XQuery Query Processing in Relational Systems

by

Yingwen Chen

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With the rapid growth of XML documents to serve as a popular and major media for storage and interchange of the data on the Web, there is an increasing interest in using existing traditional relational database techniques to store and/or query XML data. Since XQuery is becoming a standard XML query language, significant effort has been made in developing an efficient and comprehensive XQuery-to-SQL query processor.

In this thesis, we design and implement an *XQuery-to-SQL Query Processor* based on the *Dynamic Intervals* approach. We also provide a comprehensive translation for XQuery basic operations and FLWR expressions. The query processor is able to translate a complex XQuery query, which might include arbitrarily composed and nested FLWR expressions, basic functions, and element constructors, into a single SQL query for RDBMS and a physical plan for the *XQuery-enhanced Relational Engine*.

In order to produce efficient and concise SQL queries, succinct XQuery to SQL translation templates and the optimization algorithms for the SQL query generation are proposed and implemented. The preferable *merge-join* approach is also proposed to avoid the inefficient *nested-loop* evaluation for FLWR expressions. *Merge-join* patterns and query rewriting rules are designed to identify XQuery fragments that can utilize the efficient *merge-join* evaluation. Proofs of correctness of the approach are provided in the thesis. Experimental results justify the correctness of our work.

Acknowledgement

My two-years' adventure at the University of Waterloo turns out to be a highly rewarding experience. I was able to not only upgrade my professional knowledge and research skills, but also explore my interests and reshape my attitude for life and challenges.

I owe all these accomplishments to my supervisor, David Toman, who led me into the Master's program and guided me throughout the whole process of the study. His profound knowledge, insight and expressive clarity have guided me to overcome obstacles during the research and the preparation of the thesis. I am deeply grateful for his constant support and endless patience. Whenever I was in doubt he never hesitated to offer his time and assistance. His encouragement and understanding reinforce my confidence in my research skills. Studying under his supervision is one of the luckiest things in my life. His enthusiasm for research and passion for life have been an inspiration during my two-years' study and will continue to inspire me in my future career.

My heartfelt thanks to Professor M. Tamer Özsu and Professor Grant Weddell for reviewing the thesis. Many thanks for their valuable comments and encouragement.

Always, my unending gratitude goes to my family for their endless love and support. I am forever indebted to my parents, Enqiang Chen and Hanying Li, who never stop giving me a world of love. Special thanks to Yuwen for being a great brother. My ongoing love and thanks to my husband for his understanding and support.

Dedication

To my parents, Enqiang and Hanying, who are the best parents in the world.

And to my husband, Yunxiang, for his love.

Trademarks

- DB2, DB2 Universal Database are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
- Oracle is a registered trademark of Oracle Corporation.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Problem Scenario

With the rapid growth of XML document to serve as a popular and major media for the storage and interchange of the data on the Web, much work has been done to explore the effective techniques to store, query, and retrieve XML data. One of the important research directions is to use the existing relational database techniques to store and/or query XML data.

Various problems have been addressed and solved in the XML-relational problem domain. Some issues and approaches are related to publishing existing relational data in XML as a uniform data source and translating XML queries into SQL to query the actual relational data source [11, 14, 16]; others are related to storing XML data in RDBMS and evaluating XML queries by translating them into SQL queries over relational tables that represent the XML data [9, 13, 19, 21]. A wide variety of XML query languages, e.g., XQuery [4], XPath, XML-QL, are used in this problem space. Among these languages XQuery is quickly becoming the standard XML query language. Thus there is a growing interest in evaluating XQuery queries using relational techniques.

1.2 Why Use Relational Techniques?

There are several reasons for the interaction between XML and relational database systems. One reason is the need of integrating different sources of data, such as XML data and relational data, under a global XML schema. Another reason

is the requirement to store XML data in a relational database or to construct XML documents from relational data. A typical reason lay to the advantage of utilizing the sophisticated storage, highly optimized query processor and powerful data management services provided by mature Relational Database Management Systems (RDBMS).

1.3 Brief Overview of XQuery

XQuery is a strongly-typed and functional language that is still under development by the World Wide Web Consortium (W3C) [4]. The basic building block of XQuery is an *expression*. Since XQuery is a functional language, expressions can be arbitrarily nested. XQuery includes a wide variety of functions and operators, which include arithmetic functions, string and regular expressions, element constructors, boolean comparison operations, sequence construction, logical operations, function calls, etc.

Path expressions are particularly designed to navigate nodes within XML trees. Sequences of resulting nodes obtained by evaluating a path expression are combined in their original document order. FLWR expressions are the core building blocks of XQuery and the name comes from the *for*, *let*, *where*, *return* keywords used by the expressions. The *for* clauses provide variable iteration over intermediate results; the *let* clauses assign sub-expressions to variables. These assignments can be filtered by the *where* clauses. The *return* clauses construct ordered sequences of results that should be returned by the expression in the *return* clauses. FLWR expressions are very useful in restructuring XML data.

1.4 Challenges

Since XQuery is emerging as the standard of XML query languages, considerable effort has been directed towards developing XQuery query processor. There are two major approaches. The first approach is to implement a native XQuery processor; the other one is to develop an efficient and comprehensive XQuery-to-SQL query processor. The thesis pursues the second approach. However, not all features of XQuery can be translated into SQL queries. Researchers are facing many difficulties in evaluating XQuery queries using relational engines. One typical difficulty comes from the arbitrarily nesting nature of FLWR expressions. Some features of XQuery are difficult to translate because of the semantic mismatch between XQuery and

SQL language. A typical semantic mismatch lays to the fact that XQuery handles ordered XML data while the SQL handles unordered relational data. Preserving document order while evaluating arbitrarily nested expressions is also a great challenge for the implementation of an XQuery query processor. For example, [14] claims that it is impossible to map an order-based nested XQuery query to a single equivalent SQL query without the help of materialized intermediate XML results.

Other difficulties arise in XML-to-relational mappings. A typical challenge may lie in the mismatch between the complex recursive structure of XML data and the simple flat structure of relational data. Although there are many challenges in evaluating XML queries in RDBMS, there is a growing interest in this research area because of the significant advantage of utilizing the mature RDBMS technology.

Among the approaches to XQuery-to-SQL translation, the *Dynamic Intervals* technique [9] handles the core fragment of XQuery with arbitrarily nested FLWR expressions. [9] proposed a comprehensive translation of XQuery expressions to SQL queries that enables relational engines to produce predictably good query plans. The *Dynamic Intervals* approach overcomes the difficulties that arise from the incompatible features between XQuery and SQL language. The thesis continues that work and makes further improvements[1].

1.5 Related Work

In recent years, many techniques have been proposed for various issues in utilizing the mature RDBMS techniques to store and query XML documents. There are two major subproblems to the problems in this domain:

- What kind of XML-relational mapping schema is used to store and retrieve the XML data from RDBMS, or retrieve relational data in the view of XML?
- Given an XML query, how to convert it into one or more SQL queries, which can be evaluated in RDBMS to obtain the results?

There is a diversity of approaches to the problems and can be classified into two main categories based on the research goals. One kind of approaches focus on the XML publishing problems, querying existing relational data viewed as XML [11, 14, 16]. The other approaches are about XML storage and the use of RDBMS to store and query existing XML data [9, 10, 13, 18, 19, 21]. The thesis focuses on

[1]The improvements are summarized in Section 1.6.

the XML storage scenario, encoding XML documents as relational data and using relational engines to evaluate XQuery queries.

1.5.1 XML Publishing Scenario

Many techniques have been published to handle XML publishing using either local-as-view approach (LAV), or global-as-view approach (GAV), or both, to define an XML view of relational data [11, 14, 16]. In this scenario we are interested in the part of translating XML queries into SQL queries.

XPeranto [16] proposes a general framework for processing arbitrarily complex XQuery queries with features such as nested expressions and nested order over XML views of relational data. Given an XQuery, the query is converted to an internal query representation called *XML Query Graph Model* (XQGM). The XQGM is then modified to eliminate the construction of intermediate XML fragments and to push down predicates. Finally, the modified XQGM is translated into a single SQL query to be evaluated in RDBMS. Although XPeranto claims to be able to handle a general XQuery, the class of XQuery queries it actually handles is unclear.

Agora integration system [14] uses the LAV approach under an XML global schema and handles XQuery FLWR expressions. The XQuery-to-SQL translation proceeds in three steps. First, the input XQuery is normalized based on certain rewriting rules. Then, the rewritten XQuery query is translated into a SQL query on the virtual generic relational schema. Finally, the SQL query is rewritten based on the real relational storage schema. [14]'s state-of-the-art query rewriting algorithms for SQL semantics do not efficiently handle arbitrary levels of nesting, grouping, etc. XQuery expressions that rely on document order, etc., cannot be translated.

MARS system [11] handles publishing as XML data from the mixed (relational+XML) storage. It translates XQuery queries into SQL by rewriting XQuery queries into a set of decorrelated queries (called XBind query), which split an XQuery query into the navigation part and the tagging template. The XBind queries along with the views and integrity constraints are compiled to produce relational queries and constraints. Then a `ChaseandBackChase` (C&B) algorithm is used to find a minimal reformulation of relational queries under relational integrity constraints. Although [11] deals with XQuery, it does not precisely characterize the class of XQuery queries it handles.

Other XML publishing techniques for evaluating XML queries in RDBMS have been studied in [12, 22].

1.5.2 XML Storage Scenario

In the XML storage scenario, the goal is to design a relational schema for storing XML data in relations and to covert XML queries into SQL queries that can be evaluated in RDBMS. The work of the thesis falls into this research area.

Many XML-to-relational encoding schema have been designed for XML storage. Some of them deal with arbitrary XML documents without considering XML schema [9, 10, 13, 19, 21]. Others select relational schema based on an XML schema or a DTD [7, 8, 17, 18].

[7] proposes a cost-based approach for the choice of a relational encoding schema in situations in which an XML schema, an XQuery workload, and the target XML application are provided. The goal of [7] is to maximize query performance for a given application. However, [7] does not propose translation algorithms for XQuery queries. [18] uses DTD to generate the relational encoding schema and stores the XML data in relational tables based on the encoding schema. It only handles XML-QL queries. Other relational encodings for storing XML data in relations based on the XML schema or DTD have been studied [8, 17]. However, most of these published works illustrate the XML query translation by examples; few of them gave details about the query translation algorithm.

Many relational encodings that handle arbitrary XML documents have been proposed [13, 19, 21]. [13] proposes an edge-based encoding to view an arbitrary XML document as a graph and store all the edges of the graph in a single table without considering XML schema and DTD. XRel [21] introduces a path-based interval encoding that views XML documents as trees and stores an arbitrary XML document in relational tables according to the node type and the root-to-node path information. For each element, a path id is stored for each root-to-node path. XRel provides an algorithm for translating a core part of XPath expressions called *XPathCore* into SQL queries. XRel efficiently handles path expressions with predicates over nonrecursive data sources, however, it does not address mapping the FLWR expressions into relational queries.

Encodings that preserve document order have been studied in [19], where *order encoding methods* that represent XML order in the relational data model are used to translate order based XPath expressions into SQL. The approach is a modified edge-based encoding that stores extra order information along with document structure and data. For each node in XML trees, information of the node's global document order, position among its siblings and path information for each root-to-node path are recorded. [19] handles XPath expressions with positional predicates but hardly considers the FLWR expressions as well.

Other relational encodings that deal with arbitrary XML documents are proposed by [9, 10, 20]. [10] uses two schema to store XML data: relational schema and overflow graph. It provides an algorithm to translate STORED query, which is query language used to express a mapping between XML data and relational data, into SQL.

In the XML storage scenario, most of the relational encodings proposed only deal with a limited subset of XQuery/XPath expressions, such as restricted path expressions, and do not handle the arbitrarily-nested FLWR expressions and the construction of new nested documents. Among these approaches, [9] is only one that handles more general XQuery queries. [9] proposes a novel encoding technique called *Dynamic Intervals*, which is an improved interval encoding that handles arbitrary XML documents and fully supports arbitrarily nested FLWR expressions, arbitrary combinations and nesting of basic functions, element constructors, XPath expressions, etc. Such encoding technique enables the preserve of document order of the results throughout the entire query evaluation for nested FLWR expressions. [9] provides a comprehensive SQL translation for the core features of XQuery. Using the *Dynamic Intervals* approach, a complex XQuery query, which might contain arbitrarily nested FLWR expressions, element constructors and built-in functions, can be translated into one single SQL query while preserving the required document order.

The *Dynamic Intervals* approach handles a much larger class of XQuery queries than the other approaches introduced here. The thesis continues the work of [9] to implement an XQuery-to-SQL query processor. The thesis also develops optimization approaches to produce preferable relational plans.

1.6 Contributions

The contributions of the thesis are summarized as follows:

- We design and implement an *XQuery-to-SQL Query Processor* based on the *Dynamic Intervals* approach introduced in [9]. The query processor handles a comprehensive subset of XQuery queries. Given a complex XQuery query, which might contain arbitrarily nested FLWR expressions, element constructors and basic functions, the query processor is able to translate the query into one single SQL query for RDBMS and one single *physical plan* for the *XQuery-enhanced Relational Engine* [15].

- We correct several imprecisions in the SQL translations of XQuery expressions in [9] and provide a comprehensive translation for XQuery basic operations and FLWR expressions.

- We propose and implement a series of optimization approaches to produce efficient and concise SQL queries. The optimization approaches include the succinct SQL translation templates for XQuery expressions, the optimization algorithms for the SQL query generation, etc.

- We propose the preferable *merge-join* approach to avoid the inefficient *nested-loop* evaluation for FLWR expressions. *Merge-join* patterns and query rewriting rules are designed to capture a significant number of the XQuery fragments that can utilize the efficient *merge-join* evaluation. Proofs of correctness for the approach are also provided in the thesis.

- The experimental results justify the correctness of our work.

1.7 Thesis Organization

This chapter has introduced the framework of the XML-relational problem scenarios and provided a brief overview of the related research work. The rest of the thesis is organized as follows:

- Chapter 2, *Preliminary Knowledge*, is an overview of the background concepts and techniques based on which our *XML-to-SQL Query Processor* is built. The content in this chapter is a revised version of that in [9], where several imprecisions of the SQL translations for XQuery expressions are corrected.

- Chapter 3, *XQuery-to-SQL Query Processor*, provides a detailed introduction to the *XQuery-to-SQL Query Processor* implemented as part of the thesis. This chapter presents comprehensive SQL translation templates for XQuery expressions handled. Optimization algorithms for SQL query generation are also introduced in this chapter.

- Chapter 4, *Optimization of the XQuery-to-SQL Translation*, introduces the optimized SQL translation for XQuery expressions and proposes the *merge-join* approach to efficiently handle the FLWR expressions. SQL translation for the *merge-join* approach and proofs of correctness for the approach are also provided in this chapter.

- Chapter 5, *Experiments*, provides part of the experiments we conduct to verify the correctness of our work.
- Chapter 6, *Conclusions and Future Work*, summarizes the thesis and shows a road map for future work based on the thesis.

A comprehensive translation for XQuery basic operations is provided in Appendix A. Appendix B contains the rest of the experiments we conduct.

Chapter 2

Preliminary Knowledge

In the thesis, a relational XQuery processor is implemented based on the *Dynamic Intervals* approach introduced in [9], where a simple relational schema for XML data and an XQuery-like language that captures core features of XQuery are proposed. This chapter provides a brief overview of background concepts, techniques and assumptions, on which our XQuery-to-SQL query processor is built.

2.1 XML Document Definition

Ever since the first W3C Recommendation for *XML 1.0* was published in February 1998, different versions of XML specifications have been proposed, such as *XML 1.1*. Most of the models for XML data describe an XML document as a tree structure. [9] introduces a concise syntax for XML documents by describing an XML document as an ordered forest composed of a sequence of rooted, node-labelled trees.

The XML forest (XF) proposed by [9] is constructed using three kinds of constructors: the empty forest constructor ([ ]), the element constructor (<s> XF </s>), which construct a single tree by adding a root labelled "*s*" to the forest XF, and the concatenation constructor (XF @ XF), which concatenates two ordered forests.

Although this formulation of an XML forest is quite simple, it can still distinguish node identity and node types (*element*, *attribute* and *text*), given additional encoding conventions that relate either to node labelling or to a subtree pattern. For example, an XML element "<*book year='1994'*>" can be encoded as a subtree of the element node *book*, and the attribute node is labelled as "*attribute:year*". Given an XML document, all the attribute nodes are encoded as subtrees of their

corresponding element nodes and are labelled by adding a prefix "*attribute:*" to the attribute names. The attribute values can be encoded as subtrees of their attribute nodes and are labelled by adding a prefix "*text:*" to the attribute values. The text nodes are encoded in the same way as the attribute value. Figure 2.1 illustrates the process of converting the original XML data to the XML forest that matches the abstract syntax.

Original XML data:

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        ...
    </book>
  ...
</bib>
```

The corresponding XML forest in the abstract syntax:

```
<bib>
    <book>
      <attribute:year> <text:1994> </text:1994> </attribute:year>
        <title>
            <text:TCP/IP Illustrated> </text:TCP/IP Illustrated>
        </title>
        <author>
            <last> <text:Stevens> </text:Stevens> </last>
            <first> <text:W.> </text:W.> </first>
        </author>
        ...
    </book>
  ...
</bib>
```

Figure 2.1: Converting XML Data to the Abstract Syntax XML Forest

Since converting XML documents to the abstract syntax XML forests is not the focus of the thesis, in the rest of the thesis, we do not handle this issue any more. By default, the XML documents discussed in the scope of the thesis are in the format of *abstract syntax XML forests*.

2.2 Syntax of Minimal XQuery

In [9], a more compact definition for the XQuery core language, called *Minimal XQuery*, is introduced. *Minimal XQuery* expressions can be classified into three major categories: the variable expression, the FLWR expressions and the basic operations on XML forests.

2.2.1 Variable Expression

The variable expression ($e ::= x$) is equivalent to the *variable reference* in the XQuery *primary expressions* [4]. A variable expression must match a name in the input environment that contains the values (forest trees) for local variables. A variable may be added to the environment by a *for* or *let* expression, or be provided at the beginning of an XQuery query.

2.2.2 Basic Operations

Basic operations ($e ::= \mathsf{XFn}(e_1, \ldots, e_k)$) in *Minimal XQuery* enable various XQuery expressions, including constructors, path expressions, comparison expressions, etc. Constructor operations include *element constructor*, *empty forest constructor* and the *concatenation operator*. These constructors are often used to construct the abstract syntax XML forests introduced in Section 2.1.

Path expressions can be constructed using the *Horizontal Operations* and *Vertical Operations* introduced in [9]. A path expression is a sequence of one or more steps separated by / or //. For a path expression, $\$people/person$ for example, the equivalent expression in the *Minimal XQuery* syntax is as follows:

$$\texttt{select}('person', \texttt{children}(\$people))$$

The expression is constructed in two steps: first, a *children* operation is used to obtain all the children of the *people* element; second, a *select* operation is applied to the result of the first step to select the trees whose root nodes are labelled "*person*". Analogous approach is applied to the ancestor-descendant(//) step by using a *subtreesdfs*[1] operation at the first step.

[1] *subtreesdfs* is an XQuery basic operation that returns an XML forest containing all of the subtrees of an input XML forest in the *depth-first-search* order. Details about XQuery basic operations can be found in [9].

[9] also handles *path expression with predicates* using *head* and *tail* operations. For instance, the XQuery path expression "*book*[1]", which is to return the first *book* element in the *book* element forest, is equivalent to the expression "`head`(*book*)" in the syntax of *Minimal XQuery*. For the path expression with existential predicates, $people/person[@id = 'person0']$, beside using *children* and *select* operations, a *where* expression is also used to filter the unexpected *person* elements.

One important kind of XQuery expressions are the comparison expressions. Boolean operations (*empty*, *equal*, and *less*) are used to construct comparison expressions that serve as the boolean conditions within the *where* clauses of FLWR expressions.

2.2.3 FLWR Expressions

FLWR expressions are the core features of XQuery that are used to combine data from one or more XML documents. They support local variables bindings and iterations over intermediate results.

The *for* clauses ($e ::= \mathsf{for}\, x \in e\, \mathsf{do}\, e'$) and the *let* clauses ($e ::= \mathsf{let}\, x = e\, \mathsf{in}\, e'$) in a FLWR expression generate binding values (XML forests) for variables. The bindings of variables form an *environment* to supply values for free variables in subexpressions of a FLWR expression. The *for* clause binds its variable to the iteration over the resulting subtrees obtained by evaluating its associated expression. The *let* clause, however, binds its variable to the resulting subforest of its expression without iteration.

A *where* clause ($\mathsf{where}\, \varphi\, \mathsf{return}\, e$) filters the values (XML forests) using its boolean condition. The *return* clause is evaluated using the values of the bound variables and composes the result of the FLWR expression.

2.3 Interval Encoding: A Relational Schema Design

As we mentioned in Chapter 1, there are two major problems in translating XQuery to SQL:

- XML-to-relational mapping schema: which relational schema should be used for XML documents to be stored into and retrieved from a relational system?

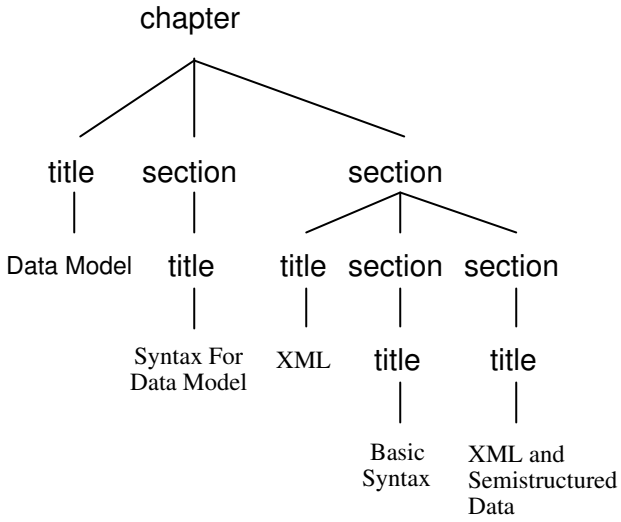


Figure 2.2: XML Forest of Books.xml

- Query translation: what kind of algorithms are used to translate XML queries into SQL queries?

2.3.1 Initial Interval Encoding

The relational schema chosen in [9] is an interval-based schema. The basic idea is that, for each node in an XML forest, a tuple consisting of the node's label and the interval, which is associated with the region representing the subtrees under the node, are used to represent the node. To encode an XML document, the XML document is first rewritten in the form of the abstract syntax XML forest introduced in Section 2.1. Then, the XML forest is stored in a single relational table where exists one and only one tuple (s, l, r) for each element, attribute and leaf text node. Each tuple contains a string of node label name, s, a natural number representing the left endpoint value of the associated interval, l, and a natural number representing the right endpoint value of the interval, r. Detail definition for the interval encoding can be found in [9]. An example of the interval encoding for the sample XML data "*books.xml*" taken from XML Use Cases[3], is shown in Figure 2.3. The source XML forest of "*books.xml*" is shown in Figure 2.2.

s	l	r
chapter	0	29
title	1	4
Data Model	2	3
section	5	10
title	6	9
Syntax For Data Model	7	8
section	11	28
title	12	15
XML	13	14
section	16	21
title	17	20
Basic Syntax	18	19
section	22	27
title	23	26
XML and Semistructured Data	24	25

Figure 2.3: Initial Encoding of Sample Data books.xml

2.3.2 Dynamic Interval Encoding

The goal of applying *Dynamic Interval Encoding* is to enable evaluating arbitrary FLWR expressions within a single relational query. *Environment* E is introduced as combinations of the values (XML forests) of the bound variables. A combination of bindings for the bound variables forms a *subenvironment* in the environment E. [9] introduces a dynamic interval encoding, which includes an index set I and relational representations $\{T_{x_1}, \ldots, T_{x_m}\}$ for the bound variables, to relationally represent sequences of environments generated within the query evaluation process. The index set I, which contains natural number index values, is used to identify the range of the values of a binding for the bound variables within the whole interval encoding.

The *width* of a relational representation of an XML forest is defined as follows:

- for the initial relational encoding of an XML document (XML forest), a natural number w is assigned as its *width*. This value is greater than the maximum value of the right endpoint r in the initial relational encoding;
- for the representation of the XML forests resulting from a evaluation of an XQuery expression, the value of the *width* is determined by performing arithmetic computation on the *widths* for the expression's subexpressions. Functions for computing the *widths* for various XQuery expressions can be found in [9].

For an environment with an index set value i, the related tuples (s, l, r) in T_{x_i} (with a width w_{x_i}) for a bound variable x_i satisfy the inequalities: $i * w_{x_i} \leq$

I	$T_{\texttt{title}}$		
i	s	l	r
0	`title`	2	5
	`Data on the Web`	3	4
	`title`	16	19
	`Advanced Programming in the Unix environment`	17	18
	`title`	30	33
	`TCP/IP Illustrated`	31	32

Figure 2.4: Title in an Initial Environment

l `AND` $r < (i+1) * w_{x_i}$. The dynamic interval representation enables the values bound to variables to keep track their corresponding environments while being able to keep the values of a variable as the concatenation of the results over sequences of environments. A general definition for *Dynamic Interval Encoding* can be found in [9]. Example 2.3.1 illustrates the dynamic interval encoding for a path expression.

Example 2.3.1 For the following path expression,

```
document("reviews.xml")/reviews/entry/title,
```

Figure 2.4 shows the resulting dynamic interval encoding $T_{\texttt{title}}$ for the path expression evaluated using the initial environment. The initial environment contains only an index set I, which has only one tuple (0). The path expression has a resulting width that is the same as the width of the source XML document: $w_{\texttt{title}} = w_{\texttt{document}} = 44$. □

2.4 XQuery to SQL Translation

The essential idea of *Dynamic Intervals* approach is to begin with an initial interval encodings for the XML documents and construct the dynamic interval encoding for the intermediate results during the query evaluation.

Based on the semantic of the *Minimal XQuery*, SQL translation templates for the XQuery expressions are used to translate a complex XQuery query into a SQL query by composing the SQL templates for the subexpressions of the query. Each SQL template can be treated as a relational view with input table parameters $(T_{x_1}, \ldots, T_{x_m})$, which are the SQL fragments for the sub-expressions.

2.4.1 Simple SQL Templates for Basic Operations

The translations for basic operations in Minimal XQuery can be firstly simplified by assuming that the basic operation is evaluated within one unique input environment instead of sequences of environments. For each basic operation in *Minimal XQuery*, there is a corresponding SQL template along with a width function to compute the width of the resulting XML forest. Example 2.4.1 illustrates the SQL translation for a complex XQuery query.

Example 2.4.1 Consider the following XQuery expression:

$\texttt{children}(\texttt{document}('reviews.xml'))$

The SQL template for the *document* constructor, $\mathsf{document}('reviews.xml')$, is defined as follows:

```
CREATE VIEW DOCUMENT(reviews) AS
  SELECT s, l, r
  FROM   reviews
```

The width of the result for the *document* constructor is $w_{\mathsf{document}} = 44$. The SQL template for translating the *children* operation, $\texttt{children}(T_e)$, is defined as follows:

```
CREATE VIEW CHILDREN(T_e) AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM   T_e u
  WHERE EXISTS (
    SELECT *
    FROM   T_e v
    WHERE  v.l < u.l AND u.r < v.r )
```

The resulting width of the *children* operation is exactly that of the input parameter T_e. The final SQL query for the XQuery expression is composed as follows:

```
SELECT u.s AS s, u.l AS l, u.r AS r
FROM
  ( SELECT s, l, r
    FROM   reviews  ) u
WHERE EXISTS (
  SELECT *
  FROM
     ( SELECT s, l, r
       FROM   reviews  ) v
  WHERE  v.l < u.l AND u.r < v.r  )
```

The width for the results of the XQuery expression is $w_{\mathsf{children}} = w_{\mathsf{document}} = 44$. □

2.4.2 SQL Translation for Minimal XQuery Expressions

When FLWR expressions are evaluated over sequences of separated environments, the environment index set I for a sequence of environments should be involved in the SQL translation templates. The following paragraphs show the SQL translation templates for basic operators (functions), *let* expressions, *where* expressions, and *for* expressions.

Basic XQuery Functions

In order to evaluate XQuery basic operations over a sequence of environments, we need to introduce the environment index set I in the translations. In the translation, the input tuples for a basic operation are first separated based on the environments they belong to by using the index values $i \in I$; then, the basic operation is evaluated over the separated input tuples. The result of the basic operation is the concatenation of the results of the evaluations over the sequence of environments. A revised SQL template based on that from [9] is shown as follows:

```
CREATE VIEW T_XFn(T_x1, ..., T_xm) AS
 SELECT s, l + i * w_XFn AS l, r + i * w_XFn AS r
 FROM   Q_XFn (
  ( SELECT i, s, l − i * w_x1, r − i * w_x1
    FROM   I, T_x1
    WHERE  i * w_x1 ≤ l  AND r < (i + 1) * w_x1 ),
    ···,
  ( SELECT i, s, l − i * w_xm, r − i * w_xm
    FROM   I, T_xm
    WHERE  i * w_xm ≤ l  AND r < (i + 1) * w_xm )
 )
```

where w_{x_i} represents the width of the input table parameter T_{x_i} for the SQL template and w_{XFn} is the width of the output. This revised template enables us to simplify the SQL translations for some basic operations such as the *children* operation and the *roots* operation.

LET Expressions: $\mathsf{let}\, x = e\, \mathsf{in}\, e'$

The *let* expression is an assignment expression. The expression e in the *let* expression is first evaluated using its input environment $E = \{I, T_{x_1}, \ldots, T_{x_m}\}$. As a result of the evaluation, the environment is increased by adding a binding of $x = e$. The increased environment $E' = \{I, T_{x_1}, \ldots, T_{x_m}, T_x\}$ serves as an input environment to evaluate expression e'. The templates for creating the new environment can be found in [9].

WHERE Expressions: $\mathsf{where}\, \varphi\, \mathsf{return}\, e$

The *where* expression is a filter expression that filters out the undesired results which are derived from certain environments. In other words, the *where* expression actually filters out the unwanted environments within the sequence of input environments using its condition φ. Then, the selected environments serve as the input environment to evaluate an expression e. Similar the *let* expression, the *where* expression produces a new environment for e. The template for creating the new environment is provided in [9].

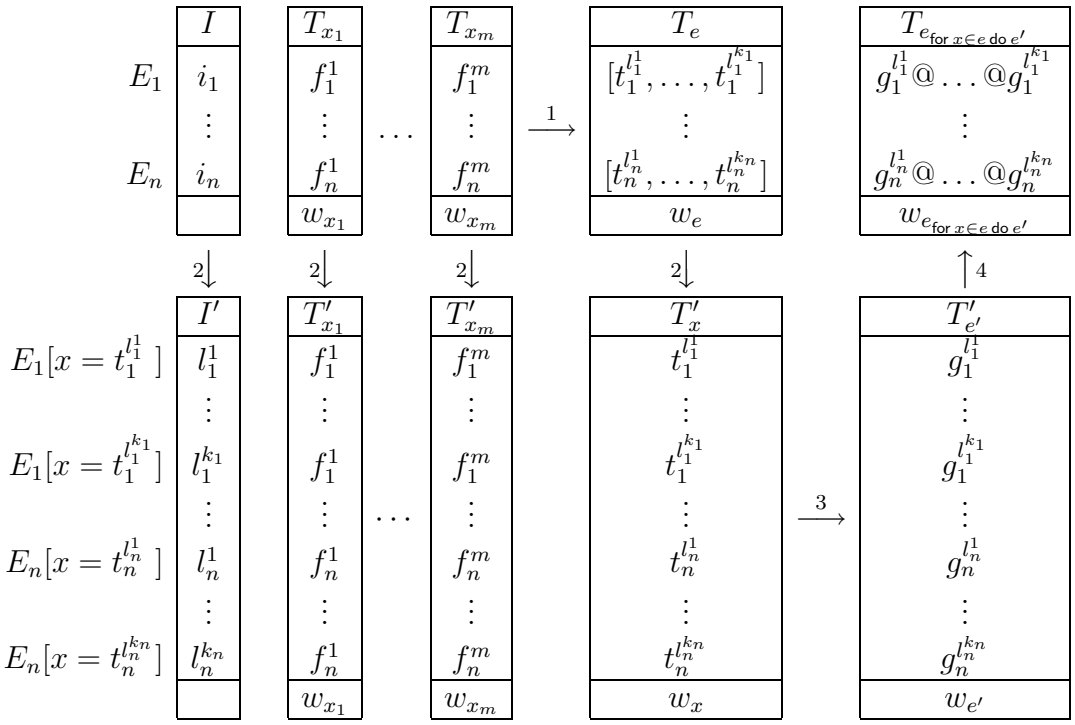


Figure 2.5: TRANSLATION OF "$\mathsf{for}\, x \in e\, \mathsf{do}\, e'$" IN THE ENVIRONMENT FOR $\{x_1, \ldots, x_m\}$.

FOR Expressions: $\mathsf{for}\, x \in e\, \mathsf{do}\, e'$

The *for* expression is an iteration expression that expands both the size and the number of the input environments by adding a series of bindings for the variable x with the values (XML forest) obtained from each of the input environments. The *for* expression iterates the bound variable x over each binding (XML tree) to evaluate expression e'. The number of the expanded environments is the total number of resulting trees obtained by evaluating e over a sequence of input environments.

The following paragraph shows the SQL templates for creating the new environment for the *for* expression, which are the revised versions that correct the imprecisions in the original translation templates in [9]:

