2})} \circ \dots \circ X_{cn}^{(i_k)} \circ \\ & \quad \quad Y_{p_1}^{(i_{k+1})} \circ Y_{p_2}^{(i_{k+2})} \circ \dots \circ Y_{p_n}^{(i_{l-1})} \circ \\ & \quad \quad \quad \#^{(i_l)}. \end{aligned}$$

The superscript (i_x) denotes the permutation of these operators that is consistent with the given XQuery expression. For every query, we have $(i_1) = 1$ and $(i_l) = l$, i.e. the projection is the outer-most operator and the singleton scan is the inner-most operator of this expression. The list `theApplicationOrder` stored in a block represents this permutation of operators that maps positions in the resulting algebraic expressions to pointers of the operator at this position. Thus, after translation, the order of the entries in this list is consistent with the occurrence in the textual query representation. This is too restrictive because a partial order of the expressions would suffice. But later rewrites can simplify these order constraints. The list of grouping attributes is used to implement the `distinct` and `distinct-values` functions. Moreover, optimizations like unnesting may introduce grouping operations explicitly.

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

This chapter has presented the approach taken in the native XML database management system Natix to translate XQuery statements into algebraic expressions. As NAL, our algebra, is an extension of the relational algebra, we are able to leverage optimizations that proved useful for relational databases. However, as XQuery is based on a duplicate-aware and order-aware data model care is required. For

example, for unnesting nested query blocks we needed to reconsider optimizations known from the relational world.

Such optimizations are much easier to implement if the XQuery statement is normalized prior to the translation into the algebra. In this chapter, we have presented several rewrites we have used to normalize the parsed XQuery statement. However, this set of normalization rewrites is not complete yet, as Natix does not yet cover all features of XQuery. The formal notation of our rewrites allows for formal proofs of correctness of these rewrites – this is part of the future work. Some rewrites may change the semantics of the query statement (e.g. node identity), and thus they cannot be applied in these cases. As a consequence, several optimizations cannot be applied in subsequent steps.

Based on a normalized XQuery statement we perform a translation step into NAL, our algebra over sequences of tuples. Thanks to the prior normalization step, this translation function is rather straightforward. While this translation function is defined as a mapping of a normalized XQuery statement into an algebraic expression our implementation in Natix performs a translation into a representation closer to a calculus. The advantage of this approach is that optimizations are more efficient and easier to implement. In this chapter, we have outlined how NAL relates to this internal query representation.

Our experience clearly indicates that an algebraic approach to XQuery optimization is useful. First, it allows us to adapt optimization techniques known from relational databases for XML processing. Second, we are able to prove the correctness of optimizations which is part of future work. Finally, our algebraic approach to XML query processing can benefit from experience gained with implementing database systems, i.e. the benefit extends from the XQuery optimizer even to the query execution environment. The techniques introduced in this chapter are the basis for being able to benefit from these advantages.

REFERENCES

- Albert, J. (1991). Algebraic properties of bag data types. In *Proceedings of the seventeenth International Conference on Very Large Data Bases*. (pp. 211–219), Morgan Kaufmann, San Mateo, CA, USA.
- Amer-Yahia, S., Cho, S.R, Lakshmanan, L.V. S., & Srivastava, D. (2001). Minimization of tree pattern queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, (pp. 497–508), ACM press, New York.
- Astrahan, M., & Chamberlin, D. (1975). Implementation of a Structured English Query Language. *Communications of the ACM*, 18(10), (pp. 580–588), ACM press, New York.
- Balmin, A., Ozcan, F., Beyer, K.S., Cochrane, R., & Pirahesh, H. (2004). A framework for using materialized XPath views in XML query processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, (pp. 60–71), Morgan Kaufmann, San Mateo, CA, USA.
- Beeri, C. & Tzaban, Y. (1999). SAL: An algebra for semistructured data and XML. In *WebDB (Informal Proceedings)*, (pp. 37–42).
- Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., & Siméon, J. (2007). XQuery 1.0: An XML Query Language. World Wide Web Consortium (W3C), W3C Recommendation.
- Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., & Teubner, J. (2006). MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, (pp. 479–490), ACM press, New York.
- Brantner, M., Kanne, C.-C., Helmer, S., & Moerkotte, G. (2005). Full-fledged algebraic XPath processing in Natix. In *Proc. IEEE Conference on Data Engineering (ICDE)*, (pp. 705–716), 2005, IEEE Computing Society.

