EFFICIENT AND PARALLEL EVALUATION OF XQUERY

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Xiaogang Li, B.S.

* * * * *

The Ohio State University

2006

Dissertation Committee:
Dr. Gagan Agrawal, Adviser
Dr. P. Sadayappan
Dr. Atanas Rountev

Approved by
Adviser
Graduate Program in
Computer Science and
Engineering
© Copyright by
Xiaogang Li
2006
ABSTRACT

With the increased popularity of XML, query and processing of XML data has become a very important topic. Most recent work in this area has been in the context of XQuery, which is the XML query language developed by the World Wide Web Consortium (W3C). This dissertation presents our approach for efficient compilation of XQuery queries to facilitate the development of data intensive applications. As XML and XQuery are being used for larger datasets, parallel execution and stream processing are two solutions to reduce storage and/or the execution time. Accordingly, our efforts are focused on optimization of XQuery and generating efficient code in a cluster and streaming environment. Particularly, the issues that we investigate include: 1) Efficient optimization of XQuery by designing new analysis and transformation techniques, as well as integrating existing compiler optimization and query optimization techniques; 2) Designing new techniques toward efficient parallelization of XQuery; 3) Providing high-level abstraction of a dataset to an application developer through XML Schemas and 4) Code generation of XQuery towards the desired targets, such as clusters and streaming environment.

In the area of high-level optimizations, we have developed a new set of optimization and transformation algorithms for XQuery, which are based on a new internal representation that is referred to as Generalized Nested Loop (GNL). These optimization techniques include aggregation rewrite, loop fusion, loop interchange, and aggregation remapping. Since XQuery is a very powerful and complex functional language, to enable the above...
optimization techniques, we have developed new algorithms to handle arbitrary recursive function and type systems in XQuery. As XML and XQuery are being used for larger datasets, parallelizing XQuery execution can enable faster response. In the area of parallelization, GNL offers a convenient basis for parallelization of XQuery. We present techniques for enumerating parallelization strategies, cost-models for choosing the optimal one, and an algorithm for parallel code generation toward a middleware called ADR. Furthermore, we investigated techniques to parallelize XQuery for native XML datasets on clusters.

To further simplify application development over scientific datasets, we provide a solution by using XML Schemas as a high-level abstraction of a dataset to an application developer. A corresponding low-level Schema describes the actual layout of data and is used by the compiler for code generation. A systematic way for translating the high-level code to a low-level code that achieves high locality and efficient execution is also provided.

For stream processing of XQuery, we have designed the concept of Data Flow Graph and applied a series high-level transformation based on it. The goal of these transformation techniques is to enable a single-pass evaluation strategy for the original query. Based on a SAX parsing engine, we have proposed a new technique to generate efficient code to minimize memory usage.

We have implemented and evaluated the above techniques. Results from several XMark queries and scientific data processing queries show large improvements from new optimizations and good speedups.
This is dedicated to my parents
ACKNOWLEDGMENTS

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

My foremost thank goes to Dr. Gagan Agrawal, my thesis adviser. He has offered invaluable suggestions and advice to shape my research skills, which will help me greatly in the rest of my career. His understanding, encouraging and personal guidance have provided a good basis for this thesis.

Secondly, I owe my most sincere gratitude to Professor Sadayappan, who gave me the opportunity to work on the PARAM project and helped me in various aspects of my graduate study. I also thank Professor Rountev for spending time to serve my committee, and providing valuable feedback to help me improve the dissertation in many ways.

Also, I want to express my thankfulness to Ruoming Jin and Renato Ferreira, their support on ADR and Virtual Microscope helped me greatly on the initial stage of my Ph.D study. I would also like to thank my colleagues at Ohio State University for their help and supports. They are Wei Du, Liang Chen, Xuan Zhang, Leo Glimcher and Li Weng.

Finally and most importantly, I thank my parents in China for their love and many years of support along the path of my academic pursuits. Last but not least, I would like to express my deepest gratitude to my wife, Xiaoyang, for her continuous love, support and encouragement.
VITA

May 12, 1974 .......................... Born - Beijing, China

1997 ................................. B.S. Computer Science & Technology
Beijing University, China

2001-present ........................ Graduate Research Associate,
The Ohio State University.