```
CREATE VIEW I'(T_e) AS
  SELECT r.l AS i
  FROM   ROOTS(T_e) r

CREATE VIEW T'_x(T_e) AS
  SELECT x.s, x.l − i ∗ w_e + r.l ∗ w_e AS l,
              x.r − i ∗ w_e + r.l ∗ w_e AS r
  FROM   I, T_e x, ROOTS(T_e) r
  WHERE  i ∗ w_e ≤ r.l AND r.r < (i + 1) ∗ w_e
    AND  r.l ≤ x.l AND x.r ≤ r.r

CREATE VIEW T'_{x_i}(T_{x_i}, T_e) AS
  SELECT x.s, x.l − i ∗ w_{x_i} + r.l ∗ w_{x_i} AS l,
              x.r − i ∗ w_{x_i} + r.l ∗ w_{x_i} AS r
  FROM   I, T_{x_i} x, ROOTS(T_e) r
  WHERE  i ∗ w_e ≤ r.l AND r.r < (i + 1) ∗ w_e
    AND  i ∗ w_{x_i} ≤ x.l AND x.r < (i + 1) ∗ w_{x_i}
```

The result of the *for* expression is defined as follows:

```
CREATE VIEW T_{for x∈e do e'} AS SELECT * FROM T'_{e'}
```

The resulting width of the *for* expression is $w_{\mathsf{for}\,x\in e\,\mathsf{do}\,e'} = w_e w_{e'}$. The evaluation process of the *for* expression is illustrated in Figure 2.5, which is a revised figure originating from [9]. An example of the new environment created by evaluating the *for* expression $x \in e$ is shown in Example 2.4.2.

I'	$T'_{\$t}$		
i	s	l	r
2	title	90	93
	Data on the Web	91	92
16	title	720	723
	Advanced Programming in the Unix environment	721	722
30	title	1350	1353
	TCP/IP Illustrated	1351	1352

Figure 2.6: Each Title Forming a Separate Environment ($w_e = 44$)

Example 2.4.2 Continuing with Example 2.3.1, consider the following *for* expression:

`for` $t `in` *document*(′*reviews.xml*′)/*reviews*/*entry*/*title* `do` ...

Figure 2.6 shows a part of the expanded new environment after evaluating the *for* expression in the example. The width of document "*reviews.xml*" is set to 44, and the width of e is $w_e = 44$. □

Chapter 3

XQuery-to-SQL Query Processor

In this chapter we introduce a *XQuery-to-SQL Query Processor* we implemented to map the *Minimal XQuery* to the SQL language. Using *Dynamic Interval Encoding* technique [9], the query processor is able to handle a wide variety of XQuery expressions including arbitrarily-nested FLWR expressions, element constructors, path expressions and built-in functions. Given an input XQuery, the processor is able to generate a single SQL query, which can be executed in a traditional SQL query processor like DB2 to get the expected result from the relational encodings of XML data sources, and a physical plan, which can be executed in the *XQuery-enhanced Relational Engine* [15] to produce the XML-formatted query result.

3.1 Assumptions

There are several assumptions placed on the XML data sources and on the XQuery language that the query processor handles:

- The source XML documents that the processor deals with agree with the abstract syntax of *XML Forests* defined in Section 2.1, otherwise, the source documents should be transformed into the abstract XML forests using one of the approaches described in Section 2.1.
- The XML documents have been already mapped into the relational schema using the *Interval Encoding* technique introduced in Section 2.3.1.
- The input XQuery queries for the processor conforms to the syntax of the *Minimal XQuery* introduced in Section 2.3.2. Hence, an XQuery expression

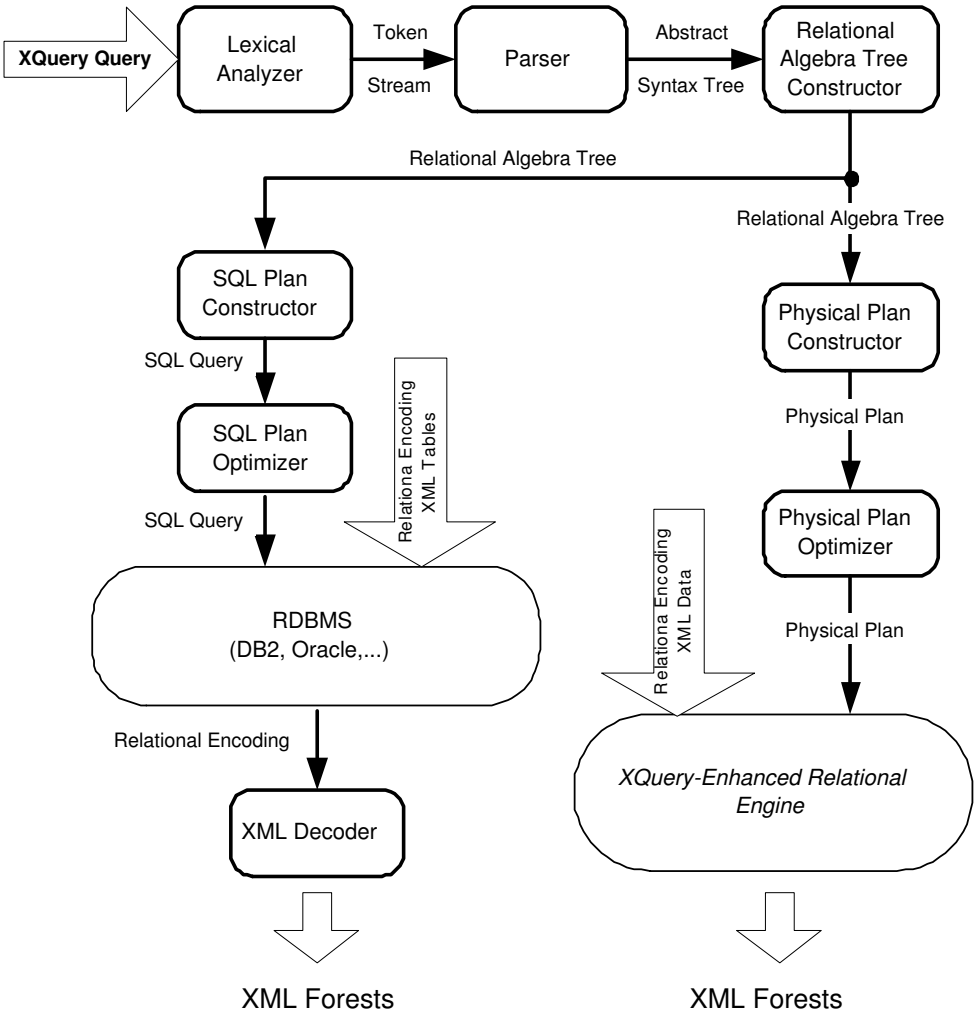


Figure 3.1: The Framework of XQuery-to-SQL Query Processor

might need to be moderately rewritten into the form of *Minimal XQuery* expression before it is input into the query processor.

For our examples we use the XML document "*auction.xml*" from the XMark Benchmark [1] and we assume that we always have an equivalent relational encoding table named "*auction*". The input XQuery is first rewritten in *Minimal XQuery* before it is input into the query processor. For example, the fragment "$\$b/id[1]$" in an input XQuery is substituted by an equivalent expression "$\mathsf{head}(\mathsf{select}('id', \mathsf{children}(\$b)))$".

3.2 Overview of the Framework

Figure 3.1 shows the framework of the *XQuery-to-SQL Query Processor*. The query processor acts as an XQuery-to-SQL compiler that translates XQuery language[1] into SQL language.

The input to the query processor is a *Minimal XQuery* expression. An analysis is performed by the compiler to complete the translation. The analysis includes the following steps:

- The lexical analysis – breaks the input query into individual words or "tokens" that are the units in the grammar of XQuery language.
- The syntax analysis – analyzes the phrase (expression) structure of XQuery language, and builds an abstract syntax tree.
- The relational algebra tree construction – determines the meaning of each expression and provides translation for each expression. A relational algebra tree, which is a tree consists of relational operations that are analogous to relational algebra operators, is generated as an intermediate result.
- The SQL query generation and optimization – traverses the relational algebra tree to construct a SQL query and optimizes the query to generate a more concise and efficient SQL query.
- The physical plan generation and optimization– traverses the relational algebra tree to generate a physical plan and provide query optimization for such a plan.

The processor produces two kinds of relational plans:

- The first one is a standard SQL query for commercial RDBMS like DB2, Oracle. Using the relational encoding tables as the source XML documents, the generated SQL query can be executed in a RDBMS to generate a relational encoding of the resulting XML forest. The relational encoding result is then converted into an XML forest by an *XML Decoder*, which transforms relational encodings back into XML forests.

[1]The *XQuery* query we mentioned here and in the following sections is referred to *Minimal XQuery*.

- The second is a relational physical plan which consists of relational operations [15] that are similar to relational algebra operators. The reason for generating such a physical plan for the *XQuery-enhanced Relational Engine* [15] is that the evaluation of the generated SQL query in standard RDBMS is slow due to a large number of *less-equal* comparison operations involved in the joins operations during the evaluation, and RDBMS can not take advantage of the ordering nature of the data. Based on this reason, the query processor generates the physical plan, where additional special efficient relational operators can be used in order to achieve better performance. Given the generated physical plan along with the relational encodings for the source XML documents, the relational engine [15] is able to generate the expected XML forest.

3.3 Lexical Analysis and Parsing

3.3.1 Lexical Analysis

Given a string representing an input XQuery, the lexical analyzer produces a stream of lexical tokens that include names, keywords, and punctuation marks. A lexical token is defined as a sequence of characters that appear as a unit in the grammar of a programming language [5]. Lexical tokens are classified into a finite set of token types. The token types for *Minimal XQuery* language can be found in Figure 3.2.

In query processor, the lexical analyzer is generated by JLex [6], an automatic lexical analyzer generator that produces a Java program from a lexical specification. A lexical specification is a file that contains a *regular expression* and an *action* for each token type in the language to be lexically analyzed. Detailed information about JLex can be found in [6].

3.3.2 Syntax Analysis

After the input XQuery query stream is broken into a sequence of lexical tokens by the lexical analyzer, the token stream is passed to the XQuery parser for syntax analysis. The grammar of XQuery language must be provided to perform the analysis.

The XQuery parser in the query processor is created using *CUP* [2] (*Construction of Useful Parser*), an automatic LALR parser-generator tool. A grammar

```
expr        ::= str
            | xfnExp
            | LET str := expr IN expr
            | WHERE cond RETURN expr
            | FOR str IN expr DO expr
            | ( expr )

cond        ::= EMPTY ( expr )
            | NOT EMPTY ( expr )
            | expr cop expr
            | ( cond )

xfnExp      ::= []()
            | DOCUMENT( 'str' )
            | XNODE( str, expr )
            | expr @ expr
            | SELECT( 'str', expr )
            | unixfnExp ( expr )

unixfnExp   ::= HEAD | LAST | TAIL | REVERSE | DISTINCT | SORT
            | ROOTS | SUBTREESDFS | CHILDREN | COUNT

str         ::= ([0-9]|[A-Za-z]|$|_|-|.|:|%|#|&|*|!|\|/)*

cop         ::= < | <= | = | > | >=
```

Figure 3.2: The Grammar for Minimal XQuery

specification that describes the grammar rules is provided for *CUP* to generate the parser. Figure 3.2 shows a simplified grammar for *Minimal XQuery* language, which captures the essential features of the language. In the implementation, we design a more complex grammar which supports a larger set of the expression formats. For example, the keywords in the input query are case insensitive to the query processor. For the element constructor, the query processor supports the expressions in both formats of xnode($'s'$, $expr$) and <s>XF</s>.

```
abstract class ExpTree {                                        // XF
    TraverseParseTree generateIterator ()
}
ExpVar (String id)                                             // var
abstract class ExpXFn extends ExpTree                          // XFn
ExpLet (String id, ExpTree e1, ExpTree e2)                  //let clause
ExpWhere (ExpBoolean b, ExpTree e)                        //where clause
ExpFor (String id, ExpTree e1, ExpTree e2)                  //for clause

abstract class ExpBoolean {
    TraverseBooleanParseTree generateIterator ()
}
ExpBoolEmpty (ExpTree e)                             //where empty(XF)
ExpBoolNotEmpty (ExpTree e)                       //where not empty(XF)
ExpBoolComp (ExpTree e1, ExpTree e2)  //where XF1 =(<,>,<=,>=) XF2

abstract class ExpXFn
ExpXFnEmptyConst ()                                 //empty constructor
ExpXFnDocument (String fn)                           //document('file')
ExpXFnConcatConst (ExpTree e1, ExpTree e2)                    //XF @ XF
ExpXFnXnodeConst (String s, ExpTree e)              //xnode('root',XF)
ExpXFnHead (ExpTree e)                                      //head(XF)
ExpXFnLast (ExpTree e)                                      //last(XF)
ExpXFnTail (ExpTree e)                                      //tail(XF)
ExpXFnSort (ExpTree e)                                      //sort(XF)
ExpXFnReverse (ExpTree e)                                 //reverse(XF)
ExpXFnDistinct (ExpTree e)                               //distinct(XF)
ExpXFnSelect (String s, ExpTree e)                    //select('label',XF)
ExpXFnRoots (ExpTree e)                                     //roots(XF)
ExpXFnChildren (ExpTree e)                               //children(XF)
ExpXFnSubtreedfs (ExpTree e)                           //subtreesdfs(XF)
ExpXFnCount (ExpTree e)                                     //count(XF)
```

Figure 3.3: Data Structure of Abstract Syntax Trees for XQuery Language

3.3.3 The Abstract Syntax Tree

The XQuery parser does more than just recognizes the XQuery expressions. It also produces an abstract syntax tree, which is a data structure that latter phases of the compiler traverses. This kind of parse tree carries the parse structure of XQuery language, with all parsing issues resolved. The SQL translation is later performed by traversing the generated abstract syntax tree.

Java classes are designed for the *abstract-syntax-tree* data structure. Figure 3.3

lists the constructors of the java class. The object field variables of the classes correspond to the variables within the constructor arguments. This kind of data structure design separates the syntax analysis from the semantic interpretation. For each form of XQuery expression an abstract syntax class is designed. A *traverser*, which is a traverse method that performs the corresponding SQL translation for the expressions, is also generated within each abstract syntax class for latter interpretations. Example 3.3.1 illustrates the abstract syntax class of the *for* expression.

Example 3.3.1 The abstract syntax class of the *for* expression is as follows:

```
class ExpFor extends ExpTree {
    public String var;
    public ExpTree exp1, exp2;
    public ExpFor (String id, ExpTree e1, ExpTree e2) {
        var = id; exp1 = e1; exp2 = e2;
    }

    public TraverseParseTree generateIterator () {
        return (new TraverseFor () );
    }
}
```

In the abstract syntax class, the constructor $ExpFor$ constructs syntax trees for the *for* expressions. A traverser $TraverseFor$ is also provided to traverse the generated syntax tree and perform the corresponding SQL translation for the *for* expressions. □

For example, for the following XQuery expression

```
for $b in document('bib')//book do <books> $b </books>
```

the corresponding abstract parse tree is shown in Figure 3.4.

3.4 Relational Algebra Tree Construction

The XQuery parser generates an abstract syntax tree along with a corresponding traverser for each node in the parse tree. Each node in the parse tree corresponds to

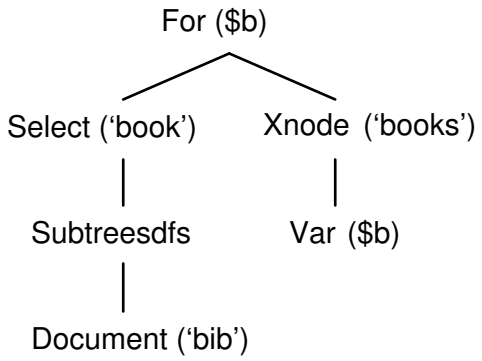


Figure 3.4: Sample Abstract Syntax Tree

an XQuery expression in the input query. For each kind of the XQuery expression, there is a corresponding SQL translation template.

In the *Relational Algebra Tree Construction* phase, the abstract syntax tree is traversed and the XQuery-to-SQL transformation is performed. The traversers traverse the parse tree and generate a *relational algebra tree*. For each form of XQuery expression, there is a corresponding *relational algebra tree*, which is an alternative representation for the corresponding SQL translation template.

Section 3.4.1 gives a brief introduction for the *relational operations* that are used to generate the relational algebra tree. Section 3.4.2 introduces the SQL translation templates for XQuery expressions and the corresponding relational algebra trees that map the SQL templates.

3.4.1 Relational Operations

For translation purposes, *relational operations* are used to construct relational algebra trees according to the SQL translation templates. The *relational operations* are similar to relational algebra operators that perform the SQL operations. The results after performing the relational operations are exactly the same as those of the corresponding SQL operations. The set of relational operations we use in the XQuery translation are implemented in *XQuery-enhanced Relational Engine* [15] and are sufficient to handle the *Minimal XQuery*.

We use the notation of Operation$[parameters]$(Operation$*$) to represent a relation operation that has an input parameter list, *parameters*, which can be arithmetic expressions or boolean expressions, and a list of input relational operations,

Operation$*$, which provide input tuples. The possible operands in the expressions in *parameters* include: constants, the incoming tuples from the input relational operations, and the *open parameters* that consist of the tuples from the first input operations of the *nested-loop* join operations in the ancestor nodes for the current relational operation in a relational algebra tree[2].

1 The Project Operation: Project[*attributlist*](*operation*)

This operation is equivalent to the duplicate-preserving projection operation in relational algebra. The parameter *attributlist* specifies the attributes that are retained by the operation from the input tuples. For example, if we want to preserve the values of the first and the third attributes from the input tuples, *attributlist* is then set to the form like "1, 3".

2 The AddColumn Operation: AddColumn[*expression*](*operation*)

This operation appends a new attribute to the input tuples. The values for the new attribute are assigned by the computation results of the expression *expression*. The *expression* can be an arithmetic expression whose operands can be either constant natural numbers or the values of the attributes from the input tuples.

3 The Select Operation: Select[*condition*](*operation*)

The *Select* operation is used to filter out the unwanted tuples from the input operation by evaluating the boolean expression *condition*. It implements the SQL WHERE clause with a single boolean condition.

4 The LoopJoin Operation: LoopJoin[*conditions*](*operation*1, *operation*2)

This operation implements the *nested-loop* join. If no boolean condition *conditions* is given, the result is the *Cartesian product* of *operation*1 and *operation*2; otherwise, the boolean expressions filter out the unwanted results of the *Cartesian product*. An example of *conditions* is "1 = 1", which means that an eligible tuples pair is the one that the 1st attribute value of the tuple from *operation*1 should equal to the 1st attribute value of that from *operation*2.

[2]In *nested-loop* join operations, the resulting tuples from their first input operations are iteratively passed to the second input operations as *open parameters*. Details about relational operations can be found in [15].

In this operation, the resulting tuples from its first operation *operation*1 are iteratively passed to the second operation *operation*2 as part of the *open parameters* for *operation*2. Let $\{\%1, \ldots, \%n\}$ be the *open parameters* for *operation*1 and let $(C1, \ldots, Cm)$ be the resulting tuples of *operation*1. The *open parameters* for *operation*2 are constructed by appending the resulting tuples from *operation*1 to the *open parameters* for *operation*1. Thus, the *open parameters* for *operation*2 are represented as $\{\%1, \ldots, \%n, \%(n+1), \ldots, \%(n+m)\}$ and can be used as operands for the expressions in *operation*2.

5 The MergeJoin Operation: MergeJoin[*conditions*](*operation*1, *operation*2)

This operator implements the *merge join* in SQL. The input operations for this operation are assumed to be properly sorted already to support the join condition *conditions*.

6 The LoopIn Operation: LoopIn[*conditions*](*op*1, *op*2)

This operator preserves the tuples from *op*1 whenever there exists a matching tuple from *op*2 that satisfies the *LoopIn* conditions. Similar to the *LoopJoin* operation, it uses the *nested-loop* join technique to evaluate the tuples from the inputs, and the tuples and *open parameters* from *op*1 are passed to *op*2 as its *open parameters*.

7 The LoopExceptIn Operation: LoopExceptIn[*conditions*](*op*1, *op*2)

In contrast to the *LoopIn* operation, this operator returns the tuples from *op*1 whenever there does not exist any matching tuple from *op*2 that satisfies the *LoopExceptIn* conditions. Similar to the *LoopJoin* operation, it uses the *nested-loop* join technique to evaluate the tuples from the inputs, and the tuples and *open parameters* from *op*1 are passed to *op*2 as its *open parameters*.

8 The MergeExceptIn Operation: MergeExceptIn[*conditions*](*op*1, *op*2)

Similar to *LoopExceptIn* operator, this operator implements the *set difference* in relational algebra by using the *merge join* technique. The input operations are assumed to be properly ordered to support the condition, *conditions*.

9 The CatUnion Operation: CatUnion[ ](*operation*1, *operation*2)

This operation is a union operation in relational algebra based on the concatenation of *operation*1 and *operation*2. The result preserves duplicates.

10 The MergeUnion Operation: MergeUnion[$SortOrder$]($op1, op2$)

This operator performs a union operation based on merging ordered input. The operation assumes that the input operations are properly ordered.

11 The Sort Operation: Sort[$SortOrder$]($operation$)

The operator sorts the tuples generated by the input operation based on the given sort order parameters. For example, a sort order "$A1$" means sorting the tuples from *operation* in the ascending order of the values of its 1st attribute.

12 The CountAggregate Operation: CountAggregate[$groupbylist$]($operation$)

The *CountAggregate* operation is a grouping operation. The parameter *groupbylist* specifies a list of attributes that are used in the grouping operation. An additional attribute with the values of $count(*)$ is added to the input tuples as the result. The operation assumes the input values have been sorted by the grouping attributes.

13 The XMLFileReader Operation: XMLFileReader[$XMLfileName$]()

This operation reads an XML file with the given file name in the parameter *XMLfileName* into a relational encoding.

14 The TextFileReader Operation: TextFileReader[$TextFileName$]()

Similar to the *XMLFileReader* operation, this operation provides access to a relational encoding with the given file name in the parameter *TextFileName*.

15 The UNIT Operation: UNIT[ ]()

This operator represents the relational table named "UNIT", which always contains only one tuple.

16 The EMPTY Operation: EMPTY[ ]()

This operator represents the relational table named "`EMPTY`", which contains no tuples.

17 The ScalarProject Operation: ScalarProject[](*operation*)

This operation is used to implement the SQL *scalar fulselect* expression. It is similar to the *Project* operation except that the *ScalarProject* operation only allows its input operation return one and only one resulting tuple, which contains all the attributes, as its output tuple, otherwise, the *ScalarProject* operation reports error and terminates the execution.

Constructors:

document($'filename'$)	the relational encoding for an XML document
[]()	the empty forest constructor
xnode($'label', e$)	the element constructor, adds a labelled root to a forest
e_1 @ e_2	the concatenation operator

Condition Operations:

empty(e)	the test for emptiness
not empty(e)	the test for not emptiness
$e_1 = e_2$	the equal condition, the test for label equality
$e_1 < e_2$	the less condition, the test for label ordering
$e_1 > e_2$	the greater condition, the test for label ordering
$e_1 \leq e_2$	the less and equal condition, the test for label ordering
$e_1 \geq e_2$	the greater and equal condition, the test for label ordering

Other Operations:

head(e)	the first tree of a forest
tail(e)	all but the first tree of a forest
last(e)	the last tree of a forest
reverse(e)	the forest in reverse order (top-level only)
select($'str', e$)	the subforest of trees with root labels are the 1st arg value
distinct(e)	the subforest of distinct trees (1st preserved)
count(e)	the number of the trees of a forest
roots(e)	the forest of root nodes
children(e)	the forest of all children in original order
subtreesdfs(e)	the forest of all subtrees in DFS order

Figure 3.5: The Implemented XQuery Basic Operations

```
abstract class PPhysicalPlan {
    //Traverser for generating the relational-operation-query plan
    PTraversePhyPlan generateIterator()

    //Traverser for generating the SQL query
    public PTraverseSQL printSQL()
}
PAddColumn (String SQLpara, PPhysicalPlan Te)           //AddColumn operator
PProject   (String SQLpara, PPhysicalPlan Te)             //Project operator
PSelect    (String SQLpara, PPhysicalPlan Te)              //Select operator
PLoopJoin  (String SQLpara, PPhysicalPlan Te1,
            PPhysicalPlan Te2)                           //LoopJoin operator
PMergeJoin (String SQLpara, PPhysicalPlan Te1,
            PPhysicalPlan Te2)                             //Merge operator
PLoopIn    (String SQLpara, PPhysicalPlan Te1,
            PPhysicalPlan Te2)                             //LoopIn operator
PLoopExceptIn  (String SQLpara, PPhysicalPlan Te1,
                PPhysicalPlan Te2)                    //LoopExceptIn operator
PMergeExceptIn (String SQLpara, PPhysicalPlan Te1,
                PPhysicalPlan Te2)                   //MergeExceptIn operator
PCatUnion   (PPhysicalPlan Te1, PPhysicalPlan Te2)       //CatUnion operator
PMergeUnion (PPhysicalPlan Te1, PPhysicalPlan Te2)     //MergeUnion operator
PSort (String SQLpara, PPhysicalPlan Te)                     //Sort operator
PTextFileReader (String SQLpara)                    //TextFileReader operator
PXMLFileReader (String SQLpara)                      //XMLFileReader operator
PUNIT ()                                                      //UNIT operator
PEMPTY ()                                                    //EMPTY operator
PScalarProject (PPhysicalPlan Te)                    //ScalarProject operator
PCountAggregate (PPhysicalPlan Te)                  //CountAggregate operator
```

Figure 3.6: Data Structure for Relational Algebra Trees

3.4.2 Mapping SQL Templates to Relational Algebra Trees

The XQuery query processor is able to handle a variety of XQuery expressions. Figure 3.5 shows a summary of the implemented XQuery basic operations. For each form of the XQuery expression, there is a corresponding SQL translation template similar to that introduced in [9]. Instead of directly using the SQL templates to generate the final SQL query, *relational algebra trees* that map the SQL templates are composed together to construct a relational algebra tree as an intermediate result for the SQL query generation. The generated relational algebra tree is analogous to the relational algebra representation for the final SQL query. The use of such intermediate result facilitates the query optimization and the query generation

for physical plans in latter phases of the translation.

In the following sections, we first give a brief introduction to the data structures for relational algebra trees; then, we provide detailed SQL translations for XQuery FLWR expressions and several examples of SQL translations for the basic operations[3].

Java classes are designed for constructing the *relational-algebra-tree* data structure. For each type of relational operation, there is a corresponding java class. Traverse methods, which are used to traverse the relational algebra tree in latter phases, are also constructed within the class. Figure 3.6 shows the constructors of the java classes for relational operations. The object field variables of the classes correspond exactly to the variables within the constructor arguments.

In relational operations, since no attribute names are provided for the input/resulting tuples, the values from the tuples are manipulated based on their attribute positions in the tuples. To facilitate the translation, attributes of resulting tuples from any relational operation are named $C1, \ldots, Cn$ according to their left-to-right positions in the tuples. The attribute names are used in the arithmetic/boolean expressions of relational operations[4]. Example 3.4.1 illustrates the process of mapping a SQL fragment into a relational algebra tree.

Example 3.4.1 The following SQL fragment is frequently used in the XQuery-to-SQL translation to identify the environments that the values of input table parameters belong to:

```
( SELECT i, s, l − i ∗ w_e AS l, r − i ∗ w_e AS r
  FROM   I, T_e
  WHERE  i ∗ w_e ≤ l  AND r < (i + 1) ∗ w_e  )  T_e_i
```

The corresponding relational algebra tree is shown in Figure 3.7. To make it easier to understand, the attribute names i, s, l, r are used in the figure instead of using the attribute names $C1, \ldots, Cn$. The matching relational algebra tree can be represented in the following form:

[3] A comprehensive translation for all of the XQuery basic operations are provided in Appendix A.

[4] In the relational algebra tree introduced in the following paragraph, not all of the relational operations use $C1, \ldots, Cn$ to represent the attribute names for the input tuples. For example, the *Project* and *LoopJoin* operations use position numbers to represent attributes.

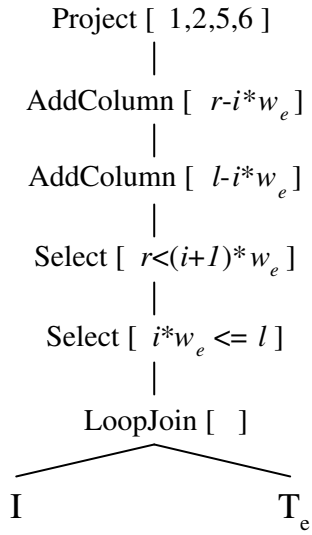


Figure 3.7: Sample Relational Algebra Tree

$$
\begin{aligned}
&T_{e\_i}(T_e) = \\
&\quad \texttt{Project}[1,2,5,6](\\
&\qquad \texttt{AddColumn}[C4 - C1 * w_e](\\
&\qquad\quad \texttt{AddColumn}[C3 - C1 * w_e](\\
&\qquad\qquad \texttt{Select}[C4 < (C1 + 1) * w_e](\\
&\qquad\qquad\quad \texttt{Select}[C1 * w_e \leq C3](\\
&\qquad\qquad\qquad \texttt{LoopJoin}[\,](I, T_e))))))
\end{aligned}
$$

where w_e is the width of the relational representation of T_e, which is computed using the width function introduced in [9]. □

I. The Translations for Basic Operations

Based on the SQL translation template for XQuery basic operations (see Section 2.4.2), there are two kinds of SQL fragments that appear in most of the SQL translation templates for the XQuery basic operations. The first kind of SQL fragment is the fragment introduced in Example 3.4.1. The notation $T_{e\_i}(T_e)$ is used to represent the relational algebra tree for the SQL fragment from Example 3.4.1. T_e is the input table for the SQL fragment.

The second SQL fragment that appears in all SQL translation templates for XQuery basic operations except for the *empty constructor* is as follows:

```
CREATE VIEW T_XFn(T_e1 ... T_em) AS
  SELECT s, l + i * w_XFn AS l, r + i * w_XFn AS r
  FROM   (
           ⋮
         ) Q_XFn
```

In the following translation, we only provide the relational algebra tree for the SQL fragment Q_{XFn} in the SQL templates for basic operations. Q_{XFn} always returns tuples (i, s, l, r). Given the mapping relational algebra tree $T_{Q_{\mathsf{XFn}}}$ (with resulting tuples $(C1, C2, C3, C4)$) for the SQL fragment Q_{XFn}, the final relational algebra tree for the SQL template is as follows:

$$
\begin{aligned}
&T_{\mathsf{XFn}}(T_{Q_{\mathsf{XFn}}}) = \\
&\quad \mathtt{Project}[2,5,6](\\
&\qquad \mathtt{AddColumn}[C4 + C1 * w_{\mathsf{XFn}}](\\
&\qquad\quad \mathtt{AddColumn}[C3 + C1 * w_{\mathsf{XFn}}](T_{Q_{\mathsf{XFn}}})))
\end{aligned}
$$

1. The DOCUMENT operator: $\mathsf{document}('filename')$

This operation creates a relational encoding for the input XML document based on the input environment. We assume there already exist initial relational encoding tables for the input XML documents in the database systems.

1) The SQL Translation Template

The SQL translation template for generating the relational encoding of document "$filename$" is shown as follows:

```
CREATE VIEW T_document(T_filename) AS
  SELECT s, l + i * w_document AS l, r + i * w_document AS r
  FROM   I, T_filename
```

The width of the document, w_{document}, can be set to any natural number that greater than the maximum right endpoint value $r_{\mathtt{max}}$ in the initial relational encoding of

the document. Here, we choose a value $w_{\mathsf{document}} = r_{\mathtt{max}} + 1$.

2) The Relational Algebra Tree

If the document is specified as an XML file with a suffix "*.xml*" in the document name, the source XML file is first read into a relational encoding using the relational operation *XMLFileReader*. Thus, we have the following relational algebra tree mapping for the SQL fragment Q_{XFn} in the SQL template:

$Q_{\mathsf{XFn}} =$
`LoopJoin[ ]`$(I,$
`XMLFileReader`$[filename]()$

Otherwise, if the source document is already in a relation, we have the following alternative relational algebra tree for the SQL fragment Q_{XFn}:

$Q_{\mathsf{XFn}} =$
`LoopJoin[ ]`$(I,$
`TextFileReader`$[filename]()$

2. The Empty Constructor: [ ]()

This operator constructs an empty forest. It is often used to construct a new XML forest. We assume there is an empty relation table named "EMPTY" in the database systems. The EMPTY table is used to construct an empty forest that has the same relational encoding schema (s, l, r) (see Section 2.3.1) as that for the XML documents.

1) The SQL Translation Template

Since the *empty* constructor constructs an empty XML forest, the result does not depend on the input environment. Hence, we have the following SQL translation template:

```
CREATE VIEW T_empty  AS
  SELECT 's' AS s, 0 AS l, 1 AS r
  FROM   EMPTY
```

The width of the result is set to $w_{\texttt{empty}} = 0$ since no tuples are returned.

2) The Relational Algebra Tree

The mapping relational algebra tree for the SQL template is as follows:

$T_{empty} =$
AddColumn[1](
AddColumn[0](
AddColumn[$'s'$](
EMPTY[ ]())))

3. The Element Constructor: $\mathsf{xnode}('label', e)$

The element constructor is used to construct a new XML forest. The result of the operation is to add a root node with the given label name, e.g. "*label*", to the resulting XML forest of the subexpression e.

1) The SQL Translation Template

The SQL translation template for the operation is shown as follows:

```
CREATE VIEW T_xnode('label', T_e) AS
 SELECT s, l + i * w_xnode AS l, r + i * w_xnode AS r
 FROM  (
         ( SELECT i,'label' AS s, 0 AS l, w_e + 1 AS r
           FROM I, UNIT )
         UNION ALL
         ( SELECT i, s, l + 1 AS l, r + 1 AS r
           FROM
             ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
               FROM   I, T_e
               WHERE  i * w_e ≤ l  AND r < (i + 1) * w_e  )  T'_i
         )
       )  Q_XFn
```

Since the *xnode* operation is to add one root node to the input XML forest T_e, the width of the result of the operation is $w_{\mathsf{xnode}} = w_e + 2$.

2) The Relational Algebra Tree

The corresponding relational algebra tree for the fragment Q_{XFn} in the SQL template is as follows:

$T_{Q_{\mathsf{XFn}}} =$

```
CatUnion[ ](
   AddColumn[w_e + 1](
      AddColumn[0](
         AddColumn['label'](
            LoopJoin[ ](I,
               UNIT[ ]()))))
   Project[1, 2, 5, 6](
      AddColumn[C4 + 1](
         AddColumn[C3 + 1](T_e_i(T_e))))
)
```

4. The ROOTS Operator: $\mathsf{roots}(e)$

The *roots* operation returns the root nodes of the XML forest T_e (with width w_e) of the subexpression e.

1) The SQL Translation Template

The SQL translation template for the operation is shown as follows:

```
CREATE VIEW T_roots(T_e) AS
  SELECT s, l + i * w_roots AS l, r + i * w_roots AS r
  FROM
      ( SELECT u.i, u.s, u.l, u.r
        FROM
          ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u
        WHERE NOT EXISTS (
          SELECT *
          FROM
              ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                FROM I, Te
                WHERE i * w_e ≤ l AND r < (i + 1) * w_e
              ) v
          WHERE v.l < u.l AND u.r < v.r AND u.i = v.i )
  ) Q_XFn
```

The resulting width is set to $w_{\mathsf{roots}} = w_e$. Unlike other basic operations, given an input XML forest, the result of a *roots* operation does not depend on the input environments that the XML trees belong to. Hence, we have the following simplified SQL translation template for the *roots* operation.