- Bry, F. (1989). Towards an efficient evaluation of general queries: quantifier and disjunction processing revisited. *In Proc. of the ACM SIGMOD Conf. on Management of Data*, (pp. 193–204), ACM press, New York.
- Chaudhuri, S. (1998). An Overview of Query Optimization in Relational Systems. *In Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, (pp. 34–43), ACM press, New York.
- Ceri, S. & Gottlob, G. (1985). Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4), (pp. 324–345).
- Cluet, S. & Moerkotte, G. (1993). Nested queries in object bases. *In Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Language*, (pp. 226–242). Springer.
- Dayal, U., Goodman, N., & Katz, R.H (1982). An extended relational algebra with control over duplicate elimination. *In Proc. ACM SIGMOD/SIGACT Conf. on Principles of Database Systems. (PODS)*, (pg 117–123), ACM press, New York.
- Draper, D., Fankhauser, P., Fernandez, M, Malhotra, A, Rose, K., Rys, M., Siméon, J., & Wadler, P (2007). XQuery 1.0 and XPath 2.0 Formal Semantics. *World Wide Web Consortium (W3C)*, W3C Recommendation.
- Fegaras, L., Levine, D., Bose, S., & Chaluvadi, V. (2002). Query processing of streamed XML data. *In Proceedings of the eleventh international conference on Information and knowledge management*, (pp. 126–133), ACM Press. New York.
- Fegaras, L. & Maier, D. (1995) Towards an effective calculus for object query languages. *In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, (pp. 47–58), ACM Press, New York.
- Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., & Westmann, T (2002). Anatomy of a native XML base management system. *VLDB Journal*, 11(4), (pp. 292–314).
- Florescu, D. & Kossmann, D. (2004), XML query processing (tutorial). *In Proceedings of the 20th International Conference on Data Engineering, ICDE 2004*, (pg. 874), IEEE Computer Society.
- Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M.J., & Sundararajan, A., (2004). The BEA streaming XQuery processor. *VLDB Journal*, 13(3), (pp. 294–315).
- Frasincar, F., Houben, G.-J., & Pau, C. (2002). XAL: An algebra for XML query optimization. *In Proc. Of the Thirteenth Australasian Database Conference (ADC2002)*, (pp. 49–56), Australian Computer Society, Inc. Darlinghurst, Australia.
- Garcia-Molina, H., Ullman, J.D., & Widom, J. (2001), *Database Systems: The Complete Book*. Prentice Hall.
- Gottlob, G., Koch, C., & Pichler, R. (2002). Efficient algorithms for processing XPath queries. *In Proceedings of the Twenty- Eighth International Conference on Very Large Data Bases*, (pp. 95–106). Morgan Kaufmann, San Mateo, CA, USA.

Grust, T., Rittinger, J. & Teubner, J. (2007). eXrQuy: Order indifference in XQuery. In *Proceedings of the 23rd IEEE Int'l Conference on Data Engineering (ICDE 2007)*, (pp. 226–235), IEEE Computer Society.

Grust, T. (2002), Accelerating XPath location steps. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, (pp. 109–120), ACM press, New York, NY, USA

Grust, T., Sakr, S., & Teubner, J. (2004). XQuery on SQL hosts. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. (pp. 252–263), Morgan Kaufmann, San Mateo, CA, USA.