PUBLICATIONS

Research Publications

1. Efficient Evaluation of XQuery over Streaming Data, Xiaogang Li and Gagan Agrawal. in proceedings of the 31th International Conference on Very Large Data Bases (VLDB) 2005.


5. Using XQuery for Flat-File Based Scientific Datasets, Xiaogang Li and Gagan Agrawal. in proceedings of the 9th International Workshop on Data Base Programming Languages (DBPL), Potsdam, Germany, September 2003.


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in:

- High Performance Computing  Prof. Gagan Agrawal
- Software Engineering  Prof. Atanas Rountev
- Computational Theory  Prof. Gagan Agrawal
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
</tbody>
</table>

## Chapters:

1. Introduction                                     | 1    |
   - 1.1 Research Issues and Related Work            | 2    |
   - 1.2 Summary of Contributions                    | 7    |
   - 1.3 Dissertation Outline                       | 9    |

2. Background                                       | 10   |
   - 2.1 Background: XML, XML Schemas, and XQuery   | 10   |
     - 2.1.1 XML and XML Schemas                    | 10   |
     - 2.1.2 XML Query Language: XQuery             | 12   |
   - 2.2 Data-Intensive Reductions and XQuery      | 16   |
     - 2.2.1 Satellite Data Processing              | 16   |
     - 2.2.2 Multi-Grid Virtual Microscope          | 18   |

viii
5.5 Conclusions ................................................................. 86

6. XQuery Compilation in a Streaming Environment ................. 87
   6.1 Introduction ............................................................... 87
       6.1.1 Overview of Ongoing Efforts ................................. 88
   6.2 Preliminaries ............................................................. 90
       6.2.1 Evaluation Model ............................................. 90
       6.2.2 Problem Overview ........................................... 92
   6.3 High-level Analysis ................................................... 94
       6.3.1 Overview ....................................................... 94
       6.3.2 Stream Data Flow Graph .................................. 96
       6.3.3 High-level Transformations ................................ 98
       6.3.4 Single Pass Analysis ....................................... 103
   6.4 Low-level Analysis and Code Generation .......................... 108
       6.4.1 GNL Representation ....................................... 108
       6.4.2 Code Generation ............................................ 111
   6.5 Experimental Results ............................................... 116
   6.6 Summary ............................................................... 119

7. Parallelization of XQuery over Native XML Datasets .............. 121
   7.1 Introduction ............................................................. 121
   7.2 System Overview and Data Distribution ............................ 122
   7.3 Parallelization of XQuery .......................................... 123
       7.3.1 GNL Formation and Aggregation Rewrite .................. 124
       7.3.2 Parallel Code Generation ................................... 125
   7.4 Experimental Results ............................................... 128

8. Conclusions ............................................................... 133
   8.1 Contributions ........................................................... 134
   8.2 Future Work ............................................................ 135