```
CREATE VIEW T_roots(T_e) AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM   T_e u
  WHERE NOT EXISTS (
    SELECT *
    FROM   T_e v
    WHERE  v.l < u.l AND u.r < v.r )
```

For the same reason, the *children* operation also has two variants of SQL translation templates introduced in Appendix A.2.13.

2) The Relational Algebra Tree

For the first SQL template for the *roots* operation, the corresponding relational algebra tree for the fragment Q_{XFn} in the template is as follows:

$$T_{Q_{\texttt{XFn}}} = \texttt{LoopExceptIn}[C1 = C1, C3 > C3, C4 < C4](T_e\_i, T_e\_i)$$

For the simplified SQL translation template, the corresponding relational algebra tree for the whole template is shown as follows:

$$T_{\texttt{roots}} = \texttt{LoopExceptIn}[C2 > C2, C3 < C3](T_e, T_e)$$

II. The Translations for FLWR expressions

1. The FOR Expression: for $x \in e$ do e'

The *for* expression is an iteration expression that manipulates a sequence of input environments by adding a series of bindings of the variable x. As a result, a new environment, obtained by expanding both the size and the total number of the original input environment, is created for the evaluation of the expression e'. The new environment include the new environment index set I', the new representations $T'_{x_1}, \ldots, T'_{x_m}$ for the bound local variables from the input environment, and the relational representation T'_x of the bindings of the new variable x.

The SQL translation templates for the new environment $I', T'_{x_1}, \ldots, T'_{x_m}, T'_x$ are exactly the same as the ones introduced in Section 2.4.2. Let $Roots\_T_e$ be the relational algebra tree for the *roots* operation $\mathsf{roots}(T_e)$. The relational algebra tree for I':

$$I' = \texttt{Project}[2](Roots\_T_e)$$

The relational algebra tree for the new representation T'_{x_i} of the bound variable x_i from the input environment is as follows:

$$
\begin{array}{l}
T'_{x_i} = \\
\quad \texttt{Project}[2,8,9](\\
\quad\quad \texttt{AddColumn}[C4 - C1 * w_{x_i} + C6 * w_{x_i}](\\
\quad\quad\quad \texttt{AddColumn}[C3 - C1 * w_{x_i} + C6 * w_{x_i}](\\
\quad\quad\quad\quad \texttt{Select}[C4 < (C1+1) * w_{x_i}](\\
\quad\quad\quad\quad\quad \texttt{Select}[C1 * w_{x_i} \leq C3](\\
\quad\quad\quad\quad\quad\quad \texttt{Select}[C7 < (C1+1) * w_e](\\
\quad\quad\quad\quad\quad\quad\quad \texttt{Select}[C1 * w_e \leq C6](\\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{LoopJoin}[\,](\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{LoopJoin}[\,](I, T_{x_i}), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Roots\_T_e)))))))
\end{array}
$$

The relational algebra tree for the representation T'_x of the new bound variable x is shown as follows:

$$
\begin{array}{l}
T'_x = \\
\quad \texttt{Project}[2,8,9](\\
\quad\quad \texttt{AddColumn}[C4 - C1 * w_e + C6 * w_e](\\
\quad\quad\quad \texttt{AddColumn}[C3 - C1 * w_e + C6 * w_e](\\
\quad\quad\quad\quad \texttt{Select}[C4 \leq C7](\\
\quad\quad\quad\quad\quad \texttt{Select}[C6 \leq C3](\\
\quad\quad\quad\quad\quad\quad \texttt{Select}[C7 < (C1+1) * w_e](\\
\quad\quad\quad\quad\quad\quad\quad \texttt{Select}[C1 * w_e \leq C6](\\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{LoopJoin}[\,](\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{LoopJoin}[\,](I, T_e), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Roots\_T_e)))))))
\end{array}
$$

2. The LET expression: $\mathsf{let}\, x = e\, \mathsf{in}\, e'$

Similar to the *for* expression, the *let* expression operates on the input environment by adding a new table T'_x representing bindings for the variable x. However, different from the *for* expression, the *let* expression adds just one binding for variable x instead of a series of bindings. The SQL templates for the new environment $I', T'_{x_1}, \ldots, T'_{x_m}, T'_x$ are defined in the same way as those introduced in [9]. The relational algebra trees for the corresponding SQL templates are as follows:

$$
\begin{aligned}
I' &= \texttt{Project}[1](I) \\
T'_{x_i} &= \texttt{Project}[1,2,3](T_{x_i}) \\
T'_x &= \texttt{Project}[1,2,3](T'_x)
\end{aligned}
$$

3. The WHERE expressions: where φ return e

The *where* expression extracts the desired environments within a sequence of input environments using its condition operation φ. As a result, it produces a new environment $I', T'_{x_1}, \ldots, T'_{x_m}$ for the subexpression e.

We implement seven kinds of boolean operations (φ) for *where* expressions: the *empty*, *not empty*, *equal* ($=$), *less* ($<$), *greater* ($>$), *lessequal* ($<=$), and *greaterequal* ($>=$). The SQL translations of I' for various *where* expressions depend on the boolean conditions and are introduced in the following paragraphs.

I. SQL Templates for the New Environment Index Set I'

1) The EMPTY Operation: where empty(e) return e'

The filter condition "empty(e)" selects those environments which provide no results for the expression e. The SQL template for the new environment index set I' is as follows:

```
CREATE VIEW I'(T_e) AS
  SELECT i
  FROM   I
  WHERE NOT EXISTS  (
    SELECT *
    FROM   T_e
    WHERE  i * w_e ≤ l  AND  r < (i + 1) * w_e )
```

The corresponding relational algebra tree is shown as follows:

$$
I'_{\texttt{empty}} = \texttt{LoopExceptIn}[C1 * w_e \leq C2, (C1+1) * w_e > C3](I, T_e)
$$

If the *LoopExceptIn* operation only handles atomic equality conditions, then *AddColumn* operations are added to the above relational algebra tree to handle the arithmetic computations occur in the above *LoopExceptIn* condition.

2) The NOT EMPTY Operation: where not empty(e) return e'

In contrast to the *empty* operation, this condition operation filters out the environments in which no result is returned after evaluating the expression e. The SQL template for the new environment index set I' is as follows:

```
CREATE VIEW I'(T_e) AS
  SELECT i
  FROM   I
  WHERE EXISTS  (
    SELECT *
    FROM   T_e
    WHERE  i * w_e ≤ l  AND  r < (i + 1) * w_e  )
```

The corresponding relational algebra tree is shown as follows:

$$I'_{\texttt{empty}} = \texttt{LoopIn}[C1 * w_e \leq C2, (C1 + 1) * w_e > C3](I, T_e)$$

Similar to the translation of *empty* operation, if the *LoopIn* operation only handles atomic equality conditions, then *AddColumn* operations are added to the above relational algebra tree to handle the arithmetic computations occur in the above *LoopIn* condition.

3) The EQUAL Operation: where $e_1 = e_2$ return e'

We only consider *label equality* for the *equal* operation in the boolean conditions. This operation extracts environments that satisfy the label equality comparison condition "$e_1 = e_2$". The condition is *true* only when the label values in the results of e_1 and e_2 are unique and equal. For an index value i in the input environment index set I, there should exist one and only one resulting tuple for each of the expressions e_1 and e_2. Otherwise, an error should be reported and the execution process for the input XQuery should be terminated. The SQL *scalar fulselect* expression can properly handle such situation. *Scalar fulselect* expression is used in the following SQL template to construct the new environment index set I':

```
CREATE VIEW I'(T_e1, T_e2) AS
  SELECT i
  FROM   I
  WHERE
    ( SELECT s FROM T_e1 WHERE i * w_e1 ≤ l  AND r < (i + 1) * w_e1  )
    =
    ( SELECT s FROM T_e2 WHERE i * w_e2 ≤ l  AND r < (i + 1) * w_e2  )
```

The above SQL template is specially designed for generating the SQL query by the query processor. In order to map the above SQL template into a relational algebra tree, the template need to be rewritten in an alternative way shown as follows:

```
CREATE VIEW I'(T_e1, T_e2) AS
 SELECT i
 FROM
   ( SELECT i,
       ( SELECT s FROM T_e1 WHERE i * w_e1 ≤ l  AND r < (i + 1) * w_e1 ) AS s_1,
       ( SELECT s FROM T_e2 WHERE i * w_e2 ≤ l  AND r < (i + 1) * w_e2 ) AS s_2
     FROM  I
     WHERE s1 = s2  )  IT
```

The corresponding relational algebra tree is as follows:

$$
\begin{aligned}
&I'_{\texttt{equal}} = \\
&\quad \texttt{Project}[1](\\
&\qquad \texttt{LoopJoin}[2=1](\\
&\qquad\quad \texttt{LoopJoin}[\,](I,\\
&\qquad\qquad \texttt{Project}[1](\\
&\qquad\qquad\quad \texttt{ScalarProject}[](\\
&\qquad\qquad\qquad \texttt{Select}[\%n * w_{e_1} \leq C2](\\
&\qquad\qquad\qquad\quad \texttt{Select}[C3 < (\%n+1) * w_{e_1}](T_{e_1})))),\\
&\qquad\quad \texttt{Project}[1](\\
&\qquad\qquad \texttt{ScalarProject}[](\\
&\qquad\qquad\quad \texttt{Select}[\%n * w_{e_2} \leq C2](\\
&\qquad\qquad\qquad \texttt{Select}[C3 < (\%n+1) * w_{e_2}](T_{e_2})))))
\end{aligned}
$$

where "$\%n$" in the relational *Select* operations represents the last *open parameter* in the *open parameters* list for the first *Select* operation, $\texttt{Select}[\%n * w_{e_1} \leq C2](...)$,

in the above relational algebra tree. $\%n$ contains the values from the environment index set I.

The SQL translations of I' for the other *where* expressions (with comparison conditions like *less* (<), *greater* (>), *lessequal* (<=), *greaterequal* (>=)) are analogous to that of the *equal* operation and are omitted in the thesis.

II. SQL Template for the New Representations of Variables

The SQL templates for creating the new relational representations for the bound variables for various *where* expressions share the same template shown as follows:

```
CREATE VIEW T'_{x_i}(T_{x_i}) AS
  SELECT s, l, r
  FROM   T_{x_i}, I'
  WHERE  i * w_{x_i} ≤ l  AND  r < (i + 1) * w_{x_i}
```

The result of *where* expression is defined as:

```
CREATE VIEW T_{where φ return e}  AS  SELECT * FROM T'_e
```

The width of *where* expression is set to $w_{\mathsf{where}\,\varphi\,\mathsf{return}\,e} = w_e$. The relational algebra tree for T'_{x_i} is as follows:

$T'_{x_i} =$

```
    Project[2,3,4](
      Select[C1 * w_e ≤ C3](
        Select[C4 < (C1 + 1) * w_{x_i}](
          LoopJoin[ ](I, T_{x_i}))))
```

3.5 SQL Query Generation

The relational algebra tree serves as an intermediate result for generating a final SQL query for RDBMS and a physical plan for the *XQuery-enhanced Relational Engine* [15]. Section 3.5.1 describes the technique to translate a relational algebra tree into a single SQL query. While the resulting SQL query derived directly from the relational algebra tree is quite cumbersome and lengthy, it can be optimized to

produce a more amenable and concise query plan. The optimization techniques are provided in Section 3.5.2.

3.5.1 The Translation of Relational Operations

Similar to the data structure for the XQuery abstract syntax tree, for each of the relation operations, there is a corresponding SQL translation template. A relational algebra tree is translated into a SQL query by composing the SQL translation of the individual relational operation in the algebra tree.

During the SQL translations, the parameters for the relational operations should be properly transformed into a form that can be used in the SQL query. For example, for the *Project* operation Project[1](T_e), the parameter "1" should be transformed into "$C1$", which represents the attribute $C1$ in table T_e. We use *SQLpara* to represent the transformed parameters for the relational operations.

1. The Project Operation Translation: Project[*attributlist*](T_e)

```
SELECT attributlist
FROM   ( T_e )
```

The parameter *attributlist* contains the expression for the SQL SELECT clause.

2. The AddColumn Operation Translation: AddColumn[*expression*](T_e)

```
SELECT C_1, ..., C_n, expression AS C_{n+1}
FROM   ( T_e )
```

The *AddColumn* operation selects all the attributes in the input operation T_e along with a new attribute whose values are assigned by computing the expression in the input parameter *expression*.

3. The Select Operation Translation: Select[*condition*](T_e)

```
SELECT *
FROM   ( Te )
WHERE  condition
```

The parameter *condition* contains a condition for the SQL WHERE clause.

4. The LoopJoin Operation Translation: LoopJoin[*conditions*](T_{e_1}, T_{e_2})

There are two kinds of SQL translations based on the type of the second input operation T_{e_2}. If T_{e_2} is not a *ScalarProject* operation, the *LoopJoin* operation has the following SQL translation:

```
SELECT l.C1, ..., l.Ck, r.C1 AS Ck+1, ..., r.Cj AS Ck+j
FROM   ( Te1 ) l,  ( Te2 ) r
WHERE  conditions
```

where the parameter *conditions* contains the join conditions for the SQL WHERE clause. If T_{e_2} is a *ScalarProject* operation, the *LoopJoin* operation is translated as follows using the SQL *scalar fulselect* expression[5]:

```
SELECT l.C1, ..., l.Ck, ( Te2 )
FROM   ( Te1 ) l
WHERE  conditions
```

5. The MergeJoin Operation Translation: MergeJoin[*conditions*](T_{e_1}, T_{e_2})

The SQL translation is exactly the same as that of the *LoopJoin* operation. The reason is that, while in a physical plan we can specifically choose the relational operation at the algorithmic level, in a SQL query the choice of using the *merge join* operation or the *nested-loop join* operation is determined by the query processor in RDBMS. We can only define these operations on the conceptual level where these two operations are represented using exactly the same SQL statement.

[5]Refer to the relational algebra tree for the SQL translation of the equality comparison operation in a *where* expression (See Section 3.4.2).

6. The LoopIn Operation Translation: LoopIn[*conditions*](T_{e_1},T_{e_2})

```
SELECT *
FROM   ( T_e1 ) l
WHERE EXISTS  (
     SELECT *
     FROM   ( T_e2 ) r
     WHERE  conditions
)
```

The parameter *conditions* contains the conditions for the SQL WHERE clause.

7. The LoopExceptIn Operation Translation: LoopExceptIn[*conditions*] (T_{e_1}, T_{e_2})

```
SELECT *
FROM   ( T_e1 ) l
WHERE NOT EXISTS  (
     SELECT *
     FROM   ( T_e2 ) r
     WHERE  conditions
)
```

The parameter *conditions* contains the conditions for the SQL WHERE clause.

8. The MergeExceptIn Operation Translation: MergeExceptIn[*conditions*] (T_{e_1}, T_{e_2})

The SQL translation is exactly the same as that of the *LoopExceptIn* operation because on the conceptual level these two operations are represented using exactly the same SQL statement.

9. The CatUnion Operation Translation: CatUnion[ ](T_{e_1}, T_{e_2})

```
( SELECT *
  FROM   ( Te1 )
)
UNION ALL
( SELECT *
  FROM   ( Te2 )
)
```

There is no parameter for this operation. The result of the operation preserves duplicates.

10. The MergeUnion Operation Translation: MergeUnion$[SortOrder](T_{e_1}, T_{e_2})$

The SQL translation is exactly the same as that of the *CatUnion* operation because on the conceptual level these two operations are represented using exactly the same SQL statement.

11. The CountAggregate Operation Translation: CountAggregate $[groupbylist](T_e)$

The parameter *groupbylist* is a list of attributes that are used for the grouping operation. It is the condition of the SQL GROUPBY clause. Let $\{A1, \ldots, Ak\}$ be the attributes in *groupbylist*. There are two kinds of translation depending on the value of the parameter *SQLpara*. If the parameter *groupbylist* is not empty, then the operation is translated as follows:

```
SELECT   A1 AS C1, ..., Ak AS Ck,
         char(COUNT(*)) AS Ck+1, 0 AS Ck+2, 1 AS Ck+3
FROM     ( Te )
GROUP BY groupbylist
```

If the parameter *groupbylist* is empty, then we have the following translation:

```
SELECT   char(COUNT(*)) AS C1,0 AS C2,1 AS C3
FROM     ( Te )
```

12. The XMLFileReader Operation Translation: XMLFileReader[$XMLfile\text{-}Name$]()

Since we assume there already exist the relational encodings in RDBMS for the XML documents, the SQL translation of this operation just returns the table with the same name in the parameter *XMLfileName*.

```
SELECT C1, C2, C3
FROM   XMLfileName
```

13. The TextFileReader Operation Translation: TextFileReader[$TextFile\text{-}Name$]()

The SQL translation of the operation is the same as that for the XMLFileReader operation:

```
SELECT C1, C2, C3
FROM   TextFileName
```

14. The UNIT Operation Translation: UNIT[ ]()

In the SQL translation, the operation returns the table UNIT, which contains only one tuple. The translation is as follows:

```
SELECT 's' AS C1, 0 AS C2, 1 AS C3
FROM   UNIT
```

15. The EMPTY Operation Translation: EMPTY[]()

In the SQL translation, the table EMPTY, which contains no tuple, is used to construct a relational encoding for an empty XML forest using the exact relational schema $(C1, C2, C3)$ for the XML source data[6]. The translation is as follows:

[6]Refer to Section 2.3.1 for the relational mapping schema

```
SELECT 's' AS C1,0 AS C2,1 AS C3
FROM   EMPTY
```

16. The ScalarProject Operation Translation: $\mathsf{ScalarProject}[](T_e)$

The *ScalarProject* operation is always used together with a *LoopJoin* operation. A *LoopJoin* operation together with a *ScalarProject* operation as its second input operation are translated into a SQL *scalar fulselect* expression.

3.5.2 Optimization of the SQL Query Generation

The resulting SQL query produced directly from the relational algebra tree is quite awkward and lengthy. In order to generate a more concise SQL query, algorithms are designed for the *SQL Plan Optimizer* in the query processor to improve the SQL query generation. Example 3.5.1 compares the SQL query generated directly from the relational algebra tree with the optimized SQL query to show the effectiveness of the optimization algorithms.

Example 3.5.1 For the following relational algebra tree from Example 3.4.1:

$$
\begin{array}{l}
T_{e\_i}(T_e) = \\
\quad \texttt{Project}[1,2,5,6](\\
\quad\quad \texttt{AddColumn}[C4 - C1 * w_e](\\
\quad\quad\quad \texttt{AddColumn}[C3 - C1 * w_e](\\
\quad\quad\quad\quad \texttt{Select}[C4 < (C1+1) * w_e](\\
\quad\quad\quad\quad\quad \texttt{Select}[C1 * w_e \leq C3](\\
\quad\quad\quad\quad\quad\quad \texttt{LoopJoin}[\,](I, T_e))))))
\end{array}
$$

Let $w_e = 44$ be the width of T_e, the following SQL plan is generated automatically by traversing the above relational algebra tree:

```
(SELECT C1,C2,C5 AS C3,C6 AS C4 FROM (
  SELECT C1, C2, C3, C4, C5, C4-C1*(44) AS C6 FROM (
   SELECT C1, C2, C3, C4, C3-C1*(44) AS C5 FROM (
     SELECT * FROM (
      SELECT * FROM (
       SELECT T10_Lf.C1, T10_Rg.C1 AS C2, T10_Rg.C2 AS C3,
              T10_Rg.C3 AS C4 FROM
         I T10_Lf,
         Te T10_Rg
      ) T9
      WHERE C1*(44)<=C3
     ) T8
     WHERE C4<(C1+1)*(44)
    ) T7
   ) T6
  ) T5
 )
```

Compared to the original SQL template fragment based on which the relational algebra tree has been built,

```
( SELECT i, s, l − i ∗ w_e AS l, r − i ∗ w_e AS r
  FROM   I, T_e
  WHERE  i ∗ w_e ≤ l  AND r < (i + 1) ∗ w_e  )
```

the automatic generated SQL plan is rather lengthy. After applying the optimization algorithms the final SQL plan is as follows:

```
(SELECT T6_Lf.C1 AS C1,T6_Rg.C1 AS C2,T6_Rg.C2-T6_Lf.C1*(44) AS
        C3,T6_Rg.C3-T6_Lf.C1*(44) AS C4 FROM
    View_I T6_Lf,
    T_e    T6_Rg
 WHERE T6_Lf.C1*(44)<=T6_Rg.C2 AND T6_Rg.C3<(T6_Lf.C1+1)*(44)
)
```

We can see that the optimized SQL query is essentially the same as the original SQL translation template fragment in Example 3.5.1. In fact, by applying the optimization algorithms, the generated SQL plans can be, however, even more

concise than the SQL queries composed directly by the SQL translation templates. □

There are two ways of reducing the size of the generated SQL query. One way is to combine several SQL expressions into one single SQL SELECT block. The other one is to eliminate the duplication of the same SQL fragments in the overall SQL plan by introducing views.

Combining the SQL Expressions

Based on the SQL translations for *relational operations*, SQL fragments created by composing the SQL translations for relational operations can be combined together into one single SQL SELECT block. Example 3.5.2 shows the basic idea for combining the SQL expressions from such fragments.

Example 3.5.2 For the following fragment of the relational algebra tree from Example 3.5.1:

$$\begin{array}{l}\texttt{Select}[C4 < (C1+1) * w_e](\\ \quad \texttt{Select}[C1 * w_e \leq C3](\\ \quad\quad \texttt{LoopJoin}[\,](I, T_e))))))\end{array}$$

the corresponding SQL fragment that is generated based on the SQL translations of the relational operations is as follows:

```
...
   SELECT * FROM (
    SELECT * FROM (
     SELECT T10_Lf.C1, T10_Rg.C1 AS C2, T10_Rg.C2 AS C3,
            T10_Rg.C3 AS C4 FROM
       I T10_Lf,
       Te T10_Rg
    ) T9
    WHERE C1*(44)<=C3
   ) T8
   WHERE C4<(C1+1)*(44)
...
```

The WHERE clause "WHERE $C4 < (C1+1)*(44)$" in the outer table expression can be combined with the WHERE clause "WHERE $C1*(44) <= C3$" in the nested expression

since the outer SELECT clause "SELECT ∗" is to select all the attributes from the nested expression. By performing this combination, the above SQL fragment can be simplified into the following form:

```
...
    SELECT * FROM (
     SELECT T10_Lf.C1, T10_Rg.C1 AS C2, T10_Rg.C2 AS C3,
            T10_Rg.C3 AS C4 FROM
       I T10_Lf,
       Te T10_Rg
    ) T9
    WHERE C1*(44)<=C3 AND C4<(C1+1)*(44)
...
```

The SQL fragment can be further simplified by moving the outer WHERE clauses into the inner join expression because the outer SELECT clause is the "SELECT ∗". Thus, we have the following query:

```
...
     SELECT T10_Lf.C1, T10_Rg.C1 AS C2, T10_Rg.C2 AS C3,
            T10_Rg.C3 AS C4 FROM
       I T10_Lf,
       Te T10_Rg
     WHERE T10_Lf.C1*(44)<=T10_Rg.C2 AND T10_Rg.C3<(T10_Lf.C1+1)*(44)
...
```

As you might notice, combining the expressions is not just simply concatenating the expressions in WHERE or SELECT clauses, or relocating the WHERE clause expression; the expressions also need to be appropriately adjusted by replacing the appropriate attribute names. The simplified SQL plan in the example illustrates such adjustment. □

Based on the idea introduced in Example 3.5.2, the *SQL Plan Optimizer* is able to combine the SQL translation of the relational algebra tree fragments, which matche certain patterns, into a single SQL SELECT block. By combining the SQL expressions, the final SQL plan is more concise than or equivalent to the SQL query that is generated by directly composing the SQL translation templates for the XQuery expressions. The patterns for the relational algebra tree fragment that the *SQL Plan Optimizer* handles are shown in the following paragraphs. For the purpose of simplicity and generality, the parameters of the relational operations in the algebra tree fragment are omitted.

- Pattern 1: `Project`(`AddColumn` $*$ (`Select` $*$ (`LoopJoin`(T_{e_1}, T_{e_2}))))

  "$*$" denotes the repetition of the operation followed by the "$*$". The relational algebra tree fragment that follows the pattern can be translated into a single SQL SELECT block[7]. The SELECT clause in the SELECT block is constructed based on the parameters of the *Project* operators, the *AddColumn* operators, and the *LoopJoin* operator from the algebra tree fragment. The WHERE clause in the SELECT block is constructed based on the parameters of the *Select* operators and the *LoopJoin* operator. A typical relational algebra tree fragment that matches the pattern is the relational algebra tree that maps the following SQL fragment:

  (`SELECT` $i, s, l - i * w_e$ `AS` $l, r - i * w_e$ `AS` r
  `FROM` I, T_e
  `WHERE` $i * w_e \leq l$ `AND` $r < (i+1) * w_e$)

- Pattern 2: `Select`(`Select` $*$ (T_e))

  The algebra tree fragment that follows the pattern can be translated into a single SQL SELECT block by the *SQL Plan Optimizer*. The parameters from the *Select* operators are concatenated together to provide the expression of the WHERE clause in the resulting SQL statement.

- Pattern 3: `AddColumn`(`AddColumn` $*$ (T_e))

  The algebra tree fragment that follows the pattern can be translated into a single SQL SELECT block that consists of only one SELECT clause and one FROM clause. The expression of the SELECT clause is constructed by combining the parameters from the *AddColumn* operations.

- Pattern 4: `Project`(`AddColumn` $*$ (T_e))

  Similar to Pattern 3, a single SQL SELECT block for the algebra tree fragment is constructed by combining the parameters from the *AddColumn* operations and the *Project* operations for the SELECT clause. A typical relational algebra tree fragment that matches the pattern is the relational algebra tree that maps the following SQL fragment from the SQL translation template for the element constructor:

[7]Here, we consider the input operations T_{e_1}, T_{e_2} as views.

```
CREATE VIEW T_xnode('label', T_e) AS
 SELECT s, l + i * w_xnode AS l, r + i * w_xnode AS r
 FROM   Q_XFn
```

- Pattern 5: `Project`(`AddColumn` $*$ (`LoopJoin`(T_{e_1}, T_{e_2})))

  Similar to Pattern 1, the SELECT clause in the resulting SQL SELECT block is constructed based on the parameters of the *Project* operators, the *AddColumn* operators and the *LoopJoin* operator from the algebra tree fragment. The WHERE clause in the SELECT block is constructed based on the parameters of the *LoopJoin* operator. An example of the algebra tree fragment that matches the pattern is the algebra tree that maps the following SQL fragment from the SQL template for the XQuery *xnode* operation:

```
...
  ( SELECT i,'label',0 AS l, w_e + 1 AS r
    FROM   I, UNIT )
...
```

- Pattern 6: `Project`(`AddColumn` $*$ (`Select` $*$ (T_e)))

  Similar to Pattern 1, parameters of the *Project* operators and the *AddColumn* operators are combined to construct the SELECT clause; parameters from the *Select* operators are concatenated to construct the WHERE clause.

- Pattern 7: `Project`(`Select` $*$ (`LoopJoin`(T_{e_1}, T_{e_2})))

  The approach to translate the algebra fragment into a single SQL SELECT block is similar to Pattern 1. A sample fragment that matches the pattern is the relational algebra tree that maps the following SQL fragment from the SQL template for the XQuery *last* operation:

```
( SELECT u.i AS i, u.s AS s, u.l AS l, u.r AS r
  FROM
       ( ... ) u,
       ( ... ) vv
  WHERE vv.l ≤ u.l AND u.r ≤ vv.r AND vv.i = u.i
) Q_XFn
```

- Pattern 8: `Project`(`Select` $*$ (T_e))

  The fragment that matches the pattern is translated into a single SQL SELECT block, where the expression of the SELECT clause is provided by the parameters from the *Project* operation and the WHERE clause is constructed by concatenating the conditions from the *Select* operations.

You might notice that some patterns introduced above overlap with each other. The *SQL Plan Optimizer* chooses the longest pattern that the algebra tree fragment matches.

Eliminating Common Subexpressions

In the SQL translation templates introduced in Section 3.4.2, the input table parameters are repeatedly referenced in multiple locations within a template. For example, in the SQL translation template for the XQuery *children* operation (see Appendix A), the following *common subexpression* is referenced for three times:

```
( SELECT i, s, l − i ∗ w_e AS l, r − i ∗ w_e AS r
  FROM   I, T_e
  WHERE  i ∗ w_e ≤ l  AND r < (i + 1) ∗ w_e  )
```

Such kind of repetition frequently appears in many of the SQL templates. To simplify the query plan, *common table expressions*[8] are used to generate the named expressions for the repeatedly referenced *common subexpressions* in the SQL templates. By using *common table expressions*, the size of the generated SQL query is greatly reduced. We use *common table expressions* for the following SQL fragments:

- The SQL translation of the index set I. In the SQL translation templates, the index set I is defined as *view* and is frequently referenced in the SQL translation templates. During the SQL query generation, whenever a new index set I is constructed in the translation, a *common table expression* is used to represent the new index set I.
- The SQL plans for the input table parameters $(T_{x_1}, \ldots, T_{x_m})$ of the SQL templates. In most of the SQL translation templates for XQuery expressions, the input table parameters are repeatedly referenced for several times. Using

[8]For detailed explanation for common table expression please check SQL reference.

common table expressions to represent the table parameters can avoid the duplication of the same SQL fragments.

- Other frequently appeared SQL fragments are also replaced by the *common table expressions*.

A sampling SQL plan that is refined by using *common table expression* is shown in Example 3.5.3.

Example 3.5.3 The following XQuery expression:

`children`(`document`(*'reviews.xml'*))

can be translated into the following SQL query where *common table expressions* are used:

```
WITH
    View0 AS
        (SELECT 0 AS C1
         FROM   UNIT T1
        ) ,
    View1 AS
        (SELECT T1_Lf.C1 AS C1,T1_Rg.C1 AS C2,
                T1_Rg.C2-T1_Lf.C1*(44) AS C3,
                T1_Rg.C3-T1_Lf.C1*(44) AS C4
         FROM
           View0 T1_Lf,
           (SELECT T2_Rg.C1 AS C1,T2_Rg.C2+T2_Lf.C1*(44) AS C2,
                   T2_Rg.C3+T2_Lf.C1*(44) AS C3
            FROM   View0  T2_Lf, reviews T2_Rg
           ) T1_Rg
         WHERE  T1_Lf.C1*(44)<=T1_Rg.C2 AND
                T1_Rg.C3<(T1_Lf.C1+1)*(44)
        )
(SELECT C2 AS C1,C3+C1*(44) AS C2,C4+C1*(44) AS C3 FROM
   (SELECT *
    FROM   View1 T2_Lf
    WHERE NOT EXISTS (
       SELECT * FROM
         (SELECT *
          FROM   View1 T3_Lf
          WHERE NOT EXISTS (
             SELECT *
             FROM   View1 T3_Rg
```