Grust, T., & Teubner, J. (2004). Relational algebra: Mother tongue –XQuery: Fluent. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM) 04, (informal proceedings)*, 2004.

Haas, L.M., Freytag, J.C., Lohman, G.M., & Pirahesh, H. (1989). Extensive query processing in Starburst, In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, (pp. 377–388), ACM press, New York.

Helmer, S., Kanne, C.-C. & Moerkotte, G. (2002). Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE'02)*, (pp. 215–224). IEEE Computer Society.

Hidders, J. & Michiels, P. (2003). Avoiding unnecessary ordering operations in XPath. In *Database Programming Languages, 9th International Workshop, DBPL 2003*. (pp. 54–74), Springer.

Hidders, J., Paredaens, J., Vercammen, R., & Demeyer, S. (2004). A light but formal introduction to XQuery. In *Database and XML Technologies, Second International XML Database Symposium, XSym*, (pp. 5–20). Springer.

Hosoya, H. & Pierce, B. (2000). XDuce: A Typed XML Processing Language (Preliminary Report). *Proc. of WebDB (Selected Papers)*. (pp. 226–244), Springer.

Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L., Nierman, A., Papparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., & Yu, C., (2002). Timber: A native XML database. *VLDB Journal*, 11(4), 274–291.

Jarke, M., & Koch, J. (1984). Query optimization in database systems. *ACM Computing Surveys*, 16(2), (pp. 111–152).

Kay, M. (2008). Ten Reasons Why Saxon is Fast. *IEEE Data Eng. Bull.* 31(4), (pp. 65–74).

Krishnamurthy, R., Kaushik, R., & Naughton, J.F. (2003). Xmysql query translation literature: The state of the art and open problems. In *Proc. of the First International XML Database Symposium, XSym 2003*, (pp. 1–18), Springer.

Lerner, A. & Shasha, D. (2003). AQuery: Query language for ordered data, optimization techniques, and experiments. In *Proceedings of the 29th international conference on Very large data bases*, (pp. 345–356), VLDB Endowment.

- Liu, Z.H., Krishnaprasad, M., and Arora, V. (2005). Native XQuery processing in Oracle XML DB. *In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, (pp. 828–833), ACM press, New York.
- Maier, D. (1983). *The Theory of Relational Databases*. Rockville, Maryland, Computer Science Press.
- Manolescu, I., Florescu, D., & Kossmann, D. (2001). Answering XML queries on heterogeneous data sources. *In Proceedings of the 27th International Conference on Very Large Data Bases*, (pp. 241–250), Morgan Kaufmann, San Mateo, CA, USA.
- May, N. (2007). *An Algebraic Approach to XQuery Optimization*. doctoral dissertation, University of Mannheim.
- May, N., Helmer, S., & Moerkotte, G. (2004). Nested queries and quantifiers in an ordered context. *In Proceedings of the 20th International Conference on Data Engineering (ICDE)*, (pp. 239–250), IEEE Computer Society.
- May, N., Helmer, S., & Moerkotte, G. (2006). Strategies for query unnesting in XML databases. *ACM Transactions on Database Systems*, 31(3), 968–1013.
- Michiels, P., Hidders, J., Siméon, J., & Vercammen, R. (2006). How to recognize different kinds of tree patterns from quite a long way away. Technical Report TR UA 13-2006, University of Antwerp.
- Miklau, G. & Suciu, D. (2002), Containment and equivalence for an XPath fragment. *In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (pp. 65–76). ACM press, New York.
- Nakano, R. (1990). Translation with optimization from relational calculus to relational algebra having aggregate functions. *ACM Transactions on Database Systems*, 15(4), (pp. 518–557).
- Naughton, J. F., DeWitt, D. J., Maier, D., Aboulnaga, A., Chen, J., Galanis, L., Kang, J., Krishnamurthy, R., Luo, Q., Prakash, N., Ramamurthy, R., Shanmugasundaram, J., Tian, F., Tufte, K., Viglas, S., Wang, Y., Zhang, C., Jackson, B., Gupta, A., & Chen, R. (2001). The Niagara internet query system. *IEEE Data Eng. Bull.*, 24(2), (pp. 27–33).
- M. Negri, M., Pelagatti, G., & Sbatella, L. (1991). Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 16(3), (pp. 513–534).
- Nicola M., and van der Linden, B. (2005). Native XML support in DB2 universal database. *In Proceedings of the 31st international conference on Very large data bases*, (pp. 1164–1174), VLDB Endowment.
- Ozcan, F., Cochrane, R., Pirahesh, H., Kleewein, J., Beyer, K.S., Josifovski, V., & Zhang, C. (2005). System RX: One part relational, one part XML. *In Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, (pp. 347–358). ACM press, New York.
- Pal, S., Cseri, I., Seeliger, O., Rys, M., Schaller, G., Yu, W., Tomic, D., Baras, A., Berg, B., Churin, D., & Kogan, E. (2005). XQuery implementation in a relational database system. *In Proceedings of the 31st international conference on Very large data bases*, (pp. 1175–1186), VLDB Endowment.