Bibliography ............................................................... 137
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>84</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>XML and XML Schema</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>An Example Illustrating XQuery’s FLWR Expressions</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>An Example Using XQuery’s Unordered Expression</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Satellite Data Processing Expressed in XQuery</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Multigrid Virtual Microscope Using XQuery</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Example Showing Path Selection for GNL</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of PDG</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithm to Generate Path Dependence Graph from XQuery</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of Generated GNL</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Examples of GNLs after Optimization</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Algorithm to Perform Code Motion of Aggregation Functions</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Example of Greedy Loop Fusion</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Algorithm for Conservative Loop Fusion</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Algorithm for Greedy Loop Fusion</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Example of GNL Interchange</td>
<td>37</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.6</td>
<td>Multigrid Virtual Microscope Using Low Level XQuery</td>
<td>75</td>
</tr>
<tr>
<td>5.7</td>
<td>Relational Algebra Based Approach for High-level to Low-level Transfor-</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>mation</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Data-Centric Transformations on Virtual Microscope Code</td>
<td>82</td>
</tr>
<tr>
<td>5.9</td>
<td>Parallel Performance of satellite</td>
<td>83</td>
</tr>
<tr>
<td>5.10</td>
<td>Parallel Performance for mg-vscope</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview of the Framework</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of Stream Data Flow Graph</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Example of Horizontal Fusion</td>
<td>99</td>
</tr>
<tr>
<td>6.4</td>
<td>Horizontal and Vertical Fusion for Query 1</td>
<td>101</td>
</tr>
<tr>
<td>6.5</td>
<td>Example of Vertical Fusion (Query 3)</td>
<td>103</td>
</tr>
<tr>
<td>6.6</td>
<td>Algorithm for Vertical Fusion</td>
<td>104</td>
</tr>
<tr>
<td>6.7</td>
<td>Stream Data Flow Graphs that Require Multiple Traversals</td>
<td>105</td>
</tr>
<tr>
<td>6.8</td>
<td>Example of GNLs</td>
<td>110</td>
</tr>
<tr>
<td>6.9</td>
<td>Evaluation Code for Query 3</td>
<td>113</td>
</tr>
<tr>
<td>6.10</td>
<td>Evaluation Code for Query 4</td>
<td>114</td>
</tr>
<tr>
<td>6.11</td>
<td>Experiments Results for XMark Queries and Real Streaming Applications</td>
<td>117</td>
</tr>
<tr>
<td>7.1</td>
<td>GNLs and Optimized Code for Satellite Application</td>
<td>126</td>
</tr>
<tr>
<td>7.2</td>
<td>Algorithm for Synthesizing Global Reduction Function from GNL</td>
<td>129</td>
</tr>
<tr>
<td>7.3</td>
<td>Generated Code for Global Reduction</td>
<td>130</td>
</tr>
<tr>
<td>7.4</td>
<td>Parallel Performance: satellite</td>
<td>130</td>
</tr>
</tbody>
</table>
7.5 Parallel Performance: vmscope ................................. 131
7.6 Parallel Performance: XMARK Q20 ............................. 132
7.7 Parallel Performance: XMARK Q5 ............................. 132
CHAPTER 1

INTRODUCTION

Processing and analyzing large volumes of data is playing an increasingly important role in many domains of scientific research. Large datasets are being created by scientific simulations, or arise from digitization of images and/or from data collected by sensors and other instruments [34, 2, 17]. A variety of analysis can be performed on such datasets to better understand scientific processes.

Development of applications that process large scientific datasets is often complicated by complex and specialized data storage formats. When the datasets are disk-residents, understanding the layout and maintaining high locality in accessing them is crucial for obtaining a reasonable performance. Because of the size of the datasets involved, it is also desirable to use parallel platforms for such applications. While parallel platforms such as cluster of workstations with disk farms are well suited for data intensive applications, their use increases the complexity of programming significantly.

Recently, there has been a lot of interest in XML, XQuery and other related technologies developed by the W3C consortium [15]. XML is a flexible exchange format that can represent many classes of data, including structured documents, heterogeneous and semi-structured records, data from scientific experiments and simulations, and digitized images. Besides being a popular research topic in the database community [79], XML has rapidly
gained acceptance in industry and in a variety of scientific and engineering areas. It is very likely that more and more datasets in these areas will be supported with an XML interface for data exchanging.

XQuery [13] is a language currently being developed by the World Wide Web Consortium (W3C). It is designed to be a language in which queries are concise and easily understood, and to be flexible enough to query a broad spectrum of information sources, including both databases and documents.

The focus of this dissertation is to provide extensive compilation support for efficient execution of XQuery programs in sequential and parallel platforms. By designing new techniques and modifying existing techniques from both compiler and database communities, we show that advanced program analysis as well as loop transformation and parallelization techniques can allow efficient execution of XQuery. Although techniques we present in this dissertation were originally motivated by scientific data processing applications, most of them are also helpful in optimization and parallelization of general queries originate from database users. In the following sections, we first discuss the challenges of query processing of XQuery and some related work; then, we present an overview of our research work and list our contributions.

1.1 Research Issues and Related Work

Query Processing and Optimizations: As a flexible declarative language integrating XPath and FLWR expressions, XQuery presents challenges for efficient optimization and processing. XQuery is a strongly typed functional language that supports many features, such as arbitrary recursive functions and flexible type casting. Appropriate techniques are
required to analyze and, in some cases, rewrite them in order to support full features of XQuery. Here both validity and efficiency need to be considered.