```
            WHERE  T3_Lf.C3>T3_Rg.C3 AND T3_Lf.C4<T3_Rg.C4
                   AND T3_Lf.C1=T3_Rg.C1 )
        ) T2_Rg
      WHERE T2_Lf.C3=T2_Rg.C3 AND T2_Lf.C1=T2_Rg.C1
    )
   ) T1
)
```

In the WITH clause, $View0$ is the name of the the *common table expression* for the initial set I in the input environment for the whole query. The initial index set I always contains only one tuple (0). $View1$ is the *common table expression* for the SQL plan for the expression, document($'eviews'$), whose result serves as the input table parameter to the *children* operation. The SQL plan for "document($'reviews'$)" is repeatedly referenced for three times in the SQL template. By using a *common table expression*, the repeated *common subexpressions* that translate the XQuery fragment, document($'reviews'$), are replaced by the same result table $View1$. □

The optimization methods we introduced above are very effective and can greatly reduce the size of the final SQL query, often by up to 80%.

3.6 Physical Plan Translation

The translation from a relational algebra tree into a physical plan is straight forward. The intermediate relational algebra tree can be treated as a tree-form of the physical plan. Thus, the physical plan can be easily generated by traversing the intermediate relational algebra tree. The generated physical plan is very similar to the representation of the relational algebra tree shown in Section 3.4.2. Before the generated physical plan is output to the *XQuery-enhanced Relational Engine* [15], the physical plan should be optimized using the existing relational optimization techniques, such as *join order selection* algorithms. Since the focus of the thesis is the XQuery-to-SQL translation, the details of the physical plan construction are omitted in the thesis.

Chapter 4

Optimization of the XQuery-to-SQL Translation

This chapter introduces a series of optimization approaches for the XQuery-to-SQL translation. Section 4.1 introduces succinct SQL translation templates for XQuery expressions, which is an improvement over the original SQL translation templates introduced in Chapters 2 and 3. Section 4.2 presents simplified SQL translation of sequences of basic operations. The preferable *merge-join* approach, which can efficiently handle value joins in FLWR expressions, is introduced in Section 4.3.

4.1 SQL Translation Templates Optimization

Based on the SQL translation templates introduced in Section 3.4.2, we can see that for an index value $i \in I$ from an environment $E = \{I, T_{x_1} \dots T_{x_m}\}$, the corresponding tuples (s, l, r) from T_{x_i} always satisfy the two inequalities: $i * w_{x_i} \leq l$ `AND` $r < (i+1) * w_{x_i}$ (w_{x_i} is the width of T_{x_i}). Based on these two inequalities, two equations are derived and are shown in Lemma 4.1.1.

Lemma 4.1.1 Given an environment $E = \{I, T_{x_1} \dots T_{x_m}\}$, an index value $i \in I$ and its corresponding tuples $(s, l, r) \in T_{x_i}$ in $\{T_{x_1} \dots T_{x_m}\}$ satisfy the following equations:

$$l - i * w_{x_1} = \mathsf{MOD}(l, w_{x_i}) \quad (4.1)$$

$$i = l / w_{x_i} \quad (4.2)$$

where "MOD" and "/" are integer operators. □

Based on the above equations, the SQL translation templates for XQuery expressions can be simplified by eliminating the join operations of I and T_{x_i}. By performing arithmetic computations on the left-endpoint values l of the tuples from T_{x_i}, we are able to identify their corresponding environments without joining with the environment index set I. The modified SQL translation templates for XQuery expressions are shown in the following paragraphs. Since the join operation of I and T_{x_i} appears frequently in the SQL templates for XQuery expressions, such modification not only simplifies the final generated SQL query and the physical plan, but also reduced the size of the final relational plans.

4.1.1 The Simplified Templates for XQuery Basic Operations

In the original SQL translation template for XQuery basic operations (see Section 2.4.2), the input table parameters $T_{x_1}, \ldots, T_{x_m}$ should relate their tuples to the corresponding environments by performing join operations with the relation I. However, using the equations in Lemma 4.1.1, the environment that a given tuple in T_{x_i} belongs to can be identified without using the index set I. The template for XQuery basic operations is simplified as follows:

```
CREATE VIEW T_XFn(T_x1, ..., T_xm) AS
 SELECT s, l + i * w_XFn AS l, r + i * w_XFn AS r
 FROM   Q_XFn (
  ( SELECT l/w_x1 AS i, s, MOD(l, w_x1) AS l, MOD(r, w_x1) AS r
    FROM   T_x1 ),
    ...,
  ( SELECT l/w_xm AS i, s, MOD(l, w_xm) AS l, MOD(r, w_xm) AS r
    FROM   T_xm )
 )
```

4.1.2 The Simplified Templates for the FOR Expression

Similar to the modification for basic operations, the SQL translation of *for* expression can be simplified by using the equations from Lemma 4.1.1. The translation of the new environment index set I' remains unchanged, while the translations for the

new representations T'_{x_i} of the local variables and the translation for the representation T'_x of the new bound variable provided by the *for* expression are simplified as follows:

```
CREATE VIEW T'_x(T_e) AS
  SELECT s, MOD(x.l, w_e) + r.l * w_e AS l,
            MOD(x.r, w_e) + r.l * w_e AS r
  FROM   T_e x, ROOTS(T_e) r
  WHERE  r.l ≤ x.l AND x.r ≤ r.r
```

```
CREATE VIEW T'_{x_i}(T_{x_i}, T_e) AS
  SELECT s, MOD(x.l, w_{x_i}) + r.l * w_{x_i} AS l,
            MOD(x.r, w_{x_i}) + r.l * w_{x_i} AS r
  FROM   T_{x_i} x, ROOTS(T_e) r
  WHERE  r.l/w_e = x.l/w_{x_i}
```

4.1.3 The Simplified Templates for the WHERE Expressions

The Simplified Templates for the New Environment Index Set I'

The SQL translations for the new environment index set I' for different types of *where* expressions can be simplified as follows.

1) The EMPTY Operation

The simplified SQL template for the *empty* operation in the *where* expression "where empty(e) return ..." is shown as follows:

```
CREATE VIEW I'(T_e) AS
  SELECT i
  FROM   I
  WHERE NOT EXISTS (
    SELECT *
    FROM
     ( SELECT l/w_e AS i, s, MOD(l, w_e) AS l, MOD(r, w_e) AS r
       FROM   T_e
     ) T_e_i
    WHERE  I.i = T_e_i.i  )
```

By using the above simplified SQL template, the *less-equal* and *less* comparison operations in the original SQL template provided in Section 3.4.2 are replaced with an equality comparison operation.

2) The NOT EMPTY Operation

The simplified SQL template for the *not empty* operation in the *where* expression "where not empty(e) return ..." is shown as follows:

```
CREATE VIEW I'(T_e) AS
  SELECT i
  FROM   I
  WHERE EXISTS (
    SELECT *
    FROM
     ( SELECT l/w_e AS i, s, MOD(l, w_e) AS l, MOD(r, w_e) AS r
       FROM   T_e
     ) T_e_i
    WHERE  I.i = T_e_i.i  )
```

Similar to the *empty* operation, by using the above simplified SQL template, the *less-equal* and *less* comparison operations in the original SQL template provided in Section 3.4.2 are replaced with an equality comparison operation.

3) The Comparison Operations

For the *where* expression with equality comparison operation "where $e1 = e2$ return . . .", the following fragment from the original SQL translation template for the new index set I':

$$
\begin{array}{l}
(\ \texttt{SELECT}\ s\ \texttt{FROM}\ T_{e_1}\ \texttt{WHERE}\ i * w_{e_1} \leq l\ \ \texttt{AND}\ r < (i+1) * w_{e_1}\ \ ) \\
= \\
(\ \texttt{SELECT}\ s\ \texttt{FROM}\ T_{e_2}\ \texttt{WHERE}\ i * w_{e_2} \leq l\ \ \texttt{AND}\ r < (i+1) * w_{e_2}\ \ )
\end{array}
$$

can be replaced by the following equivalent SQL fragment:

$$
\begin{array}{l}
(\ \texttt{SELECT}\ s\ \texttt{FROM}\ T_{e_1}\ \texttt{WHERE}\ l/w_{e_1} = i\ \ ) \\
= \\
(\ \texttt{SELECT}\ s\ \texttt{FROM}\ T_{e_2}\ \texttt{WHERE}\ l/w_{e_2} = i\ \ )
\end{array}
$$

The SQL templates for the *where* expressions with the other comparison operations, such as ">","=>","<", and "<=", can use a similar modification.

The Simplified SQL Template for the New Representations of Variables

The SQL template for the new representations of the local variables, which are provided by the input environment for the *where* expression, is simplified as follows:

```
CREATE VIEW T'_{x_i} AS
  SELECT  s, l, r
  FROM    T_{x_i}, I'
  WHERE   l/w_{x_i} = i
```

Based on the two equations in Lemma 4.1.1, the above simplified SQL templates are equivalent to the original SQL templates introduced in Chapters 2 and 3, and simplified SQL templates produce exactly the same results as those from the original templates.

4.2 Simplified Translation for Sequences of Basic Operations

As we mentioned in previous chapters, the XQuery FLWR expressions change the sequence of input environments by either expanding the size of the environments or filtering out the unwanted environments. For XQuery basic operations, however, the input environments remain unchanged after the operations. For example, a *children* operation, $\mathsf{children}(e)$, returns all the children of the resulting XML forest obtained by evaluating the expression e without changing the input environment E.

The SQL translations for the various basic operations share the same pattern. At the beginning of the translation for a basic operation, the index set values i are introduced to the tuples from the input table parameters $(T_{x_1}, \ldots, T_{x_m})$ to construct the corresponding tuples (i, s, l', r') that contain the input environment information. The following SQL fragment[1] is used to construct such kind of tuples for an input table parameter T_{x_i}:

```
( SELECT l/w_{x_i} AS i, s, MOD(l, w_{x_i}) AS l', MOD(r, w_{x_i}) AS r'
  FROM   T_{x_i} ) T_{x_i}_i
```

At the end of the translation for a basic operation, the index values i are eliminated from the resulting tuples by transforming the tuples (i, s, l', r') back into the form of (s, l, r). The following SQL fragment achieves such goal:

```
SELECT s, l' + i * w_XFn AS l, r' + i * w_XFn AS r
FROM   (
         ⋮
       ) Q_XFn
```

During the evaluation of a sequence of consecutive basic operations, the environment index values i are repeatedly introduced/eliminated from the tuples. Since the environment does not change during the evaluation of a sequence of basic operations, such repeated introduction/elimination is unnecessary and should be avoided. Algorithms are designed to detect sequences of consecutive primitive

[1] In the following sections, we use the simplified templates for the SQL translations for XQuery expressions.

operations within the input query and to translate each sequence together.

Example 4.2.1 The following sequence of basic operations is an XQuery fragment from XMark query Q8 [1]:

$$\mathsf{children}(\mathsf{select}('person', \mathsf{children}(\mathsf{select}('buyer', \mathsf{children}(\$t)))))$$

During the optimized evaluation process, the environment index values i are firstly introduced to the input tuples (s, l, r) from $\$t$ before the evaluation of the beginning operation, $\mathsf{children}(\$t)$, in the sequence; at the end of the translation for the last operation, the index values i are eliminated from resulting tuples (i, s, l', r') and final tuples in the form of (s, l, r) are returned. □

The algorithm for XQuery basic operations to handle their input and output tuples within different situations (whether they are within a sequence of primitive operations or not.) is shown as follows.

Algorithm TranslateXFn. This algorithm is applied to translate an XQuery basic operation into a relational algebra tree during the evaluation process. The algorithm avoids the repeated introduction/elimination of the index set I whenever the current basic operation is a part of a sequence of consecutive basic operations.

The input of the algorithm includes the input table T_e (with a width w_e), which is the result of the child node of the current basic operation, the input environment, *environment*, the information about the type of the parent node for the current operation, *parentType*, and the type of the child node, *childType*. This information serves as a switch to control what form of tuples is returned by the current operation. The current basic operation returns different form of tuples ((s, l, r) or (i, s, l', r')) according to the parent and child expression types. □

Using the optimization approaches introduced in Section 4.1 and Section 4.2, the *XQuery-to-SQL Query Processor* is able to generate a more concise SQL query. Compared to the SQL plan that is generated without using these kinds of optimization approaches, the size of the refined SQL plan for a complex XQuery query can be reduced by up to 45%. For example, for a SQL query, which is generated without applying these optimization approaches, has a size of 18Kb, while the optimized SQL query has a much smaller size of 10Kb.

Algorithm 1 TranslateXFn

Require: T_e, *environment*, *childType*, *parentType*

1: $T_{e\_i} = T_e$
2: {If the child node is not a basic operation, T_e contains tuples (s, l, r).}
3: **if** childType is not a basic operation **then**
4: T_{e_i} = the relational algebra tree for the SQL template that transforms tuples $(s, l, r) \in T_e$ into the form of (i, s, l', r')
5: **end if**
6: Translate the current basic operation with input table T_{e_i} and obtain a relational algebra tree $T_{Q_{\mathsf{XFn}}}$, which returns tuples in the form of (i, s, l', r').
7: w_{XFn} = the width function with the input w_e {Set the width of the result of the current basic operation.}
8: {If the parent node is a FLWR expression, tuples (s, l, r) should be returned.}
9: **if** parentType is a FLWR expression **then**
10: T_{XFn} = the relational algebra tree for the SQL template that converts the resulting tuples of $T_{Q_{\mathsf{XFn}}}$ back into the form of (s, l, r)
11: **else**
12: $T_{\mathsf{XFn}} = T_{Q_{\mathsf{XFn}}}$
13: **end if**

Ensure: T_{XFn}

4.3 Merge-join Approach to FLWR Expressions

To achieve performance in processing XQuery queries comparable to that of using RDBMS, the inefficient *nested-loop* evaluation for FLWR expressions should be avoided. This section introduces the preferable *merge-join* approach that efficiently handles value joins in the FLWR expressions. By using the *merge-join* approach, the nested *for* expressions that are independent of each other are evaluated independently. The *merge join* technique[2] can be applied to evaluate the *where* expressions that correlate the variables bound by the *for* expressions. The following example illustrates the problem scenario.

Example 4.3.1 We use the query "*For each book found at both bstore1.example.com and bstore2.example.com, list the title of the book and its price from each source.*" (Q5) from W3C Use Case "XMP" [3] as an example:

[2]Similar to the relational *merge join* algorithm or other algorithmically preferable join algorithms.

```
<books-with-prices>
  {
    for $b in doc("http://bstore1.example.com/bib.xml")//book,
        $a in doc("http://bstore2.example.com/reviews.xml")//entry
    where $b/title = $a/title
    return
        <book-with-prices>
            { $b/title }
            <price-bstore2>{ $a/price/text() }</price-bstore2>
            <price-bstore1>{ $b/price/text() }</price-bstore1>
        </book-with-prices>
  }
</books-with-prices>
```

This query has two nested *for* expressions that are independent to each other (shown in boldface). Following the *for* expressions is a *where* expression (shown in italic) with an equality comparison condition whose two operands are path expressions that depend on variables $\$a$ and $\$b$, respectively.

In the *nested-loop* evaluation, given the initial input environment $E0 = \{I_0\}$ for the whole query, the first *for* expression is first evaluated to produce an environment E^b, which provides binding values for the variable $\$b$. Then, the second *for* expression is evaluated using the environment E^b to produce a new environment E^{ba} containing the binding values for variables $\$b$ and $\$a$. Let E^a represents the bindings for the variable $\$a$ generated by evaluating the second *for* expression using the environment $E0$. The new environment E^{ba} is the result of a *Cartesian product* of the two sets of binding values for the variables $\$b$ and $\$a$ ($E^{ba} = E^b \times E^a$). Finally, the equality comparison condition in the *where* clause is applied to filter out the unwanted bindings for the variables.

In the *merge-join* approach the two *for* expressions are evaluated independently using the same input environment $E0$. As a result, two new environments E^b and E^a are created, which provide sufficient information to evaluate the equality comparison condition in the *where* clause. These two environments can be **merged**[3] together with the *where* condition, using an efficient *merge join* algorithm to avoid the generation of a *Cartesian product* as that in the *nested-loop* evaluation. The resulting environment is exactly the same environment generated using the *nested-loop* evaluation.

[3]Similar to the *merge join* in RDBMS.

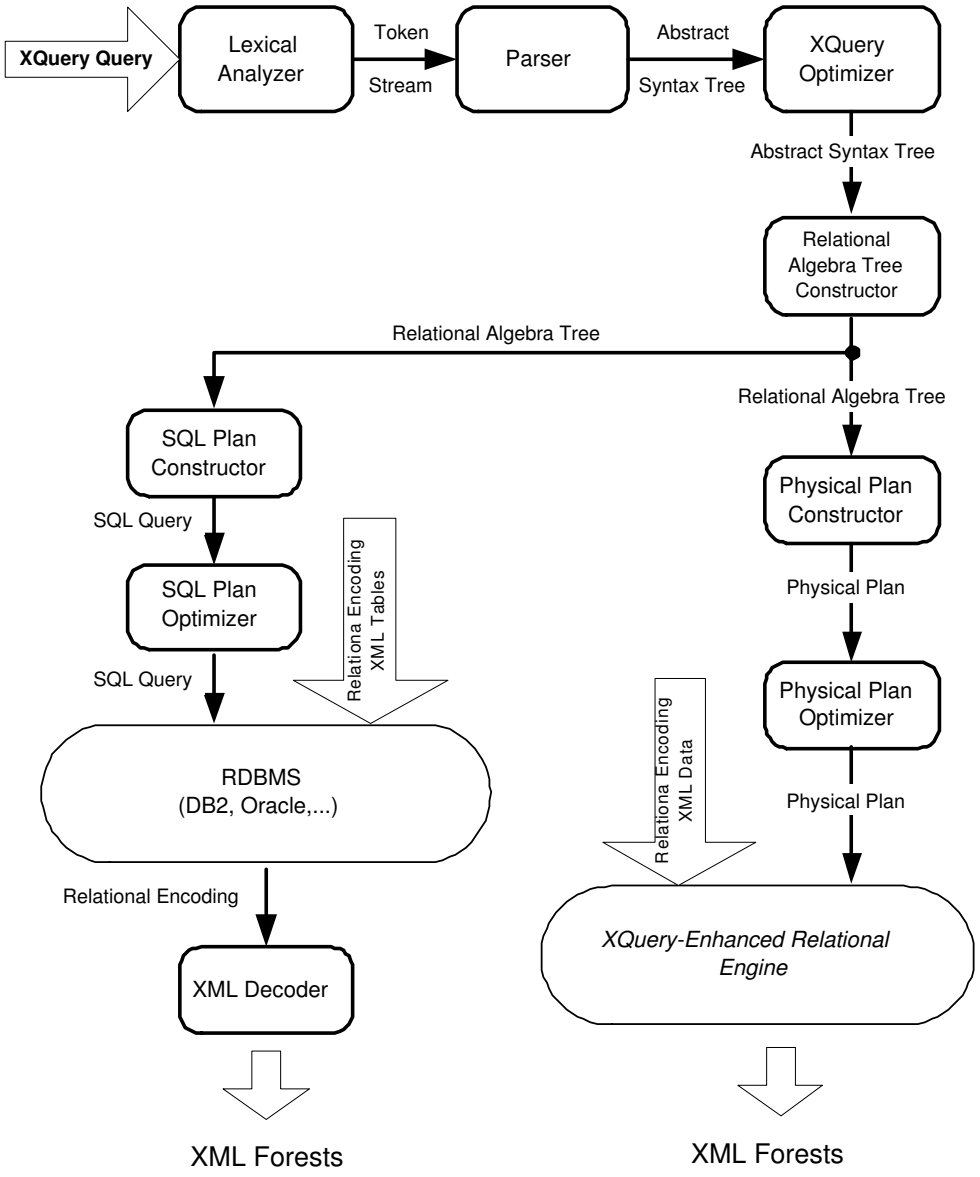


Figure 4.1: The Optimized XQuery-to-SQL Query Processor

Using the *merge-join* approach, the query processor efficiently evaluates the nested FLWR expressions using the preferable *merge join* operation and avoids the inefficient *nested-loop* join operation. □

Before we can utilize the *merge-join* approach, we must identify the *merge-join* fragments of the input XQuery that can utilize the *merge-join* approach. However,

it is very difficult to design an algorithm to capture all the possible fragments. We proposes two heuristic *merge-join* patterns, the Nested-FOR Pattern and the Multi-FOR Pattern, to identify a considerable number of the *merge-join* fragments in XQuery queries. In addition, we design *rewriting rules* for XQuery expressions that improve our chances of applying the *merge-join* approach. Details for the two *merge-join* patterns and the rewriting rules are introduced in the subsequent sections.

Based on the *merge-join* patterns and the rewriting rules, an *XQuery Optimizer* is designed and added to the *XQuery-to-SQL Query Processor*. The updated framework for the optimized XQuery processor is shown in Figure 4.1. Given an input XQuery, the *XQuery Optimizer* first identifies the *merge-join* fragments by performing query rewriting using the rewriting rules and the pattern recognition using the *merge-join* patterns. Then, the *XQuery Optimizer* rewrites the recognized fragments into *merge-join* expressions, which are specially designed to distinguish the *merge-join* fragments to other XQuery expressions[4] and to facilitate the use of the *merge-join* strategy in latter phases of the translation. The *XQuery Optimizer* is introduced in Section 4.5.

4.3.1 Definitions

Before we describe the details of the *merge-join* approach, we introduce the following definitions.

Definition 4.3.2 Given a set of variables $\{\$x_1, \ldots, \$x_m\}$ and an XQuery expression *expr*, we say that ***expression*** *expr* ***depends on variables*** $\{\$x_1, \ldots, \$x_m\}$ if $\{\$x_1, \ldots, \$x_m\}$ is the set of all variables that appear in the expression *expr*. We use the notation of $expr[\$x_1, \ldots, \$x_m]$ to represent an XQuery expression *expr* that *depends on* variables $\$x_1, \ldots, \$x_m$. □

Definition 4.3.3 Given an environment E_x and an XQuery expression *expr*, we say that ***expression*** *expr* ***depends on environment*** E_x if *expr* and E_x satisfy the following constraints:

Let $\{\$x_1, \ldots, \$x_m\}$ be the list of all the variables that the XQuery expression *expr depends on*. Let environment E_x be one of the environments $\{E_{x_1}, \ldots, E_{x_k}\}$[5]

[4]The syntax of *Minimal XQuery* is extended by adding the *merge-join* expressions.

[5]During the evaluation of an XQuery query, a new environment is produced whenever a FLWR expression in the query is evaluated. These kinds of environments that occur during the evaluation procedure for an XQuery query form the series of environments $\{E_{x_1}, \ldots, E_{x_k}\}$ in the order of their first appearance during the evaluation process.

created during the evaluation of the XQuery query. Environment E_x is the **first environment** that contains the value bindings for all variables in $\{\$x_1, \ldots, \$x_m\}$.

We use the notation of $expr[E_x]$ to represent an XQuery expression *expr* that *depends on* environment E_x. □

Definition 4.3.4 Given an environment E_x and a *variable* expression $\$x$ in an XQuery query, we say that ***variable*** $\$x$ ***depends on environment*** E_x if $\$x$ and E_x satisfy the following constraints:

Let *expr* be the expression that provides binding values for the variable $\$x$[6] in the XQuery. Let the environment E_x be one of the environments $\{E_{x_1}, \ldots, E_{x_k}\}$ created during the evaluation of the XQuery. If expression *expr depends on* the environment E_x, then we say that ***variable*** $\$x$ ***depends on environment*** E_x. Let $\{\$x_1, \ldots, \$x_m\}$ be the list of all the variables that expression *expr* depends on. Then, we also say that ***variable*** $\$x$ ***depends on variables*** $\$x_1, \ldots, \$x_m$. □

Definition 4.3.5 Given two *variable* expressions $\$x$ and $\$y$ in an XQuery query, we say that ***variables*** $\$x$ ***and*** $\$y$ ***are independent of each other*** if $\$x$ and $\$y$ satisfy the following constraints:

Let $\{\$x_1, \ldots, \$x_m\}$ and $\{\$y_1, \ldots, \$y_n\}$ be the lists of all variables on which the variables $\$x$ and $\$y$ respectively depend. We say that ***variables*** $\$x$ ***and*** $\$y$ ***are independent of each other*** if $\$x$ is not in $\{\$y_1, \ldots, \$y_n\}$ and $\$y$ is not in $\{\$x_1, \ldots, \$x_m\}$. □

4.3.2 The Nested-FOR Pattern

The Pattern

Pattern 4.3.1 (Nested-FOR) An XQuery fragment that matches the following XQuery pattern can utilize the *merge-join* approach:

```
FOR $x IN expr[E_x] DO
WHERE pathexp[$x] = pathexp[$x_i] RETURN
  (WHERE pathexp[$x] = pathexp[$x_i] RETURN )*
```

[6]For an XQuery query, only the *for* expressions "for $x in *expr* do" and the *let* expressions "let $x = *expr* in" create bindings to the variables that appear in the XQuery query. The expression *expr* in the *for*/*let* expression provides bindings to the variable $x in the *for*/*let* expression.

We call an XQuery fragment that matches the *merge-join* pattern a *pattern fragment*. The expression $expr[E_x]$ from the pattern represents an XQuery expression that depends on the environment E_x generated during the evaluation of the surrounding expressions. "$*$" means zero or more matches.

The pattern has the following constraints:

- The pattern starts with a *for* expression, which is followed by a list of *where* clauses. There must be at least one *where* clause in the *where* clauses sequence.
- The conditions in the *where* clauses are label equality comparisons. For each *where* condition, there are two path expressions $pathexp[\$x]$ and $pathexp[\$x_i]$. $pathexp[\$x]$ represents a path expression that only *depends on* the variable $\$x$ introduced by the *for* expression in the pattern. $pathexp[\$x_i]$ represents a path expression that only *depends on* a variable $\$x_i$ from outer bindings[7]. Let $\{\$x_1, \ldots, \$x_l\}$ be the list of all variables that appear in path expressions $pathexp[\$x_i]$. Variable $\$x$ is *independent* of any variable from the set $\{\$x_1, \ldots, \$x_l\}$. □

The following examples show the pattern fragments that match Nested-FOR Pattern.

Example 4.3.6 There are two fragments (shown in boldface) that match Nested-FOR Pattern in the following query:

```
for  $r in document("record.xml")//record,
     $p in document("patient.xml")//patient,
where $r/patientSSNo=$p/Ssno
return
   for    $e in document("record.xml")//entry
   where $e/diagnosis=$r/entry/diagnosis
    and $e/diagnosis = "flu"
   return <res> $p/name,
          <occured>$e/@date</occured></res>
```

The first fragment starts from the second *for* loop "**for $p in ...**" in the query. The second fragment starts from the third *for* loop "**for $e in ...**". These two fragments are replaced with two *merge-join* expressions by the *XQuery Optimizer* and can be evaluated using the *merge-join* approach. □

[7]The outer bindings could be provided by the outer *for* loops or the outer *let* expressions.

The pattern fragments shown in Example 4.3.6 are the simplest fragments that match Nested-FOR Pattern. A more complex fragment that follows Nested-FOR Pattern is shown in the following example.

Example 4.3.7 The following query has one fragment (shown in boldface) that follows Nested-FOR Pattern:

```
for $p in document("patient.xml")//patient,
    $e in $p/entry,
    $s in $e/sn
    return
      <s>
        for $r in $p/id
        where $r/patientSSNo=$e/Ssno return
           where $e/diagnosis="flu" return
           <res> $p/name,
           <occured>$e/@date</occured></res>
      </s>
```

The *for* loop "**for $r in $p/id** " together with the *where* clause can be replaced with a *merge-join* expressions by the XQuery Optimizer and be executed using *merge-join* approach. □

The Merge-join Expression

An XQuery fragment that follows Nested-FOR Pattern is replaced with one single *merge-join* expression named *NestedFOR*, which is specially designed to represent the pattern. The *merge-join* expression for Nested-FOR Pattern is shown as follows:

`NestedFOR`[*conditions*](*forloop*)

where *conditions* lists the conditions from the sequence of *where* clauses, and *forloop* lists the *for* expression. The *NestedFOR* expression is added to the *Minimal XQuery* syntax as a new kind of expression. Hence, the input query shown in Example 4.3.6 is rewritten by the XQuery Optimizer as follows:

```
for $r in document("record.xml")//record do
NestedFOR[$r/patientSSNo=$p/Ssno]
         ($p in document("patient.xml")//patient)
return
   NestedFOR[$e/diagnosis=$r/entry/diagnosis]
            ($e in document("record.xml")//entry)
   return
      where $e/diagnosis = "flu"
      return <res> $p/name, <occured>$e/@date</occured></res>)
```

Translation of the Pattern

Let $E_{in} = \{I_{in}, T_{x_1}, \ldots, T_{x_n}\}$ be the input environment for the pattern fragment of Nested-FOR Pattern. E_{in} is created by evaluating the expressions that appear before the pattern fragment and serves as an input environment for the *for* expression (the first expression) in the pattern fragment. Let $E_x = \{I_x, T^x_{x_1}, \ldots, T^x_{x_m}\}$ $(m \leq n)$ be the environment that the expression $expr[E_x]$ in the *for* expression[8] *depends on*. The *merge-join* evaluation for the pattern can be divided into four steps.

Step 1. Evaluate the FOR Expression:

Since variable $\$x$ *depends on* environment E_x, the *for* expression from the pattern, "`FOR` $\$x$ `IN` $expr[E_x]$ `DO`", can be evaluated in the environment E_x. The SQL translation technique introduced in Section 4.1 is used to translate the *for* expression. As a result, a new environment $E'_x = \{I'_x, T^{x'}_{x_1}, \ldots, T^{x'}_{x_m}, T^{x'}_x\}$ is produced.

Step 2. Evaluate the Path Expressions in WHERE Expressions:

Before performing the equality comparisons for the *where* clauses, the path expressions in the *where* conditions are evaluated. The methodology for translating path expressions is exactly the same as discussed in the previous parts of the thesis.

Let k be the total number of *where* clauses in the pattern fragment. Based on the constraints of the pattern, there are two types of path expressions in the *where* clauses and each type has a total number of k path expressions. Let $pathexp^x_1[\$x] =$

[8] For simplicity, if not specifically stated, the expressions we mention in the *merge-join* approach refer to the expressions that appear in the pattern fragment.

$pathexp_1^{in}[\$x_1]$, ..., $pathexp_k^x[\$x] = pathexp_k^{in}[\$x_l]$ be the equality comparison conditions from the *where* clauses. Path expressions, $pathexp_1^x[\$x], \ldots, pathexp_k^x[\$x]$, which only *depend on* the variable $\$x$ from the *for* expression in the pattern are evaluated in the environment E'_x and produce the corresponding results $T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}$, listed in the order of their corresponding *where* conditions. These tables along with the index set I'_x are joined together into one table $IT_x^{x'}$ to provide information for further evaluation of the *where* clauses.

The second form of path expressions, $pathexp_1^{in}[\$x_1], \ldots, pathexp_k^{in}[\$x_l]$, which depend on variables bound by the outer bindings are computed with the environment E_{in}. Let $T_{e_1}^{in}, \ldots, T_{e_k}^{in}$ be the corresponding results. These tables along with the index set I_{in} are joined into one table IT_{xi}^{in} for the further evaluation of the *where* clauses.

Step 3. Evaluate the WHERE Expressions Using the Merge Join:

The two environments $\{I'_x, T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}\}$ and $\{I_{in}, T_{e_1}^{in}, \ldots, T_{e_k}^{in}\}$ obtained from previous step are used to evaluate the *where* clauses. Since the *where* conditions are the label equality comparisons, for each index value $i \in I'_x$, there should exist one and only one corresponding tuple in $T_{e_i}^{x'} \in \{T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}\}$. Similarly, for each index value $i \in I_{in}$, there should exist one and only one corresponding tuple in $T_{e_i}^{in} \in \{T_{e_1}^{in}, \ldots, T_{e_k}^{in}\}$. If the above constraints are not satisfied, an error should be reported and the execution procedure should be terminated. The *scalar fulselect* expression is used to properly handle such situation.

Based on the constraints for the equality comparison condition, *scalar fulselect* expressions are used to join the environment $\{I'_x, T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}\}$ into table $IT_x^{x'}$ and the environment $\{I_{in}, T_{e_1}^{in}, \ldots, T_{e_k}^{in}\}$ into table IT_{xi}^{in}, respectively. $IT_x^{x'}$ and IT_{xi}^{in} both contain attributes $(i, s_1, \ldots, s_k)$ with one column for the index values from I'_x or I_{in} and a column of node labels s_i for every table $T_{e_i}^{x'} \in \{T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}\}$ or $T_{e_i}^{in} \in \{T_{e_1}^{in}, \ldots, T_{e_k}^{in}\}$.

Let $w_{e_1}^x, \ldots, w_{e_k}^x$ be the corresponding widths for tables $T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}$. To join the environment $\{I'_x, T_{e_1}^{x'}, \ldots, T_{e_k}^{x'}\}$ into table $IT_x^{x'}$, we have the following SQL translation template:

```
CREATE VIEW IT_x^x' AS
  SELECT i,
         ( SELECT s FROM T_e1^x' WHERE l/w_e1^x = I'_x.i ) AS s_1,
          ⋮
         ( SELECT s FROM T_e(k-1)^x' WHERE l/w_e(k-1)^x = I'_x.i ) AS s_(k-1),
         ( SELECT s FROM T_ek^x' WHERE l/w_ek^x = I'_x.i ) AS s_k
  FROM I'_x
```

The same SQL translation template is also applied to the environment $\{I_{in}, T^{in}_{e_1}, \ldots, T^{in}_{e_k}\}$ to generate the table IT^{in}_{xi}. Then, the two resulting tables, $IT^{x'}_{x}$ and IT^{in}_{xi}, are joined together to obtain the set of pairs of indices for the matching environments that satisfy the *where* conditions. The SQL translation template and the fragment of the physical plan for generating the indices pairs are introduced as follows.

For the two environments E_{in} and E_x that are used to evaluate the pattern fragment, E_x is either an environment generated before E_{in} or exactly the environment E_{in}[9]. Let $\$f_1, \ldots, \$f_j$ represent the variables that satisfy the following constraints:

1. the binding values of the variables are provided in the environment E_{in} while not in the environment E_x;
2. the variables are bound by the nested ***for*** expressions;
3. the *for* expressions, which provide bindings for the variables, contribute to the extension of the environment E_x into the environment E_{in}.

Example 4.3.8 is used to illustrate the variables that satisfy the constraints for a *merge-join* fragment in an XQuery.

Example 4.3.8 For the *merge-join* fragment (shown in boldface) in the following query.

[9]If the environment E_x is exactly the environment E_{in}, then the evaluation process of the *merge-join* fragment using the *merge-join* approach is exactly the same as that using the *nested-loop* strategy.