Re, C., Siméon, J., & Fernández, M.F. (2006). A complete and efficient algebraic compiler for XQuery. *In Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. (pg 14), IEEE Computer Society.

Shanmugasundaram, J., Kiernan, G., Shekita, E.J., Fan, C., & Funderburk, J.E. (2001). Querying XML views of relational data. *In Proceedings of the 27th International Conference on Very Large Data Bases*, (pp. 261–270), Morgan Kaufmann, San Mateo, CA, USA.

Slivinskas, G., Jensen, C.S., & Snodgrass, R.T. (2002). Bringing order to query optimization. *SIGMOD Record*, 31(2), (pp. 5–14).

Steenhagen, H.J., Apers, P.M.G., Blanken, H.M., & de By, R.A. (1994), From nested-loop to join queries in OODB. *In Proceedings of the 20th International Conference on Very Large Data Bases*, (pp. 618–629), Morgan Kaufmann, San Mateo, CA, USA.

Suciu, D. (2001). On database theory and XML. *SIGMOD Record*, 30(3), (pp. 39–45).

von Bülzingsloewen, G. (1987). Translating and optimizing sql queries having aggregates. *In Proceedings of the 13th International Conference on Very Large Data Bases*, (pp. 235–243), Morgan Kaufmann, San Mateo, CA, USA.

Wong, E., & Youssefi, K. (1976). Decomposition a strategy for query processing. *ACM Transactions on Database Systems*, 1(3), (pp. 223–241).

ADDITIONAL READINGS SECTION

Books

Chamberlin, D, Draper, D, Fernández, M.F., Kay, M., Robie, J., Rys, M., Siméon, J, Tivy, J., & Wadler, P. (2004) *XQuery from the Experts – A Guide to the W3C XML Query Language*. Addison Wesley.

Melton, J., Buxton, S. (2006). *Querying XML: XQuery, XPath, and SQL/XML in Context*, Morgan Kaufmann, San Mateo, CA, USA.

Journal Special Issues

Various Authors (2002). Special Issue on XML Data Management, *VLDB Journal*, 11(4), Springer.

Various Authors (2006). Celebrating 10 Years of XML, *IBM Systems Journal*, 45(2), IBM.

Various Authors (2008). Special Issue on XQuery Processing: Practice and Experience, *Bulletin of the Technical Committee on Data Engineering*, 31(4), IEEE Computer Society.

Doctoral Dissertation

Teubner, J. (2006). *Pathfinder: XQuery Compilation Techniques for Relational Database Targets*. doctoral dissertation. Technische Universität München.

May, N. (2007). *An Algebraic Approach to XQuery Optimization*. doctoral dissertation, University of Mannheim.