Much of the earlier research on XQuery focuses on optimization of XPath expressions [25, 62, 54]. Some of these efforts investigate XPath containment and equivalence, while others provide query rewriting and minimization techniques based on information from DTD or XML Schema. However, we are not aware of any work that integrates the above optimization techniques with the rest of XQuery.

One natural approach for optimizing XQuery is based on relational algebra or tree algebra. Relational approaches include [24, 77, 75, 31, 59] and native approaches based on tree algebra include [19, 45, 46] and [39, 82]. The advantage of using a high-level algebra is that traditional database query optimizations such as reordering of joins and group-bys can be easily performed. However, developing an algebra to fully exploit the power of XQuery is difficult. As an example, recursive functions allowed as part of XQuery can be very hard to model through such an algebra, and we are not aware of any existing effort which is able to do this. Most of the above approaches support only a subset of XQuery.

Galax [33] provides another approach for implementing XQuery based on formal semantics [20], in which XQuery and XPath expressions are first mapped to a core grammar by normalization. A path navigation expression is normalized to an iterative construct using a series of for expressions. The predicates of XPath are translated to conditional expressions. The core grammar approach can support most of the advanced features of XQuery. However, the information on path equivalence and dependence is hard to gather after such normalization. Moreover, opportunities for traditional database optimizations, such as join reordering and unnesting, are hard to recognize after normalization.
Efficient Parallelization of XQuery: Because of the size of the datasets involved in data intensive applications, it is also desirable to use parallel platforms for such applications. Loop transformation and parallelization techniques have been studied extensively in the scientific compilation community [81, 51]. Techniques for automatic data distribution and communication minimization have been studied in [6]. A two step algorithm has been proposed in [22] to investigate space and time trade-off on parallelization and optimization of scientific applications. However, these techniques can not be directly applied to the parallelization of XQuery because of the different language and application contexts.

There is a large body of work on parallelization of queries for relational databases [26, 67, 74]. Parallelization of aggregation queries in SQL-like languages have been investigated in [76]. Various query plans have been studied to process queries in a shared-nothing environment in [74]. Chen et al. present an approach to handle scheduling of a series of joins as well as processor allocation for parallel query execution [18]. Overall, most of these techniques investigated the inter-operator parallelism, where an operator tree is pipelined among different nodes. We are not aware of any existing work on parallelizing of XQuery to exploit intro-operator parallelism. The research issues involved include the selection of an optimal parallelization scheme in the query, efficient parallelization of sequences of joins and parallel code generation for a functional language.

Supporting High-level Abstractions through XML technologies: Development of applications that process large scientific datasets is often complicated by complex and specialized data storage formats. High-level abstractions can be used here to hide low-level details and enable rapid development of such applications.

XML, designed as a flexible data exchanging format, may be used as an abstract interface for storing, retrieving, and processing scientific datasets with specialized formats.
Limited work on supporting high-level abstraction has been done in [11, 61, 69], in which sparse computations can be automatically synthesized from dense codes. Compiler optimizations for out-of-core data-structures have been considered by several projects [35, 36, 47, 63]. However, none of the above work considers using XML as a high-level abstraction, although it is well defined and widely accepted as a language for describing formats.

NetCDF [71] and HDF [42] are among the frequently used data formats for scientific data. While such formats also enable using XML for exchanging of scientific data, they do not hide the complexities of data layout from the users.
Efficient processing XQuery over XML streams: There are limited research efforts on query evaluation on XML streams, and most of them handle only XPath fragments [68, 66, 27]. Because of the regularity of XPath expressions, automaton based approaches have been used to process XML streams. XPath is only a small subset of XQuery, and in comparison, provides limited functionality. Most of the queries supported by XPath involve only navigation and filtering of the XML stream.

Compared with XPath, XQuery is significantly more expressive, and therefore, more challenging to handle. Currently, some techniques have been proposed for processing XQuery queries over streaming data [58, 53]. Transducer networks have been used in XSM[58] to handle a small subset of XQuery, in which only join and node creation operations are allowed. Flux [53], on the other hand, uses static analysis for optimize buffer size. There are two important limitations in both these efforts. First, neither of them can handle aggregation functions, which we believe can be critical in specifying the type of analysis that is often done on streaming data. Second, neither of them have presented query transformations techniques to reduce the number of traversals, which again can be important for enabling a larger number of queries to be executed correctly on streaming data.

For XML streams with infinite length, a challenging issue is how to determine whether an arbitrary query can be evaluated on-the-fly. An algebraic approach for deciding whether a SQL-like query can be evaluated with a single pass on continuous streams has been proposed recently by Babu and Widom [7]. Their approach cannot handle user-defined aggregates and computations described with binary expressions, which are both frequently used in XQuery. Unlike SQL, developing an algebra to handle complete XQuery is hard. As an example, user defined functions allowed as part of XQuery can be very hard to model
through such an algebra, and we are not aware of any existing effort which is able to do
this.

1.2 Summary of Contributions

The compilation and optimization techniques we proposed in this dissertation provide
a number of potential benefits. First, an application developer can write sequential codes,
using XQuery, to implement data intensive applications on parallel and streaming environ-
ments. Moreover, with our high-level abstraction support, the complex low-level details
of the physical dataset are hidden and application development is further simplified. Fi-
nally, with the optimization and parallelization techniques we developed, large gains in
performance can be achieved. To summarize, we have contributed in the following areas:

- **Optimization techniques for general queries:** In the area of optimization, we
  have developed a new set of optimization and transformation algorithms for XQuery,
  which are based on a new internal representation referred to as Generalized Nested
  Loop (GNL). These optimization techniques include *aggregation rewrite, loop fu-
  sion, loop interchange* and *aggregation remapping*, which are derived from the class
  of transformations known as *loop transformations* that have been widely applied on
  Fortran and other imperative languages [80].

Since XQuery is a very powerful and complex functional language, to enable the
above optimization techniques, we also have developed new algorithms to handle
arbitrary recursive function and the type system in XQuery. Results from several
XMark queries and scientific data processing queries show large improvements from
our optimization techniques.
- **Parallelization of XQuery:** As XML and XQuery are being used for larger datasets, parallelizing XQuery execution can enable faster response. In the area of parallelization, GNL offers a convenient basis for parallelization of XQuery. We present techniques for enumerating parallelization strategies, cost-models for choosing the optimal one, and an algorithm for parallel code generation on cluster of machines. Our detailed experimental evaluation on both synthetic and real datasets shows good parallel speedups in a cluster environment.

- **Supporting High-level Abstractions of Scientific Datasets:** To further simplify application development over scientific datasets, we provide a solution using XML Schemas as a high-level abstraction of a dataset to an application developer. A corresponding low-level Schema describes the actual layout of data and is used by the compiler for code generation. A systematic way for translating the high-level code to a low-level code that achieves high locality and efficient execution is also provided.

- **Efficient processing XQuery over XML streams:** We present a new evaluation and optimization framework specialized for streaming data processing applications expressed using XQuery. Specifically, we introduce the concept of stream data flow graph and apply a series high-level transformation based on it. The goal of these transformation techniques is to enable a single-pass evaluation strategy for the original query. Based on our stream data flow graph, we present a methodology to determine if a query can be evaluated correctly in a single pass. Finally, we propose a new technique to generate efficient streaming code, using our GNL representation. Experimental results from several XMark queries and real data processing applications show large gains from our optimizations.
1.3 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, we briefly introduce some background knowledge about XML, XML Schemas and XQuery. Our targeted applications are also described in this chapter. In Chapter 3, we describe our high-level optimization and parallelization framework. The detailed code generation and parallelization techniques for flat-file datasets are introduced in Chapter 4. In Chapter 5, we propose techniques to use XML as a high-level abstraction, which hides the complexities of physical data layout from the users. For native XML datasets, we present a framework for processing XML streams in Chapter 6, followed by data distribution and parallelization techniques described in Chapter 7. Finally, Chapter 8 concludes the report by summarizing this dissertation and discussing future research directions.
CHAPTER 2

BACKGROUND

In this chapter, we will introduce the background of XML and related technologies developed by W3C. Also, we describe our motivating applications as well as how to express them with XQuery.

2.1 Background: XML, XML Schemas, and XQuery

This section gives background on XML, XML Schemas, and XQuery.

2.1.1 XML and XML Schemas

XML provided a simple and general facility which is useful for data interchange. Though the initial development of XML was mostly for representing structured and semi-structured data on the web, XML is rapidly emerging as a general medium for exchanging information between organizations. For example, a hospital generating medical data may make it available to other health organizations using XML format. Similarly, researchers generating large datasets from scientific simulations may make them available in XML format to other researchers needing them for further experiments.

XML models data as a tree of elements. Arbitrary depth and width is allowed in such a tree, which facilitates storage of deeply nested data structures, as well as large collections
of records or structures. Each element contains *character data* and can have *attributes* composed of *name-value* pairs. An XML document represents elements, attributes, character data, and the relationship between them by simply using angle brackets.

Note that XML does not specify the actual layout of large data on the disks. Rather, if a system supports a certain dataset in an XML representation, it must allow any application expecting XML data to properly access this dataset.

Applications that operate on XML data often need guarantees on the structure and content of data. XML Schema proposals [10, 12] give facilities for describing the structure and constraining the contents of XML documents.

---

*A typical VMPixel Element*  
```xml
<VMPixel>
  <XCoord> ... </XCoord>
  <YCoord> ... </YCoord>
  <Color1> ... </Color1>
  <Color2> ... </Color2>
  <Color3> ... </Color3>
</VMPixel>
```

*Schema Declaration*  
```xml
<archetype name = "VMPixel">  
  <element name="XCoord" type = 'integer'>  
    range="0""999" rangetype="unique_all_incl" >  
  </element>
  <element name="YCoord" type = 'integer'>  
    range="0""999" rangetype="unique_all_incl" >  
  </element>
  <element name="Color1" type = 'char'>  
  </element>
  <element name="Color2" type = 'char'>  
  </element>
  <element name="Color3" type = 'char'>  
  </element>
</archetype>
```

---

Figure 2.1: XML and XML Schema
We now give examples of the use of XML and XML Schemas. The particular example we use is that of a *virtual microscope*, where digitized images are stored and processed. The XML data format and XML schema for this application are presented in Figure 2.1. The logical layout of a \texttt{VMPixel} element is presented at the top of left column. The type of this element is \texttt{VMPixel}. It has 5 fields of character data associated with it, which are the X and Y coordinates (of type integer) and the values of 3 colors (of type character).

The XML Schema declaration for such dataset is shown at the bottom of the left column of Figure 2.1. This schema specifies that the 5 fields, \texttt{XCoord}, \texttt{YCoord}, \texttt{Color1}, \texttt{Color2}, and \texttt{Color3} are all required. It also specifies types of these 5 fields. The current XML Schema proposals [10, 12] do not provide any mechanism for specifying multi-dimensional collections of objects. We propose additional syntax for this purpose. In the Schema declaration, we can add a \texttt{range}, which specifies the range of values an integer field can take. Further, we specify that in the dataset this range is \texttt{unique\_all\_incl}, which means that all values for \texttt{XCoord} and \texttt{YCoord} must be present within the dataset, and no two elements have the same value of both \texttt{XCoord} and \texttt{YCoord}.

### 2.1.2 XML Query Language: XQuery

As stated previously, XQuery is a language currently being developed by the World Wide Web Consortium (W3C). It is designed to be a language in which queries are concise and easily understood, and to be flexible enough to query a broad spectrum of information sources, including both databases and documents.

XQuery is a functional language. The basic building block is an *expression*. Several types of expressions are possible, including:
• Primary expressions, which are the basic primitives of the language, and include literals, variables, function calls, and use of parentheses to control the precedence of operators.

• Path expressions, which are used to locate nodes within a tree.

• Sequences, which are ordered collection of zero or more items.

• Arithmetic expressions, which use the standard arithmetic operators for addition, subtraction, multiplication, division, etc.

• Comparison expressions, which allow comparison between values, nodes, and orders of given nodes or values within a document.

• Constructors, which can create XML documents as a result of a query.

• Conditional expressions, which are based upon the keywords if, then, and else.

• FLWR expressions, which support iteration and binding of variables to intermediate results. FLWR stands for the keywords for, let, where, and return.

• Unordered expressions, which use the keyword unordered. The unordered expression takes any sequence of items as its argument, and returns the same sequence of items in a nondeterministic order.

We illustrate the XQuery language and the for, let, where, and return expressions by an example, shown in Figure 2.2. In this example, two XML documents, depts.xml and emps.xml are processed to create a new document, which lists all departments with ten or more employees, and also lists the average salary of employees in each such department.
for $d in document("depts.xml")//deptno
let $e := document("emps.xml")//emp[deptno = $d]
where count($e) >= 10
return
<big-dept>
  {$d,
  <headcount> { count($e) } </headcount>,
  <avgsal> {avg($e/salary)} </avgsal>
  }
</big-dept>

Figure 2.2: An Example Illustrating XQuery’s FLWR Expressions

In XQuery, a for clause contains one or more variables, each with an associated expression. The simplest form of for expression, such as the one used in the example here, contains only one variable and an associated expression. The evaluation of the expression typically results in a sequence. The for clause results in a loop being executed, in which the variable is bound to each item from the resulting sequence in turn. In our example, the sequence of distinct department numbers is created from the document depts.xml, and the loop iterates over each distinct department number.

A let clause also contains one or more variables, each with an associated expression. However, each variable is bound to the result of the associated expression, without iteration. In our example, the let expression results in the variable $e being bound to the set or sequence of employees that belong to the department $d. The subsequent operations on $e apply to such sequence. For example, count($e) determines the length of this sequence.

A where clause serves as a filter for the tuples of variable bindings generated by the for and let clauses. The expression is evaluated once for each of these tuples. If the resulting
value is true, the tuple is retained, otherwise, it is discarded. A return clause is used to create an XML record after processing one iteration of the for loop. The details of the syntax are not important for our presentation.

To illustrate the use of unordered, a modification of the example in Figure 2.2 is presented in Figure 2.3. By enclosing the for loop inside the unordered expression, we are not enforcing any order on the execution of the iterations in the for loop, and in generation of the results. Without the use of unordered, the departments need to be processed in the order in which they occur in the document depts.xml. However, when unordered is used, the system is allowed to choose the order in which they are processed, or even process the query in parallel.

```
unordered(
  for $d in document("depts.xml")//deptno
  let $e := document("emps.xml")//emp[deptno = $d]
  where count($e) >= 10
  return
    <big-dept>
      {$d,
       <headcount> { count($e) } </headcount>,
       <avgsal> { avg($e/salary) } </avgsal>
      }
    </big-dept>
)
```

Figure 2.3: An Example Using XQuery’s Unordered Expression
2.2 Data-Intensive Reductions and XQuery Representation

In this section, we describe two motivating data processing applications and show how they can be expressed using XQuery. These two applications are, *satellite data processing* [17] and *multi-grid virtual microscope* [2].

2.2.1 Satellite Data Processing

```xml
unordered(
    for $i in ($minx to $maxx)
    for $j in ($miny to $maxy)
        let $p := document("satellite.xml")/data/pixel
            where(( $p/x = $i) and ($p/y = $j ))
            return
                <pixel>
                    <latitude> {$i} </latitude>
                    <longitude> {$j} </longitude>
                    <summary> { accumulate($p) } </summary>
                </pixel>
)

define function accumulate ($p)
    as double
    {
        let $inp := item-at($p,1 )
        let $NVDI := ( ($inp/band1 - $inp/band0) div
            (2*$inp/band1+$inp/band0)+1) * 512
        return
            if( empty($p) )
                then 0
            else { max($NVDI, accumulate(subsequence($p,2))) } } 
```

Figure 2.4: Satellite Data Processing Expressed in XQuery
The first application we focus on involves processing the data collected from satellites and creating composite images. A satellite orbiting the Earth collects data as a sequence of *blocks*. The satellite contains sensors for five different bands. The measurements produced by the satellite are short values (16 bits) for each band.

The typical computation on this satellite data is as follows. A portion of Earth is specified through latitudes and longitudes of end points. A time range (typically 10 days to one year) is also specified. For any point on the Earth within the specified area, all available pixels within that time period are scanned and an application dependent output value is computed. To produce such a value, the application will perform computation on the input bands to produce one output value for each input value, and then the multiple output values for the same point on the planet are combined by a reduction operation. For instance, the Normalized Difference Vegetation Index (ndvi) is computed based on bands one and two, and correlates to the “greenness” of the position at the surface of the Earth. Combining multiple ndvi values consists of execution a max operation over all of them, or finding the “greenest” value for that particular position.

XQuery specification of such processing is shown in Figure 2.4. We currently assume a simplified data representation, where the input data is simply a set of pixels. Each pixel stores the latitude, longitude, time, and 16-bit measurements for the 5 bands. The code iterates over the two-dimensional space for which the output is desired. Since the order in which the points are processed is not important, we use the directive *unordered*. Within an iteration of the nested for loop, the *let* statement is used to create a sequence of all pixels that correspond to the those spatial coordinates. The desired result involves finding the pixel with the best NDVI value. In XQuery, such reduction can only be computed recursively.
2.2.2 Multi-Grid Virtual Microscope

unordered(
    for $i$ in ($x1$ to $x2$)
    for $j$ in ($y1$ to $y2$)
        let $p :=$ document("vmscope.xml")data/pixel[(x=$i)
            and (y = $j) and (scale ≥ $z1) ]
        return
            <pixel>
                <latitude> {$i} </latitude>
                <longitude> {$j} </longitude>
                <summary> { accumulate($p) } </summary>
            </pixel>
)

define function accumulate (element pixel $p$ )
    as element
    {
        if (empty($p) )
            then $null
        else
            let $max:= accumulate(subsequence($p,2) )
            let $q:= item-at($p,1)
            return
                if ($q/scale < $max /scale) or ($max = $null)
                    then $max
                else $q
    }

Figure 2.5: Multigrid Virtual Microscope Using XQuery

The Virtual Microscope [34] is an application to support the need to interactively view and process digitized data arising from tissue specimens. The raw data for such a system is captured by digitally scanning collections of full microscope slides at high power. In a typical dataset available when a virtual microscope is used in a distributed setting, the same
portion of a slide may be available at different resolution levels, but the entire slide is not available at all resolution levels.

A particular user is interested in viewing a rectangular section of the image at a specified resolution level. In computing each component of this rectangular section (output), it is first examined if that portion is already available at the specified resolution. If it is not available, then we next examine if it is available at a higher resolution (i.e., at a smaller granularity). If so, the output portion is computed by averaging the pixels of the image at the next higher level of granularity. If it is only available at a lower resolution, then the pixels from the lower resolution image are used to create the output.

XQuery code for performing such computations is shown in Figure 2.5. We have made two simplifications. First, we assume that each pixel stores the x and y coordinates, as well as the resolution level. Second, we assume that the user is only interested in viewing the image at the highest possible resolution level, which means that averaging is never done to produce the output image. The structure of this code is quite similar to our previous example. Inside an unordered for loop, we use the let statement to compute a sequence, and then apply a recursive reduction.
CHAPTER 3

OVERALL COMPILATION FRAMEWORK

In this chapter, we present a new set of optimization and transformation algorithms for XQuery. These techniques are based upon a new internal representation, which we refer to as Generalized Nested Loop (GNL). GNL is a high-level structure, which still maintains the semantic information of the original query. GNL supports most advanced features of XQuery, including types and user-defined functions. We have the following three goals in designing and using GNL:

Enable New Optimizations: GNL enables new optimizations on XQuery codes, including aggregation rewrite, loop fusion, loop interchange, and aggregation remapping. These optimizations are derived from the class of transformations known as loop transformations that have been widely applied on Fortran and other imperative languages [80].

Enable Parallelization: As XML and XQuery are being used for larger datasets, parallelizing XQuery code execution can enable faster response. A parallel configuration also provides more aggregate memory for query processing. GNL offer a convenient basis for parallelization of XQuery.

Allow Existing Optimizations: GNL preserves high-level information to be used for existing optimizations, such as unnesting and join reordering [52, 64, 57, 21].
The techniques we have designed are suitable to be used for XML datasets that do not have specialized indexing support, and where datasets need to be scanned to retrieve the tuple stream. This includes XML streams, scientific datasets that