```
for $p in document("patient.xml")//patient,
    $e in $p/entry,
    $s in
          for $t in $e/sn
          return $t
    return
      <s>
        for $r in $p/id
        where $r/patientSSNo=$e/Ssno return
           where $e/diagnosis="flu" return
           <res> $p/name,
           <occured>$e/@date</occured></res>
      </s>
```

the variables that satisfy the constraints defined above are the variables $\$e$ and $\$s$. Although the variable $\$t$ is also bound by a *for* expression, it does not satisfy the third constraint because its host *for* expression does not contribute to the creation of the input environment for the *merge-join* fragment. □

For the variables $\$f_1, \ldots, \$f_j$ defined above, let $T_{f_1}, \ldots, T_{f_j}$ (with widths $w_{f_1}, \ldots, w_{f_j}$) be the corresponding representations in E_{in}. The SQL translation template for creating the set of pairs of indices based on the *where* conditions is as follows:

```
CREATE VIEW I_pair AS
  SELECT op0.i AS i_in, op1.i AS i_x
  FROM   IT_xi^in op0, IT_x^x' op1
  WHERE  op0.s_1 = op1.s_1 AND ... AND op0.s_k = op1.s_k
    AND  (...(op0.i/w_{f_j})/w_{f_{j-1}}...)/w_{f_1} = op1.i/w_x
```

where w_x is the width of the representation $T'_x \in E'_x$ for the variable $\$x$ whose value is assigned during the generation of T'_x. As a result, the table I_{pair} contains the set of pairs of indices for the matching environments from E_{in} and E'_x.

For the physical plan fragment that generates the set of pairs of indices, tuples in tables $IT_x^{x'}$ and IT_{xi}^{in} are ordered first in the order of the label values from the resulting tables of the path expressions, which are from the most selective equality comparison condition in the *where* clauses. Let $pathexp_i^x[\$x] = pathexp_i^{in}[\$x_i]$ be such most selective condition in the *where* clauses. Then an preferable ***merge join*** operation is applied to join the tables IT_x^x and IT_{xi}^{in} with the condition

$IT_x^x.s_i = IT_{xi}^{in}.s_i$, which corresponds to the most selective *where* condition. After that, a sequence of relational *select* operations[10] whose conditions correspond to the remainder of the *where* conditions are applied to the joined result[11]. Finally, a *project* operation is applied to the results to extract the set of pairs of indices $(IT_x^{x'}.i, IT_{xi}^{in}.i)$ for the matching environments.

Compared with the SQL translation template for I_{pair}, in the corresponding physical plan fragment, we can specifically choose the *merge join* operation to obtain the indices pairs for the matching environments, while for the SQL translation, the choice of using the *merge join* operation or the *nested-loop join* operation is determined by the query processor in RDBMS[12].

Step 4. Construct the New Environment:

There are two sets of environments obtained in the previous steps of evaluation: the environment E_{in} and the environment E'_x. The two environments and the matching indices I_{pair} are used to construct the input environment for the remainder of the query. This environment is identical to the one we would have obtained by using the *nested-loop* evaluation. Let $E' = \{I', T'_{x_1}, \ldots, T'_{x_n}, T'_x\}$ be the environment produced by the *merge-join* approach, where I' is the new environment index, $T'_{x_1}, \ldots, T'_{x_n}$ are the new representation for the local variables provided by E_{in}, and T'_x is the representation of variable $\$x$. Let w_{x_i} be the width of $T_{x_i} \in \{T_{x_1}, \ldots, T_{x_n}\}$ in E_{in}. The environment E' is defined as follows:

```
CREATE VIEW I'(I_pair) AS
  SELECT i_in * w_x + MOD(i_x, w_x) AS i
  FROM   I_pair
```

```
CREATE VIEW T'_{x_i}(I_pair, T_{x_i}) AS
  SELECT s, (i_in * w_x + MOD(i_x, w_x)) * w_{x_i} + MOD(l, w_{x_i}) AS l,
            (i_in * w_x + MOD(i_x, w_x)) * w_{x_i} + MOD(r, w_{x_i}) AS r
  FROM   I_pair, T_{x_i}
  WHERE  l/w_{x_i} = i_in
```

[10]The *select* operation is a relational operation introduced in Chapter 3.

[11]If there are only one *where* clause in the *merge-join* expression, then the *select* operations are omitted.

[12]Hence, although our processor generates the SQL queries in such a way that enables the use of the *merge join*, the RDBMS might not choose the preferable *merge join*.

```
CREATE VIEW T'_x(I_pair, T'_x) AS
  SELECT s, (i_in * w_x + MOD(i_x, w_x)) * w_x + MOD(l, w_x) AS l,
            (i_in * w_x + MOD(i_x, w_x)) * w_x + MOD(r, w_x) AS r
  FROM   I_pair, T'_x
  WHERE  l/w_x = i_x
```

Figure 4.2 illustrates the translation process for evaluating the fragments that match Nested-FOR Pattern. Example 4.3.9 illustrates the SQL translation for Nested-FOR Pattern.

Example 4.3.9 The following XQuery fragment is used to illustrate the process of the translation for Nested-FOR Pattern:

```
for $c in document("client.xml")/client/id do
for $o in document("order.xml")/order/cid do
where $c/text()=$o/text()
return ...
```

In this XQuery fragment, there is one *merge-join* fragment (shown in boldface) that matches Nested-FOR Pattern. For the path expression, `document("client.xml")/client/id`, let T_{id} be the representation of the path expression result evaluated using the initial environment $E_0 = \{I_0\}$. For the path expression, `document("order.xml")/order/cid`, let T_{cid} be the representation of the path expression result evaluated in the initial environment E_0. The tuples in T_{id} and T_{cid} are as follows:

I
i
0

$T_{\texttt{id}}(w_{\texttt{id}} = 20)$		
s	l	r
id	1	4
id1	2	3
id	11	14
id2	12	13

$T_{\texttt{cid}}(w_{\texttt{cid}} = 20)$		
s	l	r
cid	5	8
id1	6	7
cid	15	18
id3	16	17

The input environment for the *merge-join* fragment is $E^c = \{I_c, T_c\}$, which is generated by evaluating the first *for* expression "`for $c in ...`" in the query. The *Dynamic Interval* encodings for I_c and T_c are as follows:

I_c
i
1
11

$T_c(w_c = 20)$		
s	l	r
id	21	24
id1	22	23
id	231	234
id2	232	233

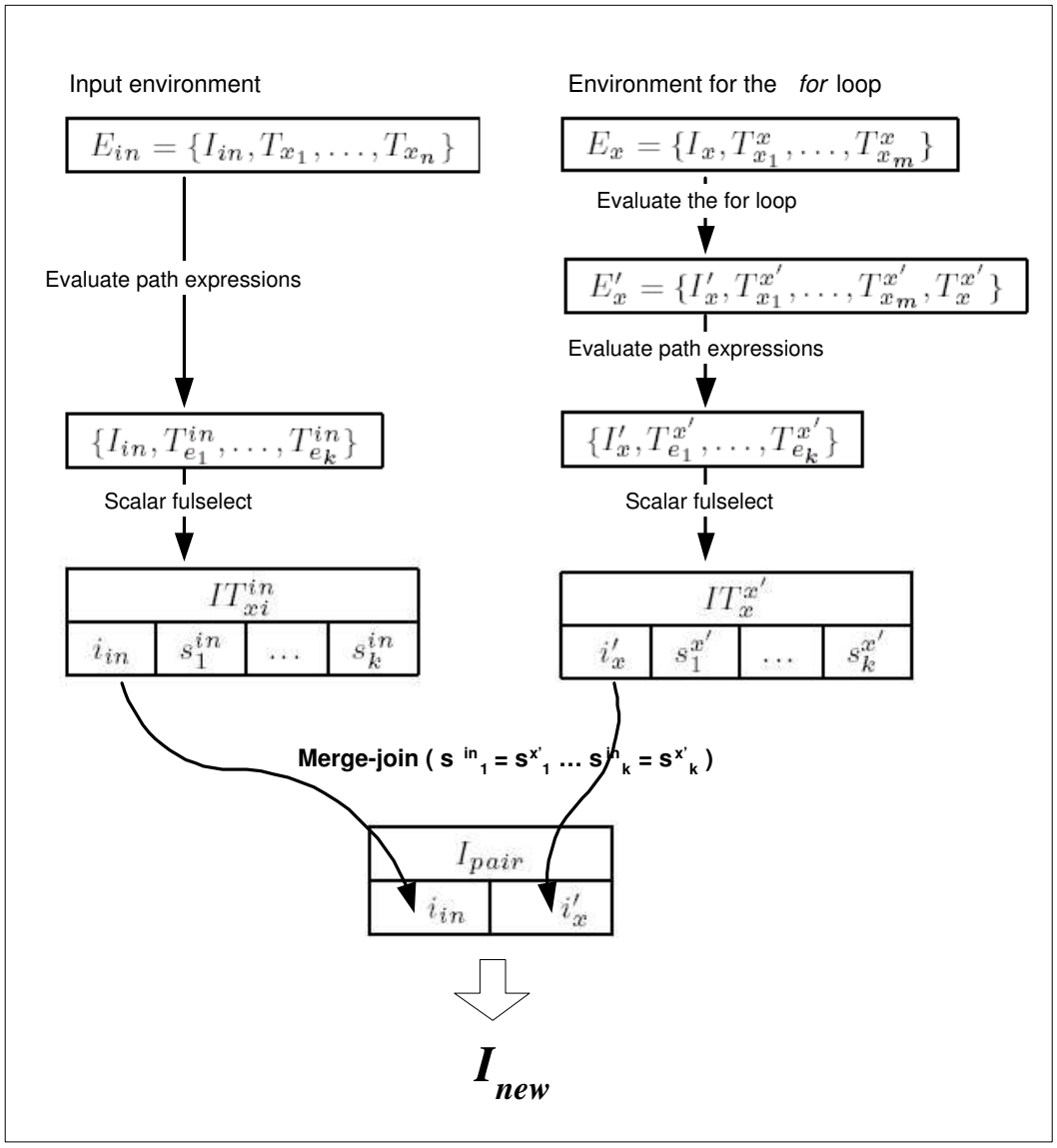


Figure 4.2: The SQL Translation for Nested-FOR Pattern

The *merge-join* fragment is computed using the following steps:

1. First, the *for* expression "**for $o in ...**" is evaluated using the initial environment E_0 since it does not *depend on* any variable. The resulting environment $E^o = \{I_o, T_o\}$ after such evaluation is as follows:

I_o
i
5
15

T_o ($w_o = 20$)		
s	l	r
cid	105	108
id1	106	107
cid	315	318
id3	316	317

2. Then, the path expression **$c/text()** in the *where* clause is computed using input environment E^c and produces the resulting table $T^c_{\$c/text}$ with the width $w_{\$c/text} = w_c$. The path expression **$o/text()** is evaluated using input environment E^o and generates the resulting table $T^o_{\$o/text}$ with the width $w_{\$o/text=w_o}$. The *Dynamic Interval* encodings for $T^c_{\$c/text}$ and $T^o_{\$o/text}$ are as follows:

I_c
i
1
11

$T^c_{\$c/text}$ ($w_{\$c/text} = 20$)		
s	l	r
id1	22	23
id2	232	233

I_o
i
5
15

$T^o_{\$o/text}$ ($w_{\$o/text} = 20$)		
s	l	r
id1	106	107
id3	316	317

3. *Scalar fulselect* expressions are then used to join the environment $\{I_c, T^c_{\$c/text}\}$ into table IT^c_c and the environment $\{I_o, T^o_{\$o/text}\}$ into table IT^o_o, respectively:

```
CREATE VIEW IT_c^c AS
  SELECT i,
     ( SELECT s FROM T^c_$c/text WHERE l/w_$c/text = I_c.i ) AS s_1
  FROM I_c

CREATE VIEW IT_o^o AS
  SELECT i,
     ( SELECT s FROM T^o_$o/text WHERE l/w_$o/text = I_o.i ) AS s_1
  FROM I_o
```

Tuples in tables IT^c_c and IT^o_o are as follows:

IT^c_c	
i	s_1
1	id1
11	id2

IT^o_o	
i	s_1
5	id1
15	id3

These two tables are joined together using a join condition corresponding to the equality comparison condition in the *where* clause. Then, the set of pairs of indices for the matching environments are extracted from the results of the joined tables. We have the following SQL translation to generate the pairs of indices :

```
CREATE VIEW I_pair AS
  SELECT op1.i AS i_in, op2.i AS i_x
  FROM   IT_c^c op1, IT_o^o op2
  WHERE  op1.s_1 = op2.s_1
```

In our example, the tuples in table I_{pair} are as follows:

I_{pair}	
i_{in}	i_x
1	5

4. After the pairs are obtained, the new environment $E' = \{I', T'_c, T'_o\}$ for the remainder of the query is constructed using the templates introduced in Step 4 of the translation of Nested-FOR Pattern:

```
CREATE VIEW I'(I_pair) AS
  SELECT i_in * w_o + MOD(i_x, w_o) AS i
  FROM   I_pair
```

```
CREATE VIEW T'_c(I_pair, T_c) AS
  SELECT s, (i_in * w_o + i_x) * w_c + MOD(l, w_c) AS l,
            (i_in * w_o + i_x) * w_c + MOD(r, w_c) AS r
  FROM   I_pair, T_c
  WHERE  l/w_c = i_in
```

```
CREATE VIEW T'_o(I_pair, T_o) AS
  SELECT s, (i_in * w_o + i_x) * w_o + MOD(l, w_o) AS l,
            (i_in * w_o + i_x) * w_o + MOD(r, w_o) AS r
  FROM   I_pair, T_o
  WHERE  l/w_o = i_x
```

The *Dynamic Interval* encoding for the environment E' is as follows:

I'
i
25

T'_c ($w'_c = 20$)		
s	l	r
id	501	504
id1	502	503

T'_o ($w'_o = 20$)		
s	l	r
cid	505	508
id1	506	507

In order to justify the correctness of the *merge-join* approach for the *merge-join* fragment in the example, the *Dynamic Interval* encodings for the environments created during the *nested-loop* evaluation are provided below.

In the *nested-loop* evaluation, the *for* expression "**for $o in ...**" is evaluated using the input environment $E^c = \{I_c, T_c\}$. The *Dynamic Interval* encodings for the resulting environment $E^n = \{I_n, T_c^n, T_o^n\}$ is as follows:

I_n
i
25
35
225
235

T_c^n ($w_c = 20$)		
s	l	r
id	501	504
id1	502	503
id	701	704
id1	702	703
id	4511	4514
id2	4512	4513
id	4711	4714
id2	4712	4713

T_o^n ($w_o^n = 20$)		
s	l	r
cid	505	508
id1	506	507
cid	715	718
id3	716	717
cid	4505	4508
id1	4506	4507
cid	4715	4718
id3	4716	4717

Then the *where* expression in the *merge-join* fragment is evaluated in the environment E^n . Only the environments from E^n that satisfy the equality comparison condition, **$c/text()=$o/text()**, are selected into the resulting environment $E^{n'} = \{I'_n, T_c^{n'}, T_o^{n'}\}$ for the remainder of the input query:

I'_n
i
25

$T_c^{n'}$ ($w_c^{n'} = 20$)		
s	l	r
id	501	504
id1	502	503

$T_o^{n'}$ ($w_o^{n'} = 20$)		
s	l	r
cid	505	508
id1	506	507

Comparing the environment E' generated by the *merge-join* approach with the environment $E^{n'}$ generated by the *nested-loop* evaluation, we can see that these two environments are identical. Proof of correctness for the *merge-join* approach is provided in the following section. □

Proof of Correctness for the Pattern

If we add one more constraint to the *for* expression in Nested-FOR Pattern, we obtain a more restricted but simpler version of Nested-FOR Pattern as follows:

```
FOR $x IN expr DO
WHERE pathexp[$x] = pathexp[$x_i] RETURN
  (WHERE pathexp[$x] = pathexp[$x_i] RETURN )*
```

where the expression *expr* in the *for* expression does not *depend on* any variables.

An example of the XQuery fragment that matches the more restricted version of Nested-FOR Pattern is shown in Example 4.3.6. The SQL translation for creating the indices pairs I_{pair} and the new environment E' for the remainder of the query that comes after the *merge-join* fragment for the restricted Nested-FOR Pattern can be simplified as follows:

```
CREATE VIEW I_pair AS
  SELECT op0.i AS i_in, op1.i AS i_x
  FROM   IT_xi^in op0, IT_x^x' op1
  WHERE  op0.s_1 = op1.s_1 AND ... AND op0.s_k = op1.s_k
```

The new environment E' is defined as follows:

```
CREATE VIEW I'(I_pair) AS
  SELECT i_in * w_x + i_x AS i
  FROM   I_pair
```

```
CREATE VIEW T'_{x_i}(I_pair, T_{x_i}) AS
  SELECT s, (i_in * w_x + i_x) * w_{x_i} + MOD(l, w_{x_i}) AS l,
            (i_in * w_x + i_x) * w_{x_i} + MOD(r, w_{x_i}) AS r
  FROM   I_pair, T_{x_i}
  WHERE  l/w_{x_i} = i_in
```

```
CREATE VIEW T'_x(I_pair, T_x) AS
  SELECT s, (i_in * w_x + i_x) * w_x + MOD(l, w_x) AS l,
            (i_in * w_x + i_x) * w_x + MOD(r, w_x) AS r
  FROM   I_pair, T_x
  WHERE  l/w_x = i_x
```

In order to prove the correctness of the *merge-join* approach for Nested-FOR Pattern, we first prove the correctness of the restricted version of Nested-FOR Pattern, which is simpler and easier to understand.

Proof 4.3.10 The following proof is provided to prove the correctness of the *merge-join* approach for the restricted version of Nested-FOR Pattern as follows:

```
FOR $x IN expr DO
WHERE pathexp[$x] = pathexp[$x_i]
 (AND pathexp[$x] = pathexp[$x_j])*
```

where the expression *expr* in the *for* expression does not *depend on* any variables.

Let $E_{in} = \{I_{in}, T_{x_1}, \ldots, T_{x_n}\}$ be the input environment for the *merge-join* fragment of the restricted Nested-FOR Pattern and $T_{x_1}, \ldots, T_{x_n}$ be the relational representations of variables $\$x_1, \ldots, \$x_n$ in E_{in}. Let $E^{n'} = \{I'_n, T^{n'}_{x_1}, \ldots, T^{n'}_{x_n}, T^{n'}_x\}$ be the resulting environment generated after evaluating the *merge-join* fragment using the *nested-loop* approach, where I'_n is the new environment index, $T^{n'}_{x_1}, \ldots, T^{n'}_{x_n}$ are the new representations of variables $\$x_1, \ldots, \$x_n$, and $T^{n'}_x$ is the representation of the variable $\$x$ provided by the *for* expression. Let $E' = \{I', T'_{x_1}, \ldots, T'_{x_n}, T'_x\}$ be the resulting environment generated by the *merge-join* approach. We prove that the environment E' and the environment $E^{n'}$ are identical, thus showing the correctness of the *merge-join* approach for the restricted Nested-FOR Pattern.

There are two major steps to prove that the two environments, E' and $E^{n'}$, are identical. We first prove that the two environments are constructed using equivalent queries. Then, we prove that these two environments contain the same data.

STEP 1: Queries Defining for Environment E' and $E^{n'}$.

1) Notation.

We firstly define notation that is needed in latter phases of the proof. Since the widths for the bound variables remain unchanged in different environments

generated during the evaluation process, $w_{x_1}, \ldots, w_{x_n}, w_x$ are used to represent the widths of the representations of variables $\$x_1, \ldots, \$x_n, \$x$ in different environments.

Since the expression *expr* in the *for* expression[13] "`FOR` $\$x$ `IN` *expr* `DO`" does not *depend on* any variables provided by E_{in}, *expr* can be evaluated using the initial environment $E_0 = \{I_0\}$. Let $T_e^0(s, l_e^0, r_e^0)$ be the resulting table of such evaluation and let $w_e = w_x$ be the width of T_e^0. Let $E'_x = \{I'_x, T_{x_1}^{x'}, \ldots, T_{x_m}^{x'}, T_x^{x'}\}$ be the environment obtained by evaluating the *for* expression in the environment E_0. Let $T_x^{x'}(s, l_x^{x'}, r_x^{x'})$ be the representation for variable $\$x$ in E'_x. We have the following queries for I'_x and $T_x^{x'}$:

$$
\begin{aligned}
I'_x &= \{l_e^0 : (s, l_e^0, r_e^0) \in Q_{roots}(T_e^0)\} \\
T_x^{x'} &= \{(s, l_{e_r}^0 * w_x + l_e^0, l_{e_r}^0 * w_x + r_e^0) : \\
&\quad (s, l_e^0, r_e^0) \in T_e^0 \ \wedge \ (s, l_{e_r}^0, r_{e_r}^0) \in Q_{roots}(T_e^0) \ \wedge \ r_{e_r}^0 \leq l_e^0 \ \wedge \ r_e^0 \leq l_{e_r}^0\}
\end{aligned}
$$

where $Q_{roots}(T_e^0)$ is the resulting table of the XQuery *roots* operation on T_e^0 and $Q_{roots}(T_e^0) \subseteq T_e^0$.

2) The computation of the environment $E^{n'}$ using the *nested-loop* strategy.

The *nested-loop* evaluation for the *merge-join* fragment follows the following steps:

- First, we evaluate the *for* expression "`FOR` $\$x$ `IN` *expr* `DO`" with the input environment E_{in}. Let $E^n = \{I_n, T_{x_1}^n, \ldots, T_{x_n}^n, T_x^n\}$ be the resulting environment after such evaluation.

  Let $T_e^{in}(s, l_e^{in}, r_e^{in})$ (with $w_e = w_x$) be the result of the expression *expr* evaluated in the environment E_{in}. Since the expression *expr* does not *depend on* any variable, the tuples in T_e^{in} and the tuples in T_e^0 satisfy the following relationship:

$$
T_e^{in} = \{(s, i_{in} * w_x + l_e^0, i_{in} * w_x + r_e^0) : (s, l_e^0, r_e^0) \in T_e^0 \ \wedge \ i_{in} \in I_{in}\}
$$

  Based on the above analysis and the SQL translation templates for the *for* expressions, we have the following relational representation for the environment E^n constructed by evaluating the second *for* loop in the pattern:

[13] For simplicity, if not specifically stated, the *expressions* we mention in the proof refer to the expressions that appear as part of the pattern.

$$
\begin{aligned}
I_n &= \{l_e^{in} : (s, l_e^{in}, r_e^{in}) \in Q_{roots}(T_e^{in})\} \\
&= \{i_{in} * w_e + l_e^0 : i_{in} \in I_{in} \ \wedge \ (s, l_e^0, r_e^0) \in Q_{roots}(T_e^0)\} \\
&= \{i_{in} * w_x + i'_x : i_{in} \in I_{in} \ \wedge \ i'_x \in I'_x\} \\
T_{x_i}^n &= \{(s, l_e^{in} * w_{x_i} + \mathsf{MOD}(l_{x_i}, w_{x_i}), l_e^{in} * w_{x_i} + \mathsf{MOD}(r_{x_i}, w_{x_i})) : \\
&\quad (s, l_e^{in}, r_e^{in}) \in Q_{roots}(T_e^{in}) \ \wedge \ (s, l_{x_i}, r_{x_i}) \in T_{x_i} \ \wedge \ l_{x_i}/w_{x_i} = l_e^{in}/w_e\} \\
&= \{(s, (i_{in} * w_x + i'_x) * w_{x_i} + \mathsf{MOD}(l_{x_i}, w_{x_i}), \\
&\quad (i_{in} * w_x + i'_x) * w_{x_i} + \mathsf{MOD}(r_{x_i}, w_{x_i})) : \\
&\quad i_{in} \in I_{in} \ \wedge \ i'_x \in I'_x \ \wedge \ (s, l_{x_i}, r_{x_i}) \in T_{x_i} \ \wedge \ l_{x_i}/w_{x_i} = i_{in}\} \\
T_x^n &= \{(s, l_{e_r}^{in} * w_e + \mathsf{MOD}(l_e^{in}, w_e), l_{e_r}^{in} * w_e + \mathsf{MOD}(r_e^0, w_e)) : \\
&\quad (s, l_{e_r}^{in}, r_{e_r}^{in}) \in Q_{roots}(T_e^{in}) \ \wedge \ (s, l_e^{in}, r_e^{in}) \in T_e^{in} \ \wedge \ r_{e_r}^{in} \leq l_e^{in} \ \wedge \ r_e^{in} \leq l_{e_r}^{in}\} \\
&= \{(s, (i_{in} * w_x + l_{e_r}^0) * w_x + l_e^0, (i_{in} * w_x + i'_x) * w_x + r_e^0) : \\
&\quad i_{in} \in I_{in} \ \wedge \ i'_x \in I'_x \ \wedge \ (s, l_e^0, r_e^0) \in T_e^0 \ \wedge \ (s, l_{e_r}^0, r_{e_r}^0) \in Q_{roots}(T_e^0)\} \\
&= \{(s, (i_{in} * w_x + i'_x) * w_x + \mathsf{MOD}(l_x^{x'}, w_x), \\
&\quad (i_{in} * w_x + i'_x) * w_x + \mathsf{MOD}(r_x^{x'}, w_x)) : \\
&\quad i_{in} \in I_{in} \ \wedge \ i'_x \in I'_x \ \wedge \ (s, l_x^{x'}, r_x^{x'}) \in T_x^{x'} \ \wedge \ l_x^{x'}/w_x = i'_x\}
\end{aligned}
$$

where $Q_{roots}(T_e^0)$ is the resulting tables of the XQuery *roots* operation on T_e^0, which is exactly the I'_x in environment E'_x.

- In the second phase the sequence of the *where* clauses are evaluated using the environment E^n. The *where* clauses filter out the unwanted environments within E^n and the remaining forms the resulting environment $E^{n'}$, which is a subset of environment E^n:

$$
\begin{aligned}
I'_n &= \{i_n : i_n \in I_n \ \wedge \ Q_\varphi(T_{x_1}^n, \ldots, T_{x_n}^n, T_x^n)\} \\
T_{x_i}^{n'} &= \{s, l, r) : i \in I_{n'} \ \wedge \ (s, l, r) \in T_{x_i}^n \ \wedge \ l/w_{x_i} = i\} \\
T_x^{n'} &= \{s, l, r) : i \in I'_n \ \wedge \ (s, l, r) \in T_x^n \ \wedge \ l/w_x = i\}
\end{aligned}
$$

where $Q_\varphi(T_{x_1}^n, \ldots, T_{x_n}^n, T_x^n)$ are the equality comparison conditions from *where* clauses.

Figure 4.3 shows the process of the *nested-loop* evaluation for the pattern fragment.

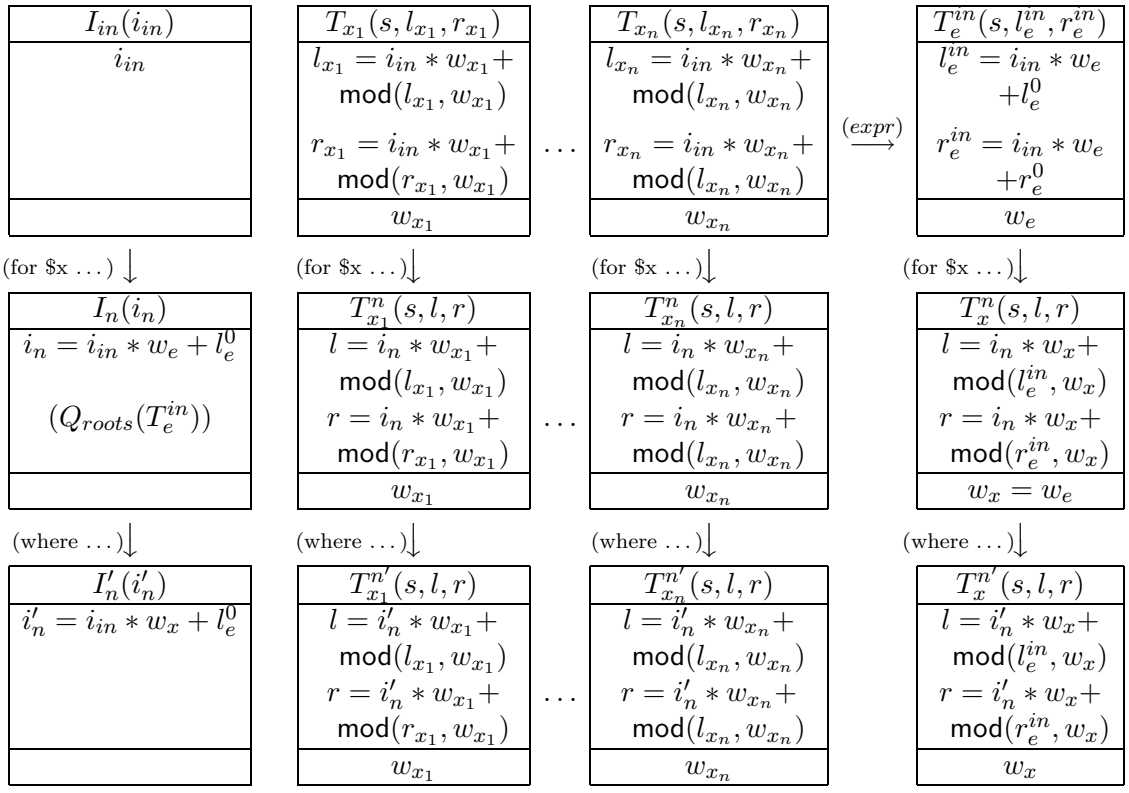


Figure 4.3: Translation of Restricted Nested-FOR Pattern Fragment Using *Nested-loop* Strategy

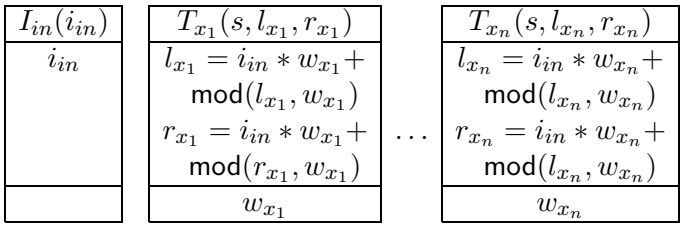
2) The computation of the environment E' using the *merge-join* strategy.

Based on the *merge-join* approach for the restricted Nested-FOR Pattern, we have the following representation for the resulting environment $E' = \{I', T'_{x_1}, \ldots, T'_{x_n}, T'_x\}$:

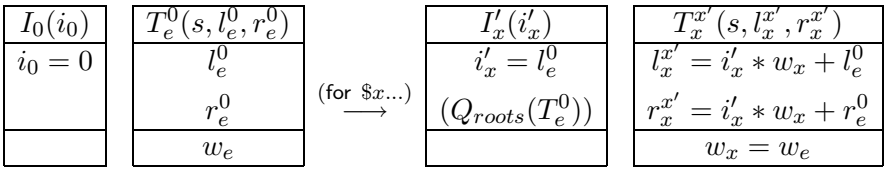
$$
\begin{aligned}
I' &= \{i_{in} * w_x + i_x : \ (i_{in}, i_x) \in I_{pair}\} \\
T'_{x_i} &= \{(s, (i_{in} * w_x + i_x) * w_{x_i} + \mathsf{MOD}(l_{x_i}, w_{x_i}), \\
&\quad (i_{in} * w_x + i_x) * w_{x_i} + \mathsf{MOD}(r_{x_i}, w_{x_i})) : \\
&\quad (i_{in}, i_x) \in I_{pair} \ \wedge \ (s, l_{x_i}, r_{x_i}) \in T_{x_i} \ \wedge \ l_{x_i}/w_{x_i} = i_{in}\} \\
T'_x &= \{(s, (i_{in} * w_x + i_x) * w_x + \mathsf{MOD}(l_x^{x'}, w_x), (i_{in} * w_x + i_x) * w_x \\
&\quad + \mathsf{MOD}(r_x^{x'}, w_x)) : \ (i_{in}, i_x) \in I_{pair} \ \wedge \ (s, l_x^{x'}, r_x^{x'}) \in T_x^{x'} \ \wedge \ l_x^{x'}/w_x = i_x\}
\end{aligned}
$$

Figure 4.4 shows the *merge-join* evaluation process for the restricted version of Nested-FOR Pattern.

1. The input environment for the *merge-join* pattern.



2. Evaluate "for $\$x \in expr$ do" in the *merge-join* pattern using initial environment E_0.



3. Evaluate the *where* clauses using the *merge join* operation.

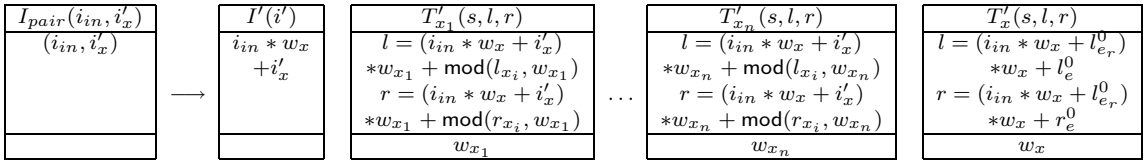


Figure 4.4: Translation of Restricted Nested-FOR Pattern Fragment Using *Merge-join* Strategy

3) Based on the analysis in 1) and 2), we can see that the environment $E^{n'}$ created by the *nested-loop* evaluation and the environment E' created by the *merge-join* approach are defined by the same query expressions.

STEP 2: The Data Sets in Environment E' and $E^{n'}$.

Based on the above queries derived for the environment E' and the environment $E^{n'}$, we can see that the difference between the E' and $E^{n'}$ is determined by the difference of the index values in $I' \in E'$ and $I'_n \in E^{n'}$. If we prove that these two index sets contain the same set of values, then the two environments are identical.

We first prove that any value in the index set I'_n generated by the *nested-loop* evaluation always appears in the index set I' generated by the *merge-join* approach.

Let $i'_{n_i} \in I'_n$. Then there must exist an index value $i_{in_i} \in I_{in}$ and an index value $i'_{x_i} \in I'_x$ that satisfy $i'_{n_i} = i_{in_i} * w_x + i'_{x_i}$. For the path expressions in the *where* clauses the resulting values corresponding to i_{in_i} or i'_{x_i} must satisfy the equality comparison conditions in the *where* clauses. We prove that the pair (i_{in_i}, i'_{x_i}) is a tuple in the table I_{pair} generated by the *merge-join* approach.

We know that the pairs I_{pair} are created by joining the two environment index sets I_{in} and I'_x with join conditions corresponding to the equality comparison conditions in the *where* clauses. Since the pair of the index values i_{in_i} and i'_{x_i} are from I_{in} and I'_x, respectively, we need to prove that the corresponding values of the path expressions in the *where* clauses satisfy the equality comparison conditions. As we have already shown in the previous analysis, the values of the path expressions that correspond to either the index value i_{in_i} or i'_{x_i} satisfy the equality comparison conditions in the *where* clauses; thus, (i_{in_i}, i'_{x_i}) is a tuple in I_{pair}. The resulting index value generated from the pair is $i'_i = i_{in_i} * w_x + i'_{x_i}$, which is exactly the same as i'_{n_i}.

Similarly, we can prove that for any index value i'_i in the index set I', there always exist one and only one index value i'_{n_i} from I'_n.

For an index value $i'_i \in I'$, there must exist an indices pair (i_{in_i}, i'_{x_i}) in I_{pair}, where $i'_i = i_{in_i} * w_x + i'_{x_i}$. The resulting values of the path expressions that corresponds to one of the index values in the indices pair must satisfy the equality comparison conditions in the *where* clauses. Based on the queries we derive from the *nested-loop* evaluation, there exist one and only one index value i'_{n_i} from I'_n where $i'_{n_i} = i_{in_i} * w_x + i'_{x_i}$.

From above, we prove that the two index sets I'_n and I' contain exactly the same set of values. Since the queries for constructing the environments $E^{n'}$ and E' are equivalent, we prove that these two environments are identical. □

Using similar approach, we can prove the correctness of the *merge-join* approach for Nested-FOR Pattern. The proof for Nested-FOR Pattern is omitted in the thesis.

4.3.3 The Multi-FOR Pattern

The Pattern

Pattern 4.3.2 (Multi-FOR) The XQuery fragments that follow the following XQuery pattern can utilize the *merge-join* approach:

```
FOR $f_1 IN expr DO
 (FOR $f_i IN expr DO)+
 (WHERE pathexp[$f_i] = pathexp[$f_j] RETURN)+
```

where *expr* represents any XQuery expression and "+" means one or more matches. The variables $\$f_i$ and $\$f_j$ represent two different variables bound by the *for* expressions in the pattern fragment. $pathexp[\$f_i]$ and $pathexp[\$f_j]$ represent path expressions that only depend on variables $\$f_i$ and $\$f_j$ respectively.

The pattern imposes the following constraints:

- The pattern starts with a list of nested *for* expressions, which are directly followed by a set of *where* clauses with no other expressions intervening.
- Let $\{\$f_1, \ldots, \$f_m\}$ $(m \geq 2)$ be the list of variables introduced by the nested *for* loops in the pattern. These variables are *independent* of each other. In other words, the *for* expressions in the pattern are independent of each other.
- The conditions in *where* clauses are equality comparisons, whose two operands are path expressions depend on different variables from $\{\$f_1, \ldots, \$f_m\}$. □

A sample XQuery fragment that matches Multi-FOR Pattern is shown in Example 4.3.11.

Example 4.3.11 The following fragment in an XQuery query follows Multi-FOR Pattern:

```
...
    for $seller in doc("users.xml")//user_tuple,
        $buyer in  doc("users.xml")//user_tuple,
        $item in  doc("items.xml")//item_tuple,
        $highbid in  doc("bids.xml")//bid_tuple
    where $seller/userid  = $item/offered_by
      and $item/itemno  = $highbid/itemno
      and $highbid/userid  = $buyer/userid
...
```

This fragment can be replaced with a *merge-join* expression by the *XQuery Optimizer* and be evaluated using the *merge-join* approach. □

The Merge-join Expression

An XQuery fragment that follows Multi-FOR Pattern is replaced with one single *merge-join* expression named *MultiFOR* in the form as follows:

$$\texttt{MultiFOR}[conditions](forloops)$$

where *conditions* is a list of the conditions from the *where* clauses, and $forloops$ lists the independent *for* expressions in the original order. The *MultiFOR* expression is added to the *Minimal XQuery* syntax as a new kind of expression. The *merge-join* expression for the *merge-join* fragment in Example 4.3.11 is shown below:

$$\begin{aligned}\texttt{MultiFOR}[\ & \$seller/userid = \$item/offered\_by,\\ & \$item/itemno = \$highbid/itemno,\\ & \$highbid/userid = \$buyer/userid\ ]\\ (\ & \$seller\ \texttt{in document}(''users.xml'')//user\_tuple,\\ & \$buyer\ \texttt{in document}(''users.xml'')//user\_tuple,\\ & \$item\ \texttt{in document}(''items.xml'')//item\_tuple,\\ & \$highbid\ \texttt{in document}(''bids.xml'')//bid\_tuple\ )\end{aligned}$$

Translation of the Pattern

Let $E_{in} = \{I_{in}, T_{x_1}, \ldots, T_{x_n}\}$ be the input environment for the whole fragment of Multi-FOR Pattern. Let $\{\$f_1, \ldots, \$f_m\}$ $(m \geq 2)$ be the list of variables introduced by the nested *for* loops in the pattern. The variables are listed in the order of the appearance of their corresponding *for* expressions in the fragment. The evaluation steps are as follows:

Step 1. Evaluate the FOR Expressions Independently:

The *for* expressions in the pattern are evaluated in the input environment E_{in} independently. Let $E_{f_1} = \{I_{f_1}, T^{f_1}_{f_1}, \ldots\}, \ldots, E_{f_m} = \{I_{f_m}, T^{f_m}_{f_m}, \ldots\}$ be the resulting environments. The new representation for the bound variables provided by the environment E_{in} are omitted since they are not relevant to the *merge-join* evaluation. Among the resulting environments, $T^{f_1}_{f_1}, \ldots, T^{f_m}_{f_m}$ (with corresponding widths $w_{f_1}, \ldots, w_{f_m}$) are the representations of variables $\$f_1, \ldots, \$f_m$.

Step 2. Evaluate the Path Expressions in the WHERE Expressions:

Based on the constraints for the pattern, the path expressions from the *where* clauses only depend on the variables $\$f_1, \ldots, \$f_m$. Let $pathexp_1[\$f_i], \ldots, pathexp_{k_i}[\$f_i]$ be the path expressions from the *where* clauses and each of them *depends on* a variable $\$f_i \in \{\$f_1, \ldots, \$f_m\}$. These path expressions are evaluated in the environment E_{f_i} and produce a set of resulting tables $T^{f_i}_{p_1}, \ldots, T^{f_i}_{p_{k_i}}$. These tables, along with the index set I_{f_i}, are joined into one table IT^{f_i}.

Step 3. Evaluate the WHERE Expressions Using the Merge Join:

By evaluating all the path expressions in *where* clauses, we obtain m sets of environments $\{I_{f_1}, T^{f_1}_{p_1}, \ldots, T^{f_1}_{p_{k_1}}\}, \ldots, \{I_{f_m}, T^{f_m}_{p_1}, \ldots, T^{f_m}_{p_{k_m}}\}$, which correspond to the variables $\$f_1, \ldots, \$f_m$. For a variable $\$f_i \in \{\$f_1, \ldots, \$f_m\}$ that does not appear in the path expressions in the *where* clauses, its corresponding environment just includes the corresponding index sets I_{f_i}. Using the *scalar fulselect* expression, these environments are joined into tables $IT^{f_1}, \ldots, IT^{f_m}$, respectively.

Let $IT^{f_i} \in \{IT^{f_1}, \ldots, IT^{f_m}\}$ be the table generated by joining the environment $\{I_{f_i}, T^{f_i}_{p_1}, \ldots, T^{f_i}_{p_{k_i}}\}$. IT^{f_i} contains attributes $(i_{f_i}, s^{f_i}_1, \ldots, s^{f_i}_{k_i})$ with one column for index I_{f_i} and a column for node labels $s^{f_i}_j$ for each table $T^{f_i}_{p_j} \in \{T^{f_i}_{p_1}, \ldots, T^{f_i}_{p_{k_i}}\}$. Let $w^{f_i}_{p_j}$ be the width of $T^{f_i}_{p_j}$. The SQL translation that uses the *scalar fulselect* expression to construct the table IT^{f_i} is as follows:

```
CREATE VIEW IT^{f_i}(I_{f_i}, T^{f_i}_{p_1}, ..., T^{f_i}_{p_{k_i}}) AS
  SELECT i_{f_i},
         ( SELECT s FROM T^{f_i}_{p_1} WHERE l/w^{f_i}_{p_1} = I_{f_i}.i ) AS s^{f_i}_1,
           ⋮
         ( SELECT s FROM T^{f_i}_{p_{k_i-1}} WHERE l/w^{f_i}_{p_{k_i-1}} = I_{f_i}.i ) AS s^{f_i}_{k_i-1},
         ( SELECT s FROM T^{f_i}_{p_{k_i}} WHERE l/w^{f_i}_{p_{k_i}} = I_{f_i}.i ) AS s^{f_i}_{k_i}
  FROM I_{f_i}
```

Using the tables $IT^{f_1}, \ldots, IT^{f_m}$, the equality comparison conditions in the *where* clauses can be evaluated by joining these tables together with join conditions correspond to the *where* conditions. As a result, a relation $I_{list}(i_{f_1}, \ldots, i_{f_m})$, which contains sets of indices for the matching environments from $E_{f_1}, \ldots, E_{f_m}$, is extracted from the result of the join. We have the following SQL translation template for creating the set of indices based on the *where* conditions:

```
CREATE VIEW I_list AS
  SELECT i_{f_1}, ..., i_{f_m}
  FROM   IT^{f_1}, ..., IT^{f_m}
  WHERE  i_{f_1}/w_{f_1} = i_{f_2}/w_{f_2} AND i_{f_1}/w_{f_1} = i_{f_3}/w_{f_3} ... AND i_{f_1}/w_{f_1} = i_{f_m}/w_{f_m}
    AND  s_l^{f_i} = s_k^{f_j} AND ...
```

where, for each equality comparison condition $pathexp_l[\$f_i] = pathexp_k[\$f_j]$ in the *where* clause, there is a corresponding condition $s_l^i = s_k^j$ in the SQL translation. In the SQL template for I_{list}, the strategy for joining the tables $IT^1, \ldots, IT^m$ are determined by the query processor in RDBMS. Since the RDBMS might not be able to take advantage of the ordering nature of the data, the preferable *merge join* operation might not be used in the query evaluation in RDBMS.

Unlike the SQL translation approach, in the physical plan translation for the *XQuery-enhanced Relational Engine* [15], preferable ***merge join*** operations can be specifically chosen to join the tables $IT^{f_1}, \ldots, IT^{f_m}$. There are many choices to join these tables using *merge join* operations. The choice of an optimal join order selection falls into the traditional relational query optimization problems for join operations, which is not in the scope of the thesis. Here, we can just pick one of the existing *join order selection* algorithms from the literature.

Step 4. Construct the New Environment:

Given the table $I_{list}(i_{f_1}, \ldots, i_{f_m})$ together with the input environment E_{in} for the pattern fragment and the environments $E_{f_1}, \ldots, E_{f_m}$ created by evaluating the *for* expressions, we are able to construct the input environment for the remainder of the query. The generated environment is identical to that we would have obtained by using the *nested-loop* evaluation.

Let $I', T'_{x_1}, \ldots, T'_{x_n}, T'_{f_1}, \ldots, T'_{f_m}$ be the resulting environment produced by the *merge-join* approach, where I' is the new environment index, $T'_{x_1}, \ldots, T'_{x_n}$ are the new representations of the local variables provided by the environment E_{in} and $T'_{f_1}, \ldots, T'_{f_m}$ are the representations of the variables $\$f_1, \ldots, \$f_m$ provided by the *for* expressions in the pattern. Let w_{x_i} be the width of $T_{x_i} \in \{T_{x_1}, \ldots, T_{x_n}\}$ from E_{in}. The new environment is defined as follows:

```
CREATE VIEW I'(I_list) AS
  SELECT (...(i_f1 * w_f2 + MOD(i_f2, w_f2)) * ...) * w_fm + MOD(i_fm, w_fm) AS i
  FROM   I_list
```

```
CREATE VIEW T'_xi(I_list, T_xi) AS
  SELECT s, ((...(i_f1 * w_f2 + MOD(i_f2, w_f2)) * ...) * w_fm + MOD(i_fm, w_fm)) * w_xi
              + MOD(l, w_xi) AS l,
            ((...(i_f1 * w_f2 + MOD(i_f2, w_f2)) * ...) * w_fm + MOD(i_fm, w_fm)) * w_xi
              + MOD(r, w_xi) AS r
  FROM   I_list, T_xi
  WHERE  i_f1 / w_f1 = l / w_xi
```

In the above SQL template for T'_{x_i}, the input table I_{list} can be substituted by the new environment index set I' generated using I_{list}. Thus, we have an alternative SQL template for T'_{x_i}, which is shown as follows:

```
CREATE VIEW T'_xi(I', T_xi) AS
  SELECT s, i * w_xi + MOD(l, w_xi) AS l, i * w_xi + MOD(r, w_xi) AS r
  FROM   I', T_xi
  WHERE  (...(i / w_fm) / ...) / w_f1 = l / w_xi
```

```
CREATE VIEW T'_fi(I_list, T_fi^fi) AS
  SELECT s, ((...(i_f1 * w_f2 + MOD(i_f2, w_f2)) * ...) * w_fm + MOD(i_fm, w_fm)) * w_fi
              + MOD(l, w_fi) AS l,
            ((...(i_f1 * w_f2 + MOD(i_f2, w_f2)) * ...) * w_fm + MOD(i_fm, w_fm)) * w_fi
              + MOD(r, w_fi) AS r
  FROM   I_list, T_fi^fi
  WHERE  i_fi = l / w_fi
```

In the above SQL template for T'_{f_i}, the input table I_{list} can be substituted by the new environment index set I'. Thus, we have an alternative SQL template for T'_{f_i}, which is shown as follows:

```
CREATE VIEW T'_{f_i}(I', T^{f_i}_{f_i}) AS
  SELECT  s, i * w_{f_i} + MOD(l, w_{f_i}) AS l, i * w_{f_i} + MOD(r, w_{f_i}) AS r
  FROM    I', T^{f_i}_{f_i}
  WHERE   (...(i/w_{f_m})/...)/w_{f_1} = (l/w_{f_i})/w_{f_i}
```

The new environment generated using the above SQL templates is exactly the same as that generated using *nested-loop* approach introduced in Chapter 3.

Proof of Correctness for the Pattern

The approach to prove the correctness of the *merge-join* approach for Multi-FOR Pattern is similar to that provided for the restricted Nested-FOR Pattern. Details of the proof are omitted in the thesis.

4.4 XQuery Rewriting Rules

The *merge-join* patterns introduced in previous sections are general enough to capture a significant fraction of the *merge-join* fragments. In order to capture a larger fragment of XQuery that can utilize the *merge-join* approach, several XQuery rewriting rules are designed to transform the input XQuery into an equivalent XQuery which might contain a larger number of fragments that match the *merge-join* patterns.

There are two types of rewriting rules. One type of rewriting rules are designed to help the emergence of the *merge-join* fragments for Nested-FOR Pattern; the other help the appearance of the *merge-join* fragments for Multi-FOR Pattern.

4.4.1 Rewriting Rules for Nested-FOR Pattern

We design two rewriting rules for Nested-FOR Pattern: one is *Nested-FOR LET Relocating Rule* that relocates the *let* expressions in the input query; the other is *Nested-FOR WHERE Relocating Rule* that relocates the *where* expressions in the input query.

I. The LET Expression Relocating Rule.

Rewriting Rule 4.4.1 (Nested-FOR LET Relocating Rule) This rewriting rule helps the emergence of the *merge-join* fragments for Nested-FOR Pattern by relocating the *let* expressions in the input query.

The query rewriting process introduced below is applied repeatedly to each *let* expression in the input query until all *let* expressions are checked:

The input query is viewed as an expression tree. Starting from a node of the *let* expression "let $\$x := expr$ in" in the expression tree, the expression tree is traversed in the child-to-parent direction until a *for* expression is found. If the variable bound by the *let* expression does not *depend on* the variable bound by the *for* expression, the *let* expression is relocated to the top of the *for* expression. The above traverse process is continued until a *for* expression that does not satisfy the above condition is met. □

Based on the semantic of the *let* expression, such a query rewriting produces a query equivalent to the original query. An example that utilizes such a rewriting rule is shown in the following example.

Example 4.4.1 For the following input query:

```
...
for $item in doc("items.xml")//item_tuple
let $bid_counts :=
    for $i in distinct-values(doc("bids.xml")//itemno)
    let $b := doc("bids.xml")//bid_tuple[itemno = $i]
    return
      <bid_count>
        <itemno>{ $i }</itemno>
        <nbids>{ count($b) }</nbids>
      </bid_count>
for $bc in $bid_counts
where $item/itemno = $bc/itemno
return ...
```

There is one *merge-join* fragment (shown in boldface) that matches Nested-FOR Pattern. However, since variable $\$bc$ *depends on* variable $\$bid\_counts$, which is bound by the *let* expression nested within the first *for* expression, the *for* expression in the *merge-join* fragment is evaluated within the iteration of variable $\$item$. By applying the *Nested-FOR LET Relocating Rule*, the *let* expression (shown in italic)

in the query is relocated to the top of the *for* expression "`for $item in ...`". The rewritten query is as follows:

```
...
let $bid_counts :=
      for $i in distinct-values(doc("bids.xml")//itemno)
      let $b := doc("bids.xml")//bid_tuple[itemno = $i]
      return
        <bid_count>
           <itemno>{ $i }</itemno>
           <nbids>{ count($b) }</nbids>
        </bid_count>
for $item in doc("items.xml")//item_tuple
for $bc in $bid_counts
where $item/itemno = $bc/itemno
return ...
```

After the rewriting, the two *for* expressions in the query can be evaluated independently and the efficient *merge-join* approach can be utilized to evaluate the fragment. □

II. The WHERE Expression Relocating Rule.

Rewriting Rule 4.4.2 (Nested-FOR WHERE Relocating Rule) This rewriting rule helps the emergence of the *merge-join* fragments for Nested-FOR Pattern by relocating the *where* expressions in the input query.

Let e_{where} denote a *where* expression of the form "where $pathexpr[\$x] = pathexpr[\$y]$ return ...". The path expressions $pathexpr[\$x]$ and $pathexpr[\$y]$ from the *where* condition depend on two different variables provided by two *for* expressions in the input query. Let e_{for_1} and e_{for_2} be such two *for* expressions and e_{for_2} appears in the scope of e_{for_1}. Each *where* expression e_{where} in the input query is relocated to the position that directly follows the *for* expression e_{for_2}. □

Based on the semantic of the *where* expression, the above query rewritten process produces equivalent queries to the original ones. Examples that utilize such a rewriting rule are shown below.

Example 4.4.2 The following input query "*List the names of persons and the names of the items they bought in Europe.*" is from XMark Q9 [1]:

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site
                  /closed_auctions/closed_auction
          let $n := for $t2 in document("auction.xml")
                                  /site/regions/europe/item
                    where $t/itemref/@item = $t2/@id
                    return $t2
          where $p/@id = $t/buyer/@person
          return <item> $n/name/text() </item>
return <person name=$p/name/text()> $a </person>
```

There are two *merge-join* fragments (shown in boldface and italic, respectively) matching Nested-FOR Pattern. However, only the fragment shown in boldface can be identified using the two *merge-join* patterns. By applying the *Nested-FOR WHERE Relocating Rule*, the query is transformed into an equivalent query as follows:

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site
                  /closed_auctions/closed_auction
          where $p/@id = $t/buyer/@person
          return
          let $n := for $t2 in document("auction.xml")
                                  /site/regions/europe/item
                    where $t/itemref/@item = $t2/@id
                    return $t2
          return <item> $n/name/text() </item>
return <person name=$p/name/text()> $a </person>
```

where two fragments that match Nested-FOR Pattern are identified. □

Example 4.4.3 For the input query:

```
for $p in document("patient.xml")//patient,
    $x in ( for $r in document("record.xml")//record,
                $e in $r/entry
            where $r/patientSSNo=$p/Ssno and $e/diagnosis="flu"
            return <res>
                      $p/name,
                      <occured>$e/@date</occured>
                   </res> )
return $x
```

There is one *merge-join* fragment (shown in boldface) that can utilize the *merge-join* approach. However, this fragment can not be identified using the two *merge-join* patterns. By applying the *Nested-FOR WHERE Relocating Rule*, the query is transformed to an equivalent query as follows:

```
for $p in document("patient.xml")//patient,
    $x in ( for $r in document("record.xml")//record
            where $r/patientSSNo=$p/Ssno
            return
               for $e in $r/entry
               where e/diagnosis="flu"
               return <res>
                         $p/name,
                         <occured>$e/@date</occured>
                      </res> )
return $x
```

where a fragment that matches Nested-FOR Pattern is identified. □

4.4.2 Rewriting Rules for Multi-FOR Pattern

Similar to the rewriting rules for Nested-FOR Pattern, we design two rewriting rules for Multi-FOR Pattern: the first is *Multi-FOR LET Relocating Rule* that relocates the *let* expressions and the other one is *Multi-FOR WHERE Relocating Rule* that relocates the *where* expressions in the input query.

I. The LET Expression Relocating Rule.

Rewriting Rule 4.4.3 (Multi-FOR LET Relocating Rule) This rewriting rule helps the emergence of the *merge-join* fragments for Multi-FOR Pattern by relocating the *let* expressions in the input query.

Let "let $\$x$:= *expr* in ..." be a *let* expression whose subexpression *expr* does not depend on any variable. If there is such a *let* expression in the input query, the *let* expression is repeatedly relocated to the top of its parent nodes until all of the *for* expressions in the input query are the descendants of such a *let* expression. □

Based on the semantic of the *let* expression, the rewritten query produces exactly the same results as the original query. A query fragment that utilizes such a rewriting rule is shown in the following example.

Example 4.4.4 Consider the following fragment:

```
for $seller in doc("users.xml")//user_tuple,
    $buyer in doc("users.xml")//user_tuple,
    $item in doc("items.xml")//item_tuple,
    $highbid in doc("bids.xml")//bid_tuple
let $bids = doc("bids.xml")//id in
where $seller/userid = $item/offered_by
  and $item/itemno = $highbid/itemno
  and $highbid/userid = $buyer/userid
return
   ...
```

There is one *merge-join* fragment (shown in boldface) that can utilize the *merge-join* approach but can not be identified. By applying *Multi-FOR LET Relocating Rule*, the query is rewritten into an equivalent query as follows:

```
let $bids = doc("bids.xml")//id in
for $seller in doc("users.xml")//user_tuple,
    $buyer in doc("users.xml")//user_tuple,
    $item in doc("items.xml")//item_tuple,
    $highbid in doc("bids.xml")//bid_tuple
where $seller/userid = $item/offered_by
  and $item/itemno = $highbid/itemno
  and $highbid/userid = $buyer/userid
return
   ...
```

As a result of the rewriting, a fragment that matches Multi-FOR Pattern is identified. □

II. The WHERE Expression Relocating Rule.

Rewriting Rule 4.4.4 (Multi-FOR WHERE Relocating Rule) This rewriting rule helps the emergence of the *merge-join* fragments for Multi-FOR Pattern by relocating the *where* expressions in the input query.

Let e_{where} denote a *where* expression of the form "where $pathexpr[\$x] = pathexpr[\$y]$ return . . .". The path expressions $pathexpr[\$x]$ and $pathexpr[\$y]$ both depend on the variables provided by the *for* expressions in the input query.

If there exist a sequence of consecutive *where* expressions in the query, *where* expressions of the form as e_{where} in the sequence are relocated within the scope of the sequence in a way such that, in the resulting sequence, all of the *where* expressions that do not conform to the form as e_{where} are the descendant nodes of the *where* expressions of the form as e_{where}. □

Based on the semantic of the *where* expression, such a relocation produces an equivalent query to the original one. An example of applying *Multi-FOR WHERE Relocating Rule* to generate the fragment for Multi-FOR Pattern is shown as follows:

Example 4.4.5 The following query fragment is from the W3C Use Case [3]:

```
...
    for $seller in doc("users.xml")//user_tuple,
        $buyer in doc("users.xml")//user_tuple,
        $item in doc("items.xml")//item_tuple,
        $highbid in doc("bids.xml")//bid_tuple
    where $seller/name = "Tom Jones"
      and $seller/userid = $item/offered_by
      and contains($item/description , "Bicycle")
      and $item/itemno = $highbid/itemno
      and $highbid/userid = $buyer/userid
...
```

There is one *merge-join* fragment (shown in boldface) in the above query that cannot be identified. Applying the rewriting rule, the query is transformed into an equivalent query as follows:

```
...
    for $seller in doc("users.xml")//user_tuple,
        $buyer in doc("users.xml")//user_tuple,
        $item in doc("items.xml")//item_tuple,
        $highbid in doc("bids.xml")//bid_tuple
    where $seller/userid = $item/offered_by
      and $item/itemno = $highbid/itemno
      and $highbid/userid = $buyer/userid
      and $seller/name = "Tom Jones"
      and contains($item/description , "Bicycle")
...
```

After the rewriting, a fragment that matches Multi-FOR Pattern is identified. □

4.4.3 Discussion

The Multi-FOR Pattern overlaps with Nested-FOR Pattern. Since Multi-FOR Pattern is able to handle a larger number of independent *for* expressions at once, Multi-FOR Pattern is often a better choice. This situation is illustrated in the following example.

Example 4.4.6 Consider the query:

```
for $p in document("patient.xml")//patient,
    $r in document("record.xml")//record,
    $e in document("record.xml")//entry
where $r/patientSSNo=$p/Ssno and $e/diagnosis = "flu"
      and $e/diagnosis=$r/entry/diagnosis
return <res> $p/name, <occured>$e/@date</occured></res>
```

If *Multi-FOR WHERE Relocating Rule* is applied to the query, a *merge-join* fragment (shown in boldface) for Multi-FOR Pattern is created as follows:

```
for $p in document("patient.xml")//patient,
    $r in document("record.xml")//record,
    $e in document("record.xml")//entry
where $r/patientSSNo=$p/Ssno
  and $e/diagnosis=$r/entry/diagnosis
  and $e/diagnosis = "flu"
return <res> $p/name, <occured>$e/@date</occured></res>
```

If *Nested-FOR WHERE Relocating Rule* is applied, then the two fragments that match Nested-FOR Pattern (shown in boldface and italic, respectively) are recognized in the following rewritten query:

```
for $p in document("patient.xml")//patient,
    $r in document("record.xml")//record
where $r/patientSSNo=$p/Ssno
return
   for $e in document("record.xml")//entry
   where $e/diagnosis=$r/entry/diagnosis
     and $e/diagnosis = "flu"
return <res> $p/name, <occured>$e/@date</occured></res>
```

For this case, using *Multi-FOR WHERE Relocating Rule* to create a fragment that matches Multi-FOR Pattern is a better choice. □

For the XQuery queries provided in the W3C Use Cases [3] and the XMark Benchmark [1], Multi-FOR Pattern seems to be a better choice whenever there exist fragments that match both Nested-FOR Pattern and Multi-FOR Pattern. Based on this analysis, we propose a heuristic architecture for the *XQuery Optimizer* to perform the query rewriting and pattern identification, where fragments that match both two *merge-join* patterns are identified as the *merge-join* fragments for Multi-FOR Pattern.

4.5 The XQuery Optimizer

As mentioned before, an *XQuery Optimizer* is added to the XQuery processor to perform the query rewriting and pattern identification. The heuristic architecture for the *XQuery Optimizer* is shown in Figure 4.5.

The optimization process of the *XQuery Optimizer* is divided into two phases:

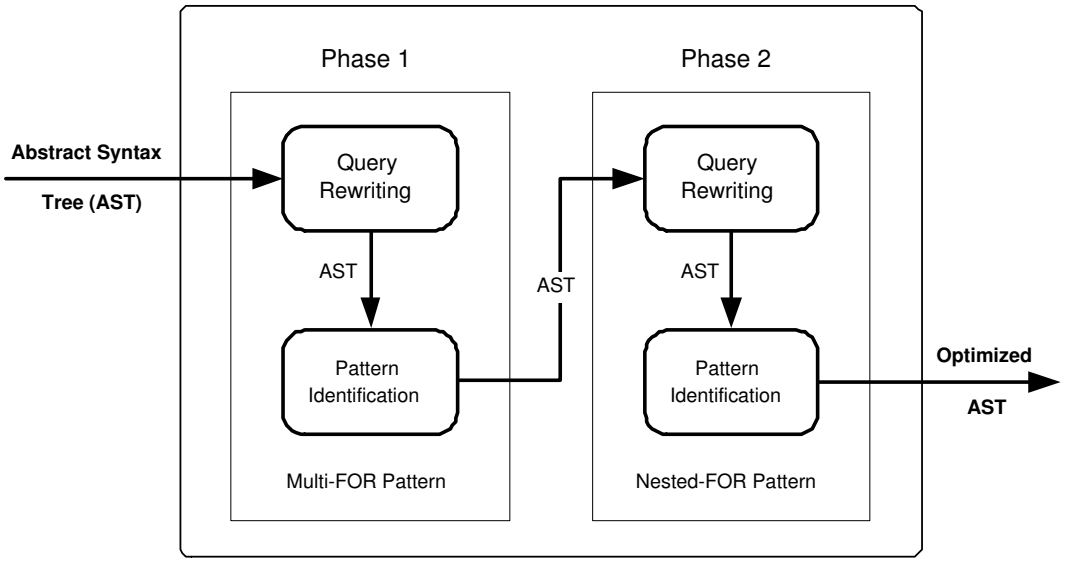


Figure 4.5: The XQuery Optimizer

- In the first phase, the rewriting rules for Multi-FOR Pattern are applied to the input *abstract syntax tree* produced by the XQuery parser. Based on the rewritten query, fragments that match Multi-FOR Pattern are identified and are rewritten as *merge-join* expressions for Multi-FOR Pattern (see Section 4.3.3). The rewritten *abstract syntax tree* is then passed to the second phase.

- Similar to the first phase, in the second phase, the rewriting rules for Nested-FOR Pattern are applied to the output of the first phase to perform the query rewriting. Then the fragments that match Nested-FOR Pattern are identified and are rewritten into *merge-join* expressions for Nested-FOR Pattern.

The output of the *XQuery Optimizer* is a refined *abstract syntax tree* that facilitates the use of the preferable *merge-join* approach in latter phase of the query translation. With the use of the *merge-join* approach, the XQuery processor is able to generate high quality relational plans for the input XQuery queries.

Chapter 5

Experiments

5.1 Experimental Setup

5.1.1 Methodology

The *XQuery-to-SQL Query Processor* we propose in the thesis is able to translate an input XQuery into a single SQL query. To justify the correctness of the translation, we choose twenty-six XQuery queries from the W3C Use Cases [3] and the XMark Benchmark [1] as test cases. We create the corresponding relational encoding tables in database systems for the XML source documents. The test cases are first rewritten in the *Minimal XQuery* syntax introduced in Section 2.2. Then, the rewritten queries are input into the query processor and translated into SQL queries. Finally, the generated SQL queries are executed in DB2 to obtain the relational encodings of the resulting XML forests. These resulting encodings can be easily decoded back to the form of XML forests by the *XML Decoder*.

The experiments conducted in the thesis focus on the regression test, which justifies the correctness of our work. The XQuery query processor is written in Java v1.4. The generated SQL queries are tested in DB2 v8.1.0.0.

5.1.2 XQuery Use Cases and XML Data

The use cases from W3C are created by the XML Query Working Group and are focused on different application areas. In the experiments, we first test the XQuery queries from the W3C Use Cases. We have several reasons: firstly, the example

input XML data are provided in the W3C Use Cases; secondly, the size of the input data is relatively small, thus it is more convenient to conduct the testing since our concern about the experiments is the correctness of the SQL queries; finally, the W3C Use Cases provides the results for each query, which enable us to check the correctness of the SQL plans that the query processor produces. The XQuery queries from the W3C Use Cases we use are in four categories: Use Case "XMP", Use Case "TREE", Use Case "SEQ" and Use Case "SGML" [3].

Queries from the XMark Benchmark [1] are also used in the experiments. The XML source documents "*auction.xml*" are generated by the data generator from the XMark Benchmark, using scale factors 0 (27KB), 0.001 (114KB), and 0.002 (207KB).

Before we test the generated SQL queries in the database systems, relational encoding tables must be created for the source XML documents. The attributes for the encoding tables are set as follows:

```
(  C1        varchar(maxlength),
   C2        bigint not null,
   C3        bigint not null  )
```

Attributes $C1$, $C2$, $C3$ are corresponding to the attributes (s, l, r) for the *interval encoding* introduced in Section 2.3.1. We use `bigint` as the data type for l and r because the left and right endpoint values of the resulting tuples can be very large based on the *Dynamic Interval Encoding* technique.

Relational tables UNIT and EMPTY are also constructed in the database systems. The UNIT table can be any kind of relational table that contains one and only one tuple. The EMPTY table is a table that has exactly the same schema as that of the relational encoding tables for the XML documents but contains no tuples.

5.2 Experiments

In this section, we only show part of the experimental results. The rest of the experiments can be found in Appendix B.

5.2.1 Experiment 1: W3C Use Case "TREE" Q4

The queries from the W3C Use Case "TREE" test the capability of extracting elements from very flexible structural documents while maintaining the original

hierarchy. We take the query $Q4$ from the Use Case as an input query in the experiment.

1. The Input XQuery Query

For the query Q4 "*How many top-level sections are in Book1?*" from the W3C Use Case "TREE", the XQuery solution from the W3C Use Cases is as follows:

```
<top_section_count>
{
  count(doc("book.xml")/book/section)
}
</top_section_count>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<top_section_count>
  count(select('section',children(select('book',document('book')))))
</top_section_count>
```

2. The Generated SQL Query

The corresponding SQL query generated by the XQuery query processor is shown in Figure 5.1. In the following experiments the generated SQL queries are omitted.

3. The Result of the SQL Query

We run the generated SQL query in DB2 and obtain the following relational encoding result:

```
C1                C2                   C3
----------------- -------------------- --------------------
top_section_count                    0                    3
2                                    1                    2

  2 record(s) selected.
```

```
WITH View0 AS
       (SELECT 0 AS C1 FROM
        UNIT T1
       ) ,
View1 AS
       (SELECT T1_Lf.C1, T1_Rg.C1 AS C2,
               T1_Rg.C2 AS C3, T1_Rg.C3 AS C4 FROM
        View0 T1_Lf,
        book T1_Rg
       ) ,
View2 AS
       (SELECT T1_Lf.C1 AS C1, T1_Lf.C2 AS C2,
               T1_Lf.C3 AS C3, T1_Lf.C4 AS C4 FROM
        View1 T1_Lf,
        (SELECT * FROM
         (SELECT * FROM
          View1 T3_Lf
         WHERE NOT EXISTS (
         SELECT * FROM
          View1 T3_Rg
         WHERE T3_Lf.C3>T3_Rg.C3 AND T3_Lf.C4<T3_Rg.C4
           AND T3_Lf.C1=T3_Rg.C1 )
        ) T2
        WHERE C2='book'
        ) T1_Rg
       WHERE T1_Rg.C3<=T1_Lf.C3 AND T1_Lf.C4<=T1_Rg.C4
         AND T1_Lf.C1=T1_Rg.C1
       ) ,
View3 AS
       (SELECT * FROM
        View2 T1_Lf
       WHERE NOT EXISTS (
       SELECT * FROM
        (SELECT * FROM
         View2 T2_Lf
        WHERE NOT EXISTS (
        SELECT * FROM
         View2 T2_Rg
        WHERE T2_Lf.C3>T2_Rg.C3 AND T2_Lf.C4<T2_Rg.C4
          AND T2_Lf.C1=T2_Rg.C1 )
        ) T1_Rg
       WHERE T1_Lf.C3=T1_Rg.C3 AND T1_Lf.C1=T1_Rg.C1 )
       ) ,
View4 AS
       (SELECT T1_Lf.C1 AS C1,T1_Lf.C2 AS C2,
               T1_Lf.C3 AS C3,T1_Lf.C4 AS C4 FROM
        View3 T1_Lf,
        (SELECT * FROM
         (SELECT * FROM
          View3 T3_Lf
         WHERE NOT EXISTS (
         SELECT * FROM
          View3 T3_Rg
          WHERE T3_Lf.C3>T3_Rg.C3 AND T3_Lf.C4<T3_Rg.C4
            AND T3_Lf.C1=T3_Rg.C1 )
         ) T2
        WHERE C2='section'
        ) T1_Rg
       WHERE T1_Rg.C3<=T1_Lf.C3 AND T1_Lf.C4<=T1_Rg.C4
         AND T1_Lf.C1=T1_Rg.C1
       ) ,
View5 AS
       (SELECT * FROM
        View4 T1_Lf
       WHERE NOT EXISTS (
       SELECT * FROM
        View4 T1_Rg
       WHERE T1_Lf.C3>T1_Rg.C3 AND T1_Lf.C4<T1_Rg.C4
         AND T1_Lf.C1=T1_Rg.C1 )
       )
( SELECT * FROM
       (SELECT C2 AS C1,C3+C1*(2+2) AS C2,
               C4+C1*(2+2) AS C3 FROM
        ((SELECT * FROM
         (SELECT C1,C2,C3+1 AS C3,C4+1 AS C4 FROM
          ((SELECT * FROM
           (SELECT C1, '0' AS C2,0 AS C3,1 AS C4 FROM
            (SELECT * FROM
             View0 T6_Lf
            WHERE NOT EXISTS (
            SELECT * FROM
             View5 T6_Rg
            WHERE T6_Lf.C1=T6_Rg.C1 )
            ) T5
           ) T4_Rg )
          UNION ALL
          (SELECT * FROM
           (SELECT C1, CHAR(COUNT(*)) AS C2,
                   0 AS C3, 1 AS C4 FROM
            View5 T7
           GROUP BY C1
           ) T4_Lf )
          ) T3
         ) T2_Rg )
        UNION ALL
        (SELECT * FROM
         (SELECT C1, 'top_section_count' AS C2,
                 0 AS C3, 2+1 AS C4 FROM
          (SELECT T9_Lf.C1 FROM
           View0 T9_Lf,
           UNIT T9_Rg
          ) T8
         ) T2_Lf )
        ) T1
       )  T_root
ORDER BY C2 )
```

Figure 5.1: The Generated SQL Query for Experiment 1

The resulting relational encoding can be easily transformed back to the following XML forest using the *XML Decoder*:

```
<top_section_count>
  <2>
  </2>
</top_section_count>
```

As mentioned in Chapter 2, by default, the XML documents discussed in the scope of the thesis are in the format of *abstract syntax XML forest*. The above resulting

XML forest can be easily converted back to the XML forest shown below, given additional encoding information for the attribute nodes and the text nodes:

```
<top_section_count>
  2
</top_section_count>
```

5.2.2 Experiment 2: W3C Use Case "TREE" Q5

1. The Input XQuery Query

For the query Q5 from the W3C Use Case "TREE": "*Make a flat list of the section elements in Book1. In place of its original attributes, each section element should have two attributes, containing the title of the section and the number of figures immediately contained in the section.*", the XQuery solution from the W3C Use Cases is as follows:

```
<section_list>
{
  for $s in doc("book.xml")//section
  let $f := $s/figure
  return
    <section title="{ $s/title/text() }" figcount="{ count($f) }"/>
}
</section_list>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<section_list>
  for $s in select('section',subtreesdfs(document('book')))do
  let $f := select('figure',children($s)) in
  <section>
      <title>
         children(select('title',children($s)))
      </title> @
      <figcount> count($f) </figcount>
  </section>
</section_list>
```

2. The Result of the SQL Query

The relational encoding result obtained by running the generated SQL plan for Q5 in DB2 is as follows:

```
C1                                      C2                  C3
----------------------------------- ------------------- -------------------
section_list                                          0           875449729
section                                        87030073            87059664
title                                          87030074            87059659
Introduction                                   87033026            87033027
figcount                                       87059660            87059663
0                                              87059661            87059662
section                                       174060145           174089736
title                                         174060146           174089731
Audience                                      174066031           174066032
figcount                                      174089732           174089735
0                                             174089733           174089734
section                                       225254305           225283896
title                                         225254306           225283891
Web Data and the Two Cultures                 225261921           225261922
figcount                                      225283892           225283895
1                                             225283893           225283894
section                                       404433865           404463456
title                                         404433866           404463451
A Syntax For Data                             404447544           404447545
figcount                                      404463452           404463455
1                                             404463453           404463454
section                                       614329921           614359512
title                                         614329922           614359507
Base Types                                    614350685           614350686
figcount                                      614359508           614359511
0                                             614359509           614359510
section                                       665524081           665553672
title                                         665524082           665553667
Representing Relational Databases             665546575           665546576
figcount                                      665553668           665553671
1                                             665553669           665553670
section                                       819106561           819136152
title                                         819106562           819136147
Representing Object Databases                 819134245           819134246
figcount                                      819136148           819136151
0                                             819136149           819136150

  36 record(s) selected.
```

The XML forest corresponding to the resulting relational encoding for this query

is as follows:

```
<section_list>
  <section>
    <title> <Introduction> </Introduction> </title>
    <figcount> <0> </0> </figcount>
  </section>
  <section>
    <title> <Audience> </Audience> </title>
    <figcount> <0> </0> </figcount>
  </section>
  <section>
    <title>
      <Web Data and the Two Cultures>
      </Web Data and the Two Cultures>
    </title>
    <figcount> <1> </1> </figcount>
  </section>
  <section>
    <title> <A Syntax For Data> </A Syntax For Data> </title>
    <figcount> <1> </1> </figcount>
  </section>
  <section>
    <title> <Base Types> </Base Types> </title>
    <figcount> <0> </0> </figcount>
  </section>
  <section>
    <title>
      <Representing Relational Databases>
      </Representing Relational Databases>
    </title>
    <figcount> <1> </1> </figcount>
  </section>
  <section>
    <title>
      <Representing Object Databases>
      </Representing Object Databases>
    </title>
    <figcount> <0> </0> </figcount>
  </section>
</section_list>
```

In the following experiments, we only show the query results in the form of decoded XML forests.

5.2.3 Experiment 3: W3C Use Case "XMP" Q1

The queries from this Use Case illustrate the samples of querying databases and document communities. We take the query $Q4$ from the Use Case as an input query in the experiment. It is a good example to show the capability of the query processor to handle the expressions that are composed of basic operations and *FLWR* expressions.

1. The Input XQuery Query

For the query Q1 "*List books published by Addison-Wesley after 1991, including their year and title.*", the XQuery solution from the W3C Use Cases is as follows:

```
<bib>
 {
  for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

The rewritten query in *Minimal XQuery* syntax is as follows:

```
<bib>
  for $b in select('book',children(select('bib',document('bib')))) do
  where not empty(
     select('addison-wesley',children(select('publisher',children($b)))))
  return
    where children(select('year',children($b))) > <1991>[]()</1991>
    return
      <book>
         <year> children(select('year',children($b))) </year>
         @ select('title',children($b))
      </book>
</bib>
```

2. The Result of the SQL Query

The resulting XML forest obtained by running the generated SQL plan in DB2 is as follows:

```
  <bib>
    <book>
      <year> <1994> </1994> </year>
      <title> <TCP/IP Illustrated> </TCP/IP Illustrated> </title>
    </book>
    <book>
      <year> <1992> </1992> </year>
      <title>
        <Advanced Programming in the Unix environment>
        </Advanced Programming in the Unix environment>
      </title>
    </book>
</bib>
```

5.2.4 Experiment 4: W3C Use Case "XMP" Q2

1. The Input XQuery Query

For the query Q2 "*Create a flat list of all the title-author pairs, with each pair enclosed in a "result" element.*" from the W3C Use Case "XMP", the XQuery solution from the W3C Use Cases is as follows:

```
<results>
  {
    for $b in doc("http://bstore1.example.com/bib.xml")/bib/book,
        $t in $b/title,
        $a in $b/author
    return
        <result>
            { $t }
            { $a }
        </result>
  }
</results>
```

The rewritten query in *Minimal XQuery* syntax is as follows:

```
<results>
  for $b in select('book',children(select('bib',document('bib')))) do
    for $t in select('title',children($b)) do
      for $a in select('author',children($b)) do
        <result>
          $t @ $a
        </result>
</results>
```

2. The Result of the SQL Query

The resulting XML forest of the SQL plan generated by the query processor is as follows:

```
<results>
  <result>
    <title> <TCP/IP Illustrated> </TCP/IP Illustrated> </title>
    <author>
      <last> <Stevens> </Stevens> </last>
      <first> <W.> </W.> </first>
    </author>
  </result>
  <result>
    <title>
      <Advanced Programming in the Unix environment>
      </Advanced Programming in the Unix environment>
    </title>
    <author>
      <last> <Stevens> </Stevens> </last>
      <first> <W.> </W.> </first>
    </author>
  </result>
  <result>
    <title> <Data on the Web> </Data on the Web> </title>
    <author>
      <last> <Abiteboul> </Abiteboul> </last>
      <first> <Serge> </Serge> </first>
    </author>
  </result>
  <result>
    <title> <Data on the Web> </Data on the Web> </title>
```

```
      <author>
        <last> <Buneman> </Buneman> </last>
        <first> <Peter> </Peter> </first>
      </author>
    </result>
    <result>
      <title> <Data on the Web> </Data on the Web> </title>
      <author>
        <last> <Suciu> </Suciu> </last>
        <first> <Dan> </Dan> </first>
      </author>
    </result>
</results>
```

5.2.5 Experiment 5: W3C Use Case "SEQ" Q1

The queries in this Use Case illustrate queries based on the order of elements in a document. We take the query $Q1$ from the Use Case as an input query in the experiment.

1. The Input XQuery Query

For the query Q1 "*In the Procedure section of Report1, what Instruments were used in the second Incision?* " from the W3C Use Case "SEQ", the XQuery solution from the W3C Use Cases is as follows:

```
for $s in
    doc("report1.xml")//section[section.title = "Procedure"]
return ($s//incision)[2]/instrument
```

The rewritten query in *Minimal XQuery* syntax is as follows:

```
for $s in select('section',subtreesdfs(document('report1'))) do
where not empty
  (select('procedure',children(
      select('section.title',children($s)))))
return
  select('instrument',children(head(tail(
      select('incision',subtreesdfs($s))))))
```

2. The Result of the SQL Query

The resulting XML forest of the SQL plan generated by the query processor is as follows:

```
<instrument>
  <electrocautery> </electrocautery>
</instrument>
```

5.2.6 Experiment 6: W3C Use Case "SGML" Q3

The example document and queries in this Use Case were originated from those created for a 1992 conference on Standard Generalized Markup Language (`SGML`). The `DTD` and the example document have been transformed from SGML to XML to be used in [3]. We take the query $Q3$ from the Use Case as an input query in the experiment.

1. The Input XQuery Query

For the query Q3 "*Locate all paragraphs in the introduction of a section that is in a chapter that has no introduction.*"from the W3C Use Case "SGML", the XQuery solution from the W3C Use Cases is as follows:

```
<result>
  {
    for $c in doc("sgml.xml")//chapter
    where empty($c/intro)
    return $c/section/intro/para
  }
</result>
```

The rewritten query in *Minimal XQuery* syntax is as follows:

```
<result>
   for $c in select('chapter',subtreesdfs(document('sgml'))) do
   where empty(select('intro',children($c)))
   return select('para',children(select('intro',
          children(select('section',children($c))))))
</result>
```

2. The Result of the SQL Query

The resulting XML forest of the SQL plan generated by the query processor is as follows:

```
<result>
  <para>
    <The Graphic Communications Association ...>
    </The Graphic Communications Association ...>
  </para>
  <para>
    <security>
      <c> </c>
    </security>
    <Exiled members of the former ...>
    </Exiled members of the former ...>
  </para>
</result>
```

Since some of the strings in the resulting XML forest are quite long, we just show the beginning part of the strings and omit the rest.

5.2.7 Experiment 7: XMark Benchmark Query Q8

1. The Input XQuery Query

For the query Q8 "*List the names of persons and the number of items they bought.*" from the XMark Benchmark:

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site
                    /closed_auctions/closed_auction
          where $t/buyer/@person = $p/@id
          return $t
return <item person=$p/name/text()> count ($a) </item>
```

The query is rewritten in *Minimal XQuery* syntax as follows:

```
for $p in select('person',children(select('people',children
          (select('site',document('auction')))))) do
let $a = for $t in select('closed_auction',children(
                    select('closed_auctions',children
                   (select('site',document('auction')))))) do
         where children(select('person',children(select('buyer',
               children($t))))) = children(select('id',children($p)))
         return $t
in <item>
      <person> children(select('name',children($p)))</person> @
      count($a)
   </item>
```

2. The Result of the SQL Query

The source XML data we use in this experiment is generated by the XMark data generator with a scale factor 0. The resulting XML forest of the SQL query for XMark $Q8$ is as follows:

```
<item>
  <person>
    <Jaak Tempesti> </Jaak Tempesti>
    <5> </5>
  </person>
</item>
```

5.2.8 Experiment 8: XMark Benchmark Query Q9

1. The Input XQuery Query

For the query Q9 "*List the names of persons and the number of items they bought in Europe.*" from the XMark Benchmark:

```
for $p in document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site
                    /closed_auctions/closed_auction
          let $n := for $t2 in document("auction.xml")
                              /site/regions/europe/item
                    where  $t/itemref/@item = $t2/@id
                    return $t2
          where $p/@id = $t/buyer/@person
          return <item> $n/name/text() </item>
return <person name=$p/name/text()> $a </person>
```

The rewritten query in *Minimal XQuery* syntax is as follows:

```
for $p in select('person',children(select('people',children
          (select('site',document('auction')))))) do
let $a = for $t in select('closed_auction',children(select('closed_auctions',
                   children(select('site',document('auction')))))) do
         let $n = for $t2 in select('item',children(select('europe',
                             children(select('regions',children(select
                             ('site',document('auction')))))))) do
                  where children(select('item',children(select('itemref',
                        children($t))))) = children(select('id',children($t2)))
                  return $t2
         in where children(select('id',children($p))) =
               children(select('person',children(select('buyer',children($t)))))
            return <item> children(select('name',children($n))) </item>
in <person> <name> children(select('name',children($p))) </name> @ $a </person>
```

2. The Result of the SQL Query

We use the same source XML data from Experiment 7. The resulting XML forest of the SQL query for XMark $Q9$ is as follows:

```
  <person>
    <name> <Jaak Tempesti> </Jaak Tempesti> </name>
    <item> </item>
    <item> </item>
    <item>
      <abhorr execution beckon rue>
```

```
      </abhorr execution beckon rue>
    </item>
    <item> </item>
    <item> </item>
  </person>
```

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we designed and implemented an *XQuery-to-SQL Query Processor* based on the *Dynamic Intervals* approach [9]. We also provided a comprehensive translation for XQuery basic operations and FLWR expressions. The query processor we implemented is able to translate a complex XQuery query, which might include arbitrarily composed and nested basic functions, element constructors, and nested FLWR expressions, into a single SQL query for RDBMS and a physical plan for the *XQuery-enhanced Relational Engine* [15].

The optimization approaches, which include the optimization algorithms to combine SQL expressions into a single SQL SELECT block and the succinct SQL translation templates for XQuery expressions, can effectively reduce the size of the final SQL query and produce preferable SQL plans.

The preferable *merge-join* approach is proposed to efficiently handle value joins in FLWR expressions[1]. By using the *merge-join* approach, the inefficient *nested-loop* evaluation can be avoided. Proofs to prove the correctness of the *merge-join* approach are provided. The proposed *merge-join* patterns are general enough to capture a considerable number of the XQuery fragments that can utilize the preferred *merge-join* approach. We also propose a series of query rewriting rules to capture a larger fraction of the *merge-join* fragments. With the help of the *merge-join* patterns and the rewriting rules, the *XQuery Query Optimizer* is able to

[1]Theoretically, the *merge join* is more efficient than the *nested-loop join*. The efficiency of using the *merge-join* operation to handle the FLWR expressions are proved by the experimental results in [15].

capture a significant number of *merge-join* fragments in the input XQuery queries. As a result, the processor is able to generate predictably efficient relational plans. The experimental results justify the correctness of our work.

6.2 Future Work

We have the following suggestions for the future work:

- A more complex syntax can be designed for *Minimal XQuery* to handle a larger class of XQuery expressions, which include XQuery basic operations like *contain*, *before*, *after*, etc.; complex arithmetic expressions; the *Order By* expression in FLWOR expressions.
- The *merge-join* approach can be further improved by designing additional *merge-join* patterns and rewriting rules to capture the a larger set of *merge-join* fragments.
- The physical plan that our system generates for the *XQuery-enhanced Relational Engine* [15] can be further optimized.
- Currently, we only implemented *head*, *tail* and *last* operations to handle path expressions with predicates. For a path expression like "$\$p/person[3]$", the rewritten XQuery query in *Minimal XQuery* syntax is quite cumbersome. Additional basic operations can be designed to efficiently handle a wider range of path expressions with predicates.

Appendix A

A Comprehensive Translation for XQuery Basic Operations

A.1 SQL Translation Fragments for Basic Operations

There are two SQL fragments that appear in most of the SQL translation templates for the XQuery basic operations. The relational algebra trees that map the two SQL fragments are provided as follows.

A.1.1 Relational Algebra Tree for the First SQL Fragment

The following fragment appears frequently in the SQL templates for XQuery basic operations:

(SELECT $i, s, l - i * w_e$ AS $l, r - i * w_e$ AS r
FROM I, T_e
WHERE $i * w_e \leq l$ AND $r < (i+1) * w_e$) T_{e_i}

The corresponding relational algebra tree is shown as follows:

$$
\begin{array}{l}
T_{e\_i}(T_e) = \\
\quad \texttt{Project}[1,2,5,6](\\
\quad\quad \texttt{AddColumn}[C4 - C1 * w_e](\\
\quad\quad\quad \texttt{AddColumn}[C3 - C1 * w_e](\\
\quad\quad\quad\quad \texttt{Select}[C4 < (C1+1) * w_e](\\
\quad\quad\quad\quad\quad \texttt{Select}[C1 * w_e \leq C3](\\
\quad\quad\quad\quad\quad\quad \texttt{LoopJoin}[\,](I, T_e))))))
\end{array}
$$

A.1.2 Relational Algebra Tree for the Second SQL Fragment

The second SQL fragment that appears in all SQL translation templates for XQuery basic operations except for the *empty constructor* is as follows:

$$
\begin{array}{l}
\texttt{CREATE VIEW } T_{\mathsf{XFn}}(T_{e_1} \ldots T_{e_m}) \texttt{ AS} \\
\quad \texttt{SELECT } s, l + i * w_{\mathsf{XFn}} \texttt{ AS } l, r + i * w_{\mathsf{XFn}} \texttt{ AS } r \\
\quad \texttt{FROM } \quad (\\
\qquad\qquad \vdots \\
\qquad\quad) \;\; Q_{\mathsf{XFn}}
\end{array}
$$

where Q_{XFn} returns tuples (i, s, l, r). Let $T_{Q_{\mathsf{XFn}}}$ (with resulting tuples $(C1, C2, C3, C4)$) be the relational algebra tree for Q_{XFn}. The corresponding relational algebra tree for the above SQL fragment is shown as follows:

$$
\begin{array}{l}
T_{\mathsf{XFn}}(T_{Q_{\mathsf{XFn}}}) = \\
\quad \texttt{Project}[2,5,6](\\
\quad\quad \texttt{AddColumn}[C4 + C1 * w_{\mathsf{XFn}}](\\
\quad\quad\quad \texttt{AddColumn}[C3 + C1 * w_{\mathsf{XFn}}](T_{Q_{\mathsf{XFn}}})))
\end{array}
$$

A.2 Translations for XQuery Basic Operations

A.2.1 The DOCUMENT operator

The expression: $\mathsf{document}('filename')$

The SQL Translation Template

```
CREATE VIEW T_document(T_filename) AS
  SELECT s, l + i * w_document AS l, r + i * w_document AS r
  FROM   I, T_filename
```

The width of the result is $w_{\mathsf{document}} \geq r_{\mathsf{max}} + 1$.

The Relational Algebra Tree

If the name of the document is specified as an XML file with a suffix "*.xml*" in the string $filename$, then the *document* operation has the following relational algebra tree mapping:

$T_{Q_{\mathsf{XFn}}} =$

```
LoopJoin[ ](I,
   XMLFileReader[filename]()
```

Otherwise, it has an alternative mapping:

$T_{Q_{\mathsf{XFn}}} =$

```
LoopJoin[ ](I,
   TextFileReader[filename]()
```

A.2.2 The Empty Constructor

The expression: $[\,]()$

The SQL Translation Template

```
CREATE VIEW T_empty AS
 SELECT 's' AS s, 0 AS l, 1 AS r
 FROM   EMPTY
```

The width of the result is $w_{\mathsf{empty}} = 0$.

The Relational Algebra Tree

$T_{empty} =$

AddColumn[1](
AddColumn[0](
AddColumn['s'](
EMPTY[ ]())))

A.2.3 The Element Constructor

The expression: $\mathsf{xnode}('label', e)$

The SQL Translation Template

```
CREATE VIEW T_xnode('label', T_e) AS
 SELECT s, l + i * w_xnode AS l, r + i * w_xnode AS r
 FROM  (
         ( SELECT i,'label' AS s, 0 AS l, w_e + 1 AS r
           FROM I, UNIT )
         UNION ALL
         ( SELECT i, s, l + 1 AS l, r + 1 AS r
           FROM
             ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
               FROM   I, T_e
               WHERE  i * w_e <= l  AND  r < (i + 1) * w_e  )  T'_i
         )
       )  Q_XFn
```

The width of the result is $w_{\mathsf{XNode}} = w_e + 2$.

The Relational Algebra Tree

Let $T_{e\_i}(T_e)$ be the relational algebra tree for the SQL fragment shown in Section A.1.1. The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$T_{Q_{\mathsf{XFn}}} =$

```
CatUnion[ ](
   AddColumn[w_e + 1](
      AddColumn[0](
         AddColumn['label'](
            LoopJoin[ ](I,
               UNIT[ ]())))))
   Project[1, 2, 5, 6](
      AddColumn[C4 + 1](
         AddColumn[C3 + 1](T_{e_i}(T_e))))
)
```

A.2.4 The Concatenation Constructor

The expression: e_1 @ e_2

The SQL Translation Template

```
CREATE VIEW T_catunion(T_e1, T_e2) AS
  SELECT s, l + i * w_catunion AS l, r + i * w_catunion AS r
  FROM (
       ( SELECT i, s, l − i * w_e1 AS l, r − i * w_e1 AS r
         FROM I, T_e1
         WHERE i * w_e1 ≤ l AND r < (i + 1) * w_e1
       )
       UNION ALL
       ( SELECT i, s, l + w_e1 AS l, r + w_e1 AS r
         FROM
           ( SELECT i, s, l − i * w_e2 ASl, r − i * w_e2 AS r
             FROM I, T_e2
             WHERE i * w_e2 ≤ l AND r < (i + 1) * w_e2
           ) T_e2
      )
  ) Q_XFn
```

The width of the result is $w_@ = w_{e_1} + w_{e_2}$.

The Relational Algebra Tree

Let $T_{e\_i}(T_{e_1})$ and $T_{e\_i}(T_{e_2})$ be the relational algebra trees for the SQL fragment, which is shown in Section A.1.1, with input tables T_{e_1} and T_{e_2} respectively. The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{aligned}
&T_{Q_{\mathsf{XFn}}} = \\
&\quad \texttt{CatUnion}[\,](T_{e\_i}(T_{e_1}), \\
&\qquad \texttt{Project}[1,2,5,6](\\
&\qquad\quad \texttt{AddColumn}[C4 + w_{e_2}](\\
&\qquad\qquad \texttt{AddColumn}[C3 + w_{e_2}](T_{e\_i}(T_{e_2})))) \\
&\quad)
\end{aligned}
$$

A.2.5 The HEAD Operator

The expression: $\mathsf{head}(e)$

The SQL Translation Template

```
CREATE VIEW T_head(T_e) AS
  SELECT s, l + i * w_head AS l, r + i * w_head AS r
  FROM
      ( SELECT u.i AS i, u.s AS s, u.l AS l, u.r AS r
        FROM
           ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
             FROM I, T_e
             WHERE i * w_e ≤ l AND r < (i + 1) * w_e
           ) u,
           ( SELECT i, s, l, r
             FROM
               ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                 FROM I, T_e
                 WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) v
             WHERE NOT EXISTS
               ( SELECT *
                 FROM
                    ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                      FROM I, T_e
                      WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) w
                 WHERE w.l < v.l AND v.i = w.i)
           ) vv
        WHERE vv.l ≤ u.l AND u.r ≤ vv.r AND vv.i = u.i
      ) Q_XFn
```

The resulting width of HEAD operation is $w_{\mathsf{head}} = w_e$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{aligned}
&T_{Q_{\mathsf{XFn}}} = \\
&\quad \texttt{Project}[1,2,3,4](\\
&\qquad \texttt{Select}[C4 \leq C8](\\
&\qquad\quad \texttt{Select}[C7 \leq C3](\\
&\qquad\qquad \texttt{LoopJoin}[1=1](T_{e\_i}(T_e), \\
&\qquad\qquad\quad \texttt{LoopExceptIn}[C1 = C1, C3 > C3](T_{e\_i}(T_e), T_{e\_i}(T_e))))))
\end{aligned}
$$

A.2.6 The LAST Operator

The expression: $\mathsf{last}(e)$

The SQL Translation Template

```
CREATE VIEW T_last(T_e) AS
  SELECT s, l + i * w_last AS l, r + i * w_last AS r
  FROM
    ( SELECT u.i AS i, u.s AS s, u.l AS l, u.r AS r
      FROM
          ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u,
          ( SELECT i, s, l, r
            FROM
                ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                  FROM I, T_e
                  WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) v
            WHERE NOT EXISTS
                ( SELECT *
                  FROM
                      ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                        FROM I, T_e
                        WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) w
                  WHERE v.r < w.r AND v.i = w.i )
          ) vv
      WHERE vv.l ≤ u.l AND u.r ≤ vv.r AND vv.i = u.i
    ) Q_XFn
```

The resulting width of *last* operation is $w_{\mathsf{last}} = w_e$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{array}{l}
T_{Q_{\mathsf{XFn}}} = \\
\quad \texttt{Project}[1,2,3,4](\\
\quad\quad \texttt{Select}[C4 \leq C8](\\
\quad\quad\quad \texttt{Select}[C7 \leq C3](\\
\quad\quad\quad\quad \texttt{LoopJoin}[1=1](T_{e\_i}(T_e), \\
\quad\quad\quad\quad\quad \texttt{LoopExceptIn}[C1 = C1, C4 < C4](T_{e\_i}(T_e), T_{e\_i}(T_e))))))
\end{array}
$$

A.2.7 The TAIL Operator

The expression: $\mathsf{tail}(e)$

The SQL Translation Template

```
CREATE VIEW T_tail(T_e) AS
    SELECT s, l + i * w_tail AS l, r + i * w_tail AS r
    FROM
       ( SELECT u.i AS i, u.s AS s, u.l AS l, u.r AS r
         FROM
           ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
             FROM I, T_e
             WHERE i * w_e ≤ l AND r < (i + 1) * w_e
           ) u,
           ( SELECT i, s, l, r
             FROM
               ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                 FROM I, T_e
                 WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) v
             WHERE NOT EXISTS
               ( SELECT *
                 FROM
                   ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                     FROM I, T_e
                     WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) w
                 WHERE w.l < v.l AND v.i = w.i )
           ) vv
         WHERE vv.r < u.l AND vv.i = u.i
    ) Q_XFn
```

The resulting width of *tail* operation is $w_{\mathsf{tail}} = w_e$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{array}{l}
T_{Q_{\mathsf{XFn}}} = \\
\quad \texttt{Project}[1,2,3,4](\\
\quad\quad \texttt{Select}[C8 < C3](\\
\quad\quad\quad \texttt{LoopJoin}[1=1](T_{e\_i}(T_e), \\
\quad\quad\quad\quad \texttt{LoopExceptIn}[C1 = C1, C3 > C3](T_{e\_i}(T_e), T_{e\_i}(T_e)))))
\end{array}
$$

A.2.8 The ROOTS Operator

The expression: $\mathsf{roots}(e)$

The SQL Translation Template

```
CREATE VIEW T_roots(T_e) AS
  SELECT s, l + i * w_roots AS l, r + i * w_roots AS r
  FROM
      ( SELECT u.i, u.s, u.l, u.r
        FROM
          ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u
        WHERE NOT EXISTS (
          SELECT *
          FROM
              ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                FROM I, Te
                WHERE i * w_e ≤ l AND r < (i + 1) * w_e
              ) v
          WHERE v.l < u.l AND u.r < v.r AND u.i = v.i )
  ) Q_XFn
```

The above SQL translation template can be simplified as follows:

```
CREATE VIEW T_roots(T_e) AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM   T_e u
  WHERE NOT EXISTS (
    SELECT *
    FROM   T_e v
    WHERE  v.l < u.l AND u.r < v.r )
```

The resulting width is set to $w_{\mathsf{roots}} = w_e$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the first SQL template is as follows:

$$T_{Q_{\mathsf{XFn}}} = \texttt{LoopExceptIn}[C1 = C1, C3 > C3, C4 < C4](T_{e\_i}(T_e), T_{e\_i}(T_e))$$

The relational algebra tree for the simplified SQL translation template is shown as follows:

$$T_{\texttt{roots}} = \texttt{LoopExceptIn}[C2 > C2, C3 < C3](T_e, T_e)$$

A.2.9 The REVERSE Operator

The expression: $\mathsf{reverse}(e)$

The SQL Translation Template

Let $Q_{ROOTS(T_e)}$ represents the SQL template for the *roots* operation with input table parameter T_e. The SQL template for the *reverse* operation is as follows:

```
CREATE VIEW T_reverse(T_e) AS
  SELECT s, l + i * w_reverse AS l, r + i * w_reverse AS r
  FROM
    ( SELECT u.i, u.s, u.l - r.l + (w_e - r.r) AS l,
             u.r - r.l + (w_e - r.r) AS r
      FROM
        ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
          FROM I, T_e
          WHERE i * w_e ≤ l AND r < (i + 1) * w_e
        ) u,
        ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
          FROM I, Q_ROOTS(T_e)
          WHERE i * w_e ≤ l AND r < (i + 1) * w_e
        ) r
      WHERE r.l ≤ u.l AND u.r ≤ r.r AND u.i = r.i
    ) Q_XFn
```

The width of the result is $w_{\mathsf{reverse}} = w_e$.

The Relational Algebra Tree

Let *Roots_*T_e be the relational algebra tree for the SQL template for the *roots* operation $\mathsf{roots}(T_e)$. The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{aligned}
&T_{Q_{\mathsf{XFn}}} = \\
&\quad \texttt{Project}[1,2,9,10](\\
&\qquad \texttt{AddColumn}[C4 - C7 + (w_e - C8)](\\
&\qquad\quad \texttt{AddColumn}[C3 - C7 + (w_e - C8)](\\
&\qquad\qquad \texttt{Select}[C4 \leq C8](\\
&\qquad\qquad\quad \texttt{Select}[C7 \leq C3](\\
&\qquad\qquad\qquad \texttt{LoopJoin}[1 = 1](T_{e\_i}(T_e), T_{e\_i}(\mathit{Roots\_}T_e)))))))
\end{aligned}
$$

A.2.10 The DISTINCT Operator

The expression: $\mathsf{distinct}(e)$

The SQL Translation Template

The SQL template for the *distinct* operation is as follows:

```
CREATE VIEW T_distinct(T_e) AS
  SELECT s, l + i * w_distinct AS l, r + i * w_distinct AS r
  FROM
      ( SELECT u.i AS i, u.s AS s, u.l AS l, u.r AS r
        FROM
          ( SELECT  i, s, l − i * w_e AS l, r − i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u,
          ( SELECT i, s, l, r
            FROM
              ( SELECT i, s, l − i * w_e AS l, r − i * w_e AS r
                FROM I, Q_ROOTS(T_e)
                WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) v
            WHERE NOT EXISTS
              ( SELECT *
                FROM
                    ( SELECT i, s, l − i * w_e AS l, r − i * w_e AS r
                      FROM I, Q_ROOTS(T_e)
                      WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) w
                WHERE w.l < v.l AND w.s = v.s AND v.i = w.i )
          ) vv
       WHERE vv.l ≤ u.l AND u.r ≤ vv.r AND vv.i = u.i
      ) Q_XFn
```

The width of the result is $w_{\mathsf{distinct}} = w_e$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{array}{l}
T_{Q_{\mathsf{XFn}}} = \\
\quad \texttt{Project}[1,2,3,4](\\
\quad\quad \texttt{Select}[C4 \leq C8](\\
\quad\quad\quad \texttt{Select}[C7 \leq C3](\\
\quad\quad\quad\quad \texttt{LoopJoin}[1=1](T_e\_i, \\
\quad\quad\quad\quad\quad \texttt{LoopExceptIn}[C1 = C1, C2 = C2, C3 > C3] \\
\quad\quad\quad\quad\quad\quad\quad (T_{e\_i}(Roots\_T_e), T_{e\_i}(Roots\_T_e))))))
\end{array}
$$

A.2.11 The SELECT Operator

The expression: $\mathsf{select}('str', e)$

The SQL Translation Template

The SQL template for the *distinct* operation is as follows:

```
CREATE VIEW T_select('str', T_e) AS
  SELECT s, l + i * w_select AS l, r + i * w_select AS r
  FROM
      ( SELECT u.i AS i, u.s AS s, u.l AS l, u.r AS r
        FROM
          ( SELECT  i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u,
          ( SELECT i, s, l, r
            FROM
              ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                FROM I, Q_ROOTS(T_e)
                WHERE i * w_e ≤ l AND r < (i + 1) * w_e ) v
            WHERE s ='str'
          ) r
        WHERE r.l ≤ u.l AND u.r ≤ r.r AND u.i = r.i
      ) Q_XFn
```

The resulting width is $w_{\mathsf{select}} = w_e$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{array}{l}
T_{Q_{\mathsf{XFn}}} = \\
\quad \texttt{Project}[1,2,3,4](\\
\quad\quad \texttt{Select}[C4 \leq C8](\\
\quad\quad\quad \texttt{Select}[C7 \leq C3](\\
\quad\quad\quad\quad \texttt{LoopJoin}[1=1](T_{e\_i}(T_e), \\
\quad\quad\quad\quad\quad \texttt{Select}[C2 = 'str'](T_{e\_i}(Roots\_T_e))))))
\end{array}
$$

A.2.12 The SUBTREESDFS Operator

The expression: $\mathsf{subtreesdfs}(e)$

The SQL Translation Template

```
CREATE VIEW T_subtreesdfs(T_e) AS
  SELECT s, l + i * w_subtreesdfs AS l, r + i * w_subtreesdfs AS r
  FROM
      ( SELECT u.i AS i, u.s AS s, u.l + v.l * w_e AS l, u.r + v.l * w_e AS r
        FROM
          ( SELECT  i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u,
          ( SELECT  i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) v
        WHERE v.l ≤ u.l AND u.r ≤ v.r AND u.i = v.i
      ) Q_XFn
```

The width of the result is $w_{\mathsf{subtreesdfs}} = w_e^2$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$T_{Q_{\mathsf{XFn}}} =$
$\texttt{Project}[1,2,9,10]($
$\texttt{AddColumn}[C4 + w_e * C7]($
$\texttt{AddColumn}[C3 + w_e * C7]($
$\texttt{Select}[C4 \leq C8]($
$\texttt{Select}[C7 \leq C3]($
$\texttt{LoopJoin}[1 = 1](T_{e\_i}(T_e), T_{e\_i}(T_e)))))))$

A.2.13 The CHILDREN Operator

The expression: $\mathsf{children}(e)$

The SQL Translation Template

```
CREATE VIEW T_children(T_e) AS
  SELECT s, l + i * w_children AS l, r + i * w_children AS r
  FROM
      ( SELECT u.i, u.s, u.l, u.r
        FROM
          ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, T_e
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u
        WHERE EXISTS (
          SELECT *
          FROM
              ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                FROM I, Te
                WHERE i * w_e ≤ l AND r < (i + 1) * w_e
              ) v
          WHERE v.l < u.l AND u.r < v.r AND u.i = v.i )
  ) Q_XFn
```

Similar to the *roots* operation, the above SQL translation can be simplified as follows:

```
CREATE VIEW T_children(T_e) AS
  SELECT u.s AS s, u.l AS l, u.r AS r
  FROM   T_e u
  WHERE EXISTS (
    SELECT *
    FROM   T_e v
    WHERE  v.l < u.l AND u.r < v.r  )
```

The width of the result is $w_{\mathsf{children}} = w_x$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the first SQL template for the *children* operation is as follows:

$$T_{Q_{\mathsf{XFn}}} = \\ \quad \mathtt{LoopIn}[C1 = C1, C3 > C3, C4 < C4](T_e\_i, T_e\_i)$$

The relational algebra tree for the simplified SQL template is shown as follows::

$$T_{\mathtt{children}} = \\ \quad \mathtt{LoopIn}[C2 > C2, C3 < C3](T_e, T_e)$$

A.2.14 The COUNT Operator

The expression: $\mathsf{count}(e)$

The SQL Translation Template

```
CREATE VIEW T_count(T_e) AS
  SELECT s, l + i * w_count AS l, r + i * w_count AS r
  FROM
    ( ( SELECT i, char(COUNT(*)) AS s, 0 AS l, 1 AS r
        FROM
          ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
            FROM I, Q_ROOTS(T_e)
            WHERE i * w_e ≤ l AND r < (i + 1) * w_e
          ) u
        GROUP BY i
      )
      UNION ALL
      ( SELECT i, '0' AS s, 0 AS l, 1 AS r
        FROM
          ( SELECT i
            FROM I i
            WHERE NOT EXISTS (
                SELECT *
                FROM
                  ( SELECT i, s, l - i * w_e AS l, r - i * w_e AS r
                    FROM I, Q_ROOTS(T_e)
                    WHERE i * w_e ≤ l AND r < (i + 1) * w_e
                  ) v
                WHERE i.i = v.i )
          ) i_r
      )
    ) Q_XFn
```

The width of the result is $w_{\mathsf{count}} = 2$.

The Relational Algebra Tree

The relational algebra tree that maps the Q_{XFn} in the SQL template is as follows:

$$
\begin{aligned}
T_{Q_{\mathtt{XFn}}} = \\
&\mathtt{CatUnion}[\,](\\
&\quad \mathtt{CountAggregate}[1](T_{e\_i}(Roots\_T_e))\\
&\quad\quad \mathtt{AddColumn}[1](\\
&\quad\quad\quad \mathtt{AddColumn}[0](\\
&\quad\quad\quad\quad \mathtt{AddColumn}[\prime 0\prime](\\
&\quad\quad\quad\quad\quad \mathtt{LoopExceptIn}[C1 = C1](I, T_{e\_i}(Roots\_T_e)))))\\
)
\end{aligned}
$$

Appendix B

Experiments

B.1 Experiment 9: W3C Use Case "XMP" Q3

1 The Input XQuery Query

For the query Q3 "*For each book in the bibliography, list the title and authors, grouped inside a "result" element.* " from the W3C Use Case "XMP":

```
<results>
{
    for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
    return
        <result>
            { $b/title }
            { $b/author  }
        </result>
}
</results>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<results>
   for $b in select('book',children(select('bib',document('bib')))) do
   <result>
      select('title',children($b)) @
      select('author',children($b))
   </result>
</results>
```

2 The Result of the SQL Query

The following XML forest corresponds to the resulting relational encoding obtained by executing the generated SQL query in DB2:

```
<results>
  <result>
    <title>
      <TCP/IP Illustrated> </TCP/IP Illustrated>
    </title>
    <author>
      <last> <Stevens> </Stevens> </last>
      <first> <W.> </W.> </first>
    </author>
  </result>
  <result>
    <title>
      <Advanced Programming in the Unix environment>
      </Advanced Programming in the Unix environment>
    </title>
    <author>
      <last> <Stevens> </Stevens> </last>
      <first> <W.> </W.> </first>
    </author>
  </result>
  <result>
    <title> <Data on the Web> </Data on the Web> /title>
    <author>
      <last> <Abiteboul> </Abiteboul> </last>
      <first> <Serge> </Serge> </first>
    </author>
    <author>
      <last> <Buneman> </Buneman> </last>
      <first> <Peter> </Peter> </first>
```

```
    </author>
    <author>
      <last> <Suciu> </Suciu> </last>
      <first> <Dan> </Dan> </first>
    </author>
  </result>
  <result>
    <title>
      <The Economics of Technology and Content for Digital TV>
      </The Economics of Technology and Content for Digital TV>
    </title>
  </result>
</results>
```

B.2 Experiment 10: W3C Use Case "XMP" Q5

1 The Input XQuery Query

For the query Q5 "*For each book found at both bstore1.example.com and bstore2.-example.com, list the title of the book and its price from each source.*" from the W3C Use Case "XMP":

```
<books-with-prices>
  {
    for $b in doc("http://bstore1.example.com/bib.xml")//book,
        $a in doc("http://bstore2.example.com/reviews.xml")//entry
    where $b/title = $a/title
    return
        <book-with-prices>
            { $b/title }
            <price-bstore2>{ $a/price/text() }</price-bstore2>
            <price-bstore1>{ $b/price/text() }</price-bstore1>
        </book-with-prices>
  }
</books-with-prices>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<books-with-prices>
  for $b in select('book',subtreesdfs(document('bib'))) do
  for $a in select('entry',subtreesdfs(document('reviews'))) do
  where
     children(select('title',children($b)))
     =
     children(select('title',children($a)))
  return
     <book-with-prices>
        select('title',children($b)) @
        <price-bstore2>
            children(select('price',children($a)))
        </price-bstore2> @
        <price-bstore1>
            children(select('price',children($b)))
        </price-bstore1>
     </book-with-prices>
</books-with-prices>
```

2 The Result of the SQL Query

The following XML forest corresponds to the resulting relational encoding obtained by executing the generated SQL query in DB2:

```
<books-with-prices>
  <book-with-prices>
    <title>
      <TCP/IP Illustrated> </TCP/IP Illustrated>
    </title>
    <price-bstore2> <65.95> </65.95> </price-bstore2>
    <price-bstore1> <65.95> </65.95> </price-bstore1>
  </book-with-prices>
  <book-with-prices>
    <title>
      <Advanced Programming in the Unix environment>
      </Advanced Programming in the Unix environment>
    </title>
      <price-bstore2> <65.95> </65.95> </price-bstore2>
      <price-bstore1> <65.95> </65.95> </price-bstore1>
  </book-with-prices>
```

```
  <book-with-prices>
    <title> <Data on the Web> </Data on the Web> /title>
    <price-bstore2> <34.95> </34.95> </price-bstore2>
    <price-bstore1> <39.95> </39.95> </price-bstore1>
  </book-with-prices>
</books-with-prices>
```

B.3 Experiment 11: W3C Use Case "XMP" Q11

1 The Input XQuery Query

For the query Q11 "*For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the book title and the editor's affiliation.*" from the W3C Use Case "XMP":

```
<bib>
{
  for $b in doc("http://bstore1.example.com/bib.xml")//book[author]
  return
     <book>
        { $b/title }
        { $b/author }
     </book>
}
{
  for $b in doc("http://bstore1.example.com/bib.xml")//book[editor]
  return
     <reference>
        { $b/title }
        { $b/editor/affiliation }
     </reference>
}
</bib>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<bib>
  ( for $b in select('book',subtreesdfs(document('bib'))) do
    where not empty (select('author',children($b)))
    return
      <book>
        select('title',children($b)) @ select('author',children($b))
      </book> )
@
  ( for $b2 in select('book',subtreesdfs(document('bib'))) do
    where not empty (select('editor',children($b2)))
    return
      <reference>
         select('title',children($b2)) @
         select('affiliation',children(select('editor',children($b2))))
      </reference> )
</bib>
```

2 The Result of the SQL Query

```
<bib>
  <book>
    <title>
      <TCP/IP Illustrated> </TCP/IP Illustrated>
    </title>
    <author>
      <last> <Stevens> </Stevens> </last>
      <first> <W.> </W.> </first>
    </author>
  </book>
  <book>
    <title>
      <Advanced Programming in the Unix environment>
      </Advanced Programming in the Unix environment>
    </title>
    <author>
      <last> <Stevens> </Stevens> </last>
      <first> <W.> </W.> </first>
    </author>
  </book>
  <book>
    <title> <Data on the Web> </Data on the Web> /title>
```

```
    <author>
      <last> <Abiteboul> </Abiteboul> </last>
      <first> <Serge> </Serge> </first>
    </author>
    <author>
      <last> <Buneman> </Buneman> </last>
      <first> <Peter> </Peter> </first>
    </author>
    <author>
      <last> <Suciu> </Suciu> </last>
      <first> <Dan> </Dan> </first>
    </author>
  </book>
  <reference>
    <title>
      <The Economics of Technology and Content for Digital TV>
      </The Economics of Technology and Content for Digital TV>
    </title>
    <affiliation> <CITI> </CITI> </affiliation>
  </reference>
</bib>
```

B.4 Experiment 12: W3C Use Case "TREE" Q3

1 The Input XQuery Query

For the query Q3 "*How many sections are in Book1, and how many figures?*" from the W3C Use Case "TREE":

```
<section_count>
  { count(doc("book.xml")//section) }
</section_count>,
<figure_count>
  { count(doc("book.xml")//figure) }
</figure_count>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<section_count>
  count(select('section',subtreesdfs(document('book'))))
</section_count> @
<figure_count>
  count(select('figure',subtreesdfs(document('book'))))
</figure_count>
```

2 The Result of the SQL Query

```
<section_count> <7> </7> </section_count>
<figure_count> <3> </3> </figure_count>
```

B.5 Experiment 13: W3C Use Case "SEQ" Q2

1 The Input XQuery Query

For the query Q2 "*In the Procedure section of Report1, what are the first two Instruments to be used?*" from the W3C Use Case "SEQ":

```
for $s in doc("report1.xml")//
          section[section.title = "Procedure"]
return ($s//instrument)[position()<=2]
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $s in select('section',subtreesdfs(document('report1'))) do
where
  not empty(select('procedure',children(
            select('section.title',children($s)))))
return
  let $i := select('instrument',subtreesdfs($s))
  in head($i) @ head(tail($i))
```

2 The Result of the SQL Query

```
<instrument>
  <using electrocautery.> </using electrocautery.>
</instrument>
<instrument>
  <electrocautery> </electrocautery>
</instrument>
```

B.6 Experiment 15: W3C Use Case "SGML" Q1

1 The Input XQuery Query

For the query Q1 "*Locate all paragraphs in the report (all "para" elements occurring anywhere within the "report" element).*" from the W3C Use Case "SGML":

```
<result>
  {
    doc("sgml.xml")//report//para
  }
</result>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<result>
    select('para',subtreesdfs(select('report',
    subtreesdfs(document('sgml')))))
</result>
```

2 The Result of the SQL Query

```
<result>
  <para>
    <With the ever-changing ...> </With the ever-changing ...>
  </para>
  <para>
    <As part of the ...> </As part of the ...>
  </para>
  <para>
```

```
  <While SGML is a ...> </While SGML is a ...>
  <emph> <markup> </markup> </emph>
  <in computer-generated ...> </in computer-generated ...>
</para>
<para>
  <Markup is everything ...> </Markup is everything ...>
  <emph> <marking> </marking> </emph>
  <up of typewritten ...> </up of typewritten ...>
  <emph>
    <procedural markup> </procedural markup>
  </emph>
  <.> </.>
</para>
<para>
  <Most electronic ...> </Most electronic ...>
</para>
<para>
  <Generic markup (also ...> </Generic markup (also ...>
  <emph> <purpose> </purpose> </emph>
  <of the text in ...> </of the text in ...>
</para>
<para>
  <Industries involved ...> </Industries involved ...>
</para>
<para>
  <SGML defines a ...> </SGML defines a ...>
</para>
<para>
  <SGML can describe ...> </SGML can describe ...>
</para>
<para>
  <You can break a ...> </You can break a ...>
</para>
<para>
  <At the heart of ...> </At the heart of ...>
</para>
<para>
  <A database schema ...> </A database schema ...>
  <emph> <rules> </rules> </emph>
  <to help ensure ...> </to help ensure ...>
</para>
<para>
  <Content is the ...> </Content is the ...>
  <emph> <tagging> </tagging> </emph>
  <. Tagging must conform ...> </. Tagging must conform ...>
  <xref> <xrefid> <top4> </top4> </xrefid> </xref>
  <).> </).>
```

```
  </para>
  <para>
    <SGML does not...> </SGML does not...>
  </para>
  <para>
    <The Graphic Communications ...>
    </The Graphic Communications ...>
  </para>
  <para>
    <security> <c> </c> </security>
    <Exiled members of ...> </Exiled members of ...>
  </para>
</result>
```

B.7 Experiment 16: W3C Use Case "SGML" Q2

1 The Input XQuery Query

For the query Q2 "*Locate all paragraph elements in an introduction (all "para" elements directly contained within an "intro" element).*" from the W3C Use Case "SGML":

```
<result>
  {
    doc("sgml.xml")//intro/para
  }
</result>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<result>
    select('para',children(select('intro',
           subtreesdfs(document('sgml')))))
</result>
```

2 The Result of the SQL Query

```
<result>
  <para>
    <With the ever-changing ...> </With the ever-changing ...>
  </para>
  <para>
    <As part of the ...> </As part of the ...>
  </para>
  <para>
    <While SGML is ...> </While SGML is ...>
    <emph> <markup> </markup> </emph>
    <in computer-generated ...> </in computer-generated ...>
  </para>
  <para>
    <Markup is everything ...> </Markup is everything ...>
    <emph> <marking> </marking> </emph>
    <up of typewritten ...> </up of typewritten ...>
    <emph>
      <procedural markup> </procedural markup>
    </emph>
    <.> </.>
  </para>
  <para>
    <SGML defines a ...> </SGML defines a ...>
  </para>
  <para>
    <SGML can describe ...> </SGML can describe ...>
  </para>
  <para>
    <You can break a ...> </You can break a ...>
  </para>
  <para>
    <The Graphic Communications ...> </The Graphic Communications ...>
  </para>
  <para>
    <security> <c> </c> </security>
    <Exiled members of ...> </Exiled members of ...>
  </para>
</result>
```

B.8 Experiment 17: W3C Use Case "SGML" Q5

1 The Input XQuery Query

For the query Q5 "*Locate all classified paragraphs (all "para" elements whose "security" attribute has the value "c").*" from the W3C Use Case "SGML":

```
<result>
  {
    doc("sgml.xml")//para[@security = "c"]
  }
</result>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<result>
    for $c in select('para',subtreesdfs(document('sgml'))) do
    where
      not empty(select('c',children(select('security',children($c)))))
    return $c
</result>
```

2 The Result of the SQL Query

```
<result>
  <para>
    <security> <c> </c> </security>
    <Exiled members of the former ...>
    </Exiled members of the former ...>
  </para>
</result>
```

B.9 Experiment 18: W3C Use Case "SGML" Q6

1 The Input XQuery Query

For the query Q6 "*List the short titles of all sections (the values of the "shorttitle" attributes of all "section" elements, expressing each short title as the value of a new element.)*" from the W3C Use Case "SGML":

```
<result>
  {
    for $s in doc("sgml.xml")//section/@shorttitle
    return <stitle>{ $s }</stitle>
  }
</result>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<result>
   for $s in children(select('shorttitle',children(
             select('section',subtreesdfs(document('sgml'))))))) do
   <stitle> $s </stitle>
</result>
```

2 The Result of the SQL Query

```
<result>
  <stitle>
    <What is markup?> </What is markup?>
  </stitle>
  <stitle>
    <What is SGML?> </What is SGML?>
  </stitle>
  <stitle>
    <How does SGML work?> </How does SGML work?>
  </stitle>
</result>
```

B.10 Experiment 19: W3C Use Case "SGML" Q9

1 The Input XQuery Query

For the query Q9 "*Locate all the topics referenced by a cross-reference anywhere in the report (all the "topic" elements whose "topicid" attribute value is the same as an "xrefid" attribute value of any "xref" element).*" from the W3C Use Case "SGML":

```
<result>
  {
    for $id in doc("sgml.xml")//xref/@xrefid
    return doc("sgml.xml")//topic[@topicid = $id]
  }
</result>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
<result>
    for $id in children(select('xrefid',children(select
               ('xref',subtreesdfs(document('sgml')))))) do
    for $t in select('topic',subtreesdfs(document('sgml'))) do
    where children(select('topicid',children($t))) = $id
    return $t
</result>
```

2 The Result of the SQL Query

```
<result>
  <topic>
    <topicid> <top4> </top4> </topicid>
    <title> <Structure> </Structure> </title>
    <para>
      <At the heart of ...> </At the heart of ...>
    </para>
    <para>
      <A database schema ...> </A database schema ...>
      <emph>
        <rules>
          <to help ensure ...> </to help ensure ...>
        </reles>
      </emph>
    </para>
  </topic>
</result>
```

B.11 Experiment 20: XMark Benchmark Query Q1

1 The Input XQuery Query

For the query Q1 "*Return the name of the person with ID 'person0' registered in North America.*" from the XMark Benchmark:

```
for    $b in document("auction.xml")/site/people/person[@id="person0"]
return $b/name/text()
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $b in select('person',children(select('people',
    children(select('site',document('auction')))))) do
where children(select('id',children($b))) = <person0>[]()</person0>
return children(select('name',children($b)))
```

2 The Result of the SQL Query

The source XML data used in this experiment is generated by the XMark data generator with a scale factor 0.

```
<Jaak Tempesti> </Jaak Tempesti>
```

B.12 Experiment 21: XMark Benchmark Query Q2

1 The Input XQuery Query

For the query Q2 "*Return the initial increases of all open auctions.*" from the XMark Benchmark:

```
for $b in document("auction.xml")/site/open_auctions/open_auction
return <increase> $b/bidder[1]/increase/text() </increase>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $b in select('open_auction',children(select('open_auctions',
          children(select('site',document('auction')))))) do
<increase>
   children(select('increase',children(
            head(select('bidder',children($b))))))
</increase>
```

2 The Result of the SQL Query

The source XML data used in this experiment is generated by the XMark data generator with a scale factor 0.

```
<increase> <55.50> </55.50> </increase>
```

B.13 Experiment 22: XMark Benchmark Query Q6

1 The Input XQuery Query

For the query Q6 "*How many items are listed on all continents?*" from the XMark Benchmark:

```
for    $b in document("auction.xml")/site/regions
return count ($b//item)
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $b in select('regions',children(select('site',
          document('auction')))) do
count(select('item',subtreesdfs($b)))
```

2 The Result of the SQL Query

The source XML data used in this experiment is generated by the XMark data generator with a scale factor 0.

```
  <6> </6>
```

B.14 Experiment 23: XMark Benchmark Query Q13

1 The Input XQuery Query

For the query Q13 "*List the names of items registered in Australia along with their descriptions.*" from the XMark Benchmark:

```
for $i in document("auction.xml")/site/regions/australia/item
return <item name=$i/name/text()> $i/description </item>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $i in select('item',children(select('australia',
          children(select('regions',children
          (select('site',document('auction')))))))) do
<item>
  <name> children(select('name',children($i))) </name> @
  select('description',children($i))
</item>
```

2 The Result of the SQL Query

The source XML data used in this experiment is generated by the XMark data generator with a scale factor 0.

```
<item>
  <name>
    <scarce brook> </scarce brook>
  </name>
  <description>
    <parlist>
      <listitem>
        <text>
          <senses concave ...> </senses concave ...>
          <keyword>
            <preparation rejoice> </preparation rejoice>
          </keyword>
```

```
    </text>
  </listitem>
  <listitem>
    <text>
      <swear canker ...> </swear canker ...>
      <emph>
        <untruths misgives ...> </untruths misgives ...>
      </emph>
      <error discontent ...> </error discontent ...>
      <keyword>
        <season presently victory women beating>
        </season presently victory women beating>
      </keyword>
      <deprive almost ...> </deprive almost ...>
    </text>
  </listitem>
  <listitem>
    <text>
      <spotted attend ...> </spotted attend ...>
      <bold>
        <naturally sanctuary...> </naturally sanctuary ...>
      </bold>
      <service cricket ...> </service cricket ...>
    </text>
  </listitem>
  <listitem>
    <parlist>
      <listitem>
        <text>
          <maintained peril ...> </maintained peril ...>
        </text>
      </listitem>
      <listitem>
        <text>
          <bold>
            <friar prophetess> </friar prophetess>
          </bold>
          <spirits delays ...> </spirits delays ...>
        </text>
      </listitem>
      <listitem>
        <text>
          <piece hours cruelly april league winged>
          </piece hours cruelly april league winged>
          <keyword>
            <tract element ...> </tract element ...>
          </keyword>
```

```
            <words blessing ...> </words blessing ...>
          </text>
        </listitem>
      </parlist>
    </listitem>
    <listitem>
      <parlist>
        <listitem>
          <text>
            <sent fled bids ...> </sent fled bids ...>
            <emph>
              <preventions spurr ...> </preventions spurr ...>
            </emph>
            <valorous line ...> </valorous line ...>
            <bold> <sold> </sold> </bold>
            <marriage sampson ...> </marriage sampson ...>
            <emph>
              <cars livery stand> </cars livery stand>
            </emph>
            <denay> </denay>
            <keyword>
              <cimber paper admittance tread character>
              </cimber paper admittance tread character>
            </keyword>
            <battlements seen ...> </battlements seen ...>
          </text>
        </listitem>
        <listitem>
          <text>
            <traduc barks ...> </traduc barks ...>
            <keyword>
              <transformed nourish breeds north>
              </transformed nourish breeds north>
            </keyword>
          </text>
        </listitem>
      </parlist>
    </listitem>
  </parlist>
 </description>
</item>
```

B.15 Experiment 24: XMark Benchmark Query Q15

1 The Input XQuery Query

For the query Q15 "*Print the keywords in emphasis in annotations of closed auctions.*" from the XMark Benchmark:

```
for $a in document("auction.xml")/site/closed_auctions/closed_auction
          /annotation/description/parlist/listitem/parlist/listitem
          /text/emph/keyword/text()
return <text> $a </text>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $a in children(select('keyword',children(select('emph',children
          (select('text',children(select('listitem',children
          (select('parlist',children(select('listitem',children
          (select('parlist',children(select('description',children
          (select('annotation',children(select('closed_auction',children
          (select('closed_auctions',children(select('site',
           document('auction'))))))))))))))))))))))))) do
<text>$a</text>
```

2 The Result of the SQL Query

The source XML data "*auction.xml*" used in this experiment is generated by the XMark data generator with a scale factor 0.002 (207KB).

```
<text>
  <wax> </wax>
</text>
<text>
  <pursuivant sparrow hamlet>
  </pursuivant sparrow hamlet>
</text>
```

B.16 Experiment 25: XMark Benchmark Query Q16

1 The Input XQuery Query

For the query Q16 "*Return the IDs of those auctions that have one or more keywords in emphasis.*" from the XMark Benchmark:

```
for $a in document("auction.xml")/site/closed_auctions
          /closed_auction
where not empty ($a/annotation/description/parlist
                 /listitem/parlist/listitem/text
                 /emph/keyword/text())
return <person id=$a/seller/@person />
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $a in select('closed_auction',children(select('closed_auctions',
          children(select('site',document('auction')))))) do
where not empty (children(select('keyword',children(select('emph',
      children(select('text',children(select('listitem',children
      (select('parlist',children(select('listitem',children
      (select('parlist',children(select('description',children
      (select('annotation',children($a)))))))))))))))))))
return
  <person>
     <id>
       children(select('person',children(select('seller',children($a)))))
     </id>
  </person>
```

2 The Result of the SQL Query

The source XML data "*auction.xml*" used in this experiment is generated by the XMark data generator with a scale factor 0.002 (207KB).

```
<person>
  <id>
    <person21> </person21>
  </id>
</person>
<person>
  <id>
    <person19> </person19>
  </id>
</person>
```

B.17 Experiment 26: XMark Benchmark Query Q17

1 The Input XQuery Query

For the query Q17 "*Which persons don't have a homepage?*" from the XMark Benchmark:

```
for    $p in document("auction.xml")/site/people/person
where  empty($p/homepage/text())
return <person name=$p/name/text()/>
```

The query is rewritten in the syntax of *Minimal XQuery* as follows:

```
for $p in select('person',children(select('people', children
          (select('site',document('auction')))))) do
where empty(children(select('homepage',children($p))))
return
   <person> <name>
          children(select('name',children($p)))
   </name> </person>
```

2 The Result of the SQL Query

The source XML data "*auction.xml*" used in this experiment is generated by the XMark data generator with a scale factor 0.001 (114KB).

```
<person>
  <name> <Huei Demke> </Huei Demke> </name>
</person>
<person>
  <name> <Kawon Unni> </Kawon Unni> </name>
</person>
<person>
  <name> <Ewing Andrade> </Ewing Andrade> </name>
</person>
<person>
  <name> <Bassem Manderick> </Bassem Manderick> </name>
</person>
<person>
  <name> <Masanao Marsiglia> </Masanao Marsiglia> </name>
</person>
<person>
  <name> <Saul Schaap> </Saul Schaap> </name>
</person>
<person>
  <name> <Martti Halgason> </Martti Halgason> </name>
</person>
<person>
  <name> <Laurian Grass> </Laurian Grass> </name>
</person>
<person>
  <name> <Shooichi Oerlemans> </Shooichi Oerlemans> </name>
</person>
<person>
  <name> <Uzi Atrawala> </Uzi Atrawala> </name>
</person>
<person>
  <name> <Aloys Singleton> </Aloys Singleton> </name>
</person>
<person>
  <name> <Nestoras Gausemeier> </Nestoras Gausemeier> </name>
</person>
<person>
  <name> <Yechezkel Calmet> </Yechezkel Calmet> </name>
</person>
<person>
  <name> <Slavian Usery> </Slavian Usery> </name>
</person>
<person>
  <name> <Shaoyun Morreau> </Shaoyun Morreau> </name>
</person>
```

Bibliography

[1] XMark – An XML Benchmark Project. Available from http://www.xml-benchmark.org.

[2] Cup user's manual, 1999. Available from http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[3] XML Query Use Cases. Technical report, W3C, 2003.

[4] XQuery 1.0 An XML Query Language. Technical report, W3C, 2003.

[5] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1999.

[6] E. Berk. JLex: A lexical analyzer generator for Java(TM), 2000. Available from http://www.cs.princeton.edu/ appel/modern/java/JLex/.

[7] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost- based approach to XML storage. *Proc. 18th Intl. Conf. on Data Eng.*, pages 64–75, 2002.

[8] Y. Chen, S. B. Davidson, and Y. Zheng. Constraint preserving XML Storage in Relations. *Proc. WebDB Workshop*, 2002.

[9] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. *Proc. SIGMOD Conference*, pages 623–634, 2003.

[10] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. *Proc. SIGMOD Conference*, pages 431–442, 1999.

[11] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. *Proc. VLDB Conference*, pages 201–212, 2003.

[12] M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middle-ware Queries. *Proc. SIGMOD Conference*, pages 103–114, 2002.

[13] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22:27–34, 1999.

[14] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Heterogenous Data Sources. In *Proc. 27th VLDB Conference*, pages 241–250, 2001.

[15] L. Nie. Efficient XQuery Processing using B+ Tree Indices. Master's thesis, University of Waterloo, 2004.

[16] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. *Proc. 27th VLDB Conference*, 2001.

[17] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, S. D. Viglas, J. Naughton, and I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *ACM SIGMOD Record*, 30(3), 2001.

[18] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. 25th VLDB Conference*, pages 302–314, 1999.

[19] I. Tatarinov, S. Viglas, E. J. S. Kevin S. Beyer, Jayavel Shanmugasundaram, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. SIGMOD Conference*, pages 204–215, 2002.

[20] D. Toman and G. E. Weddell. Querying XML: On the Utility of Interval Encoding. Technical report, University of Waterloo, 2002.

[21] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1:110–141, 2001.

[22] X. Zhang, B. Pielech, and E. Rundesnteiner. Honey, I Shrunk the XQuery! – An XML Algebra Optimization Approach. *Proc. 4th WIDM Workshop*, pages 15–22, 2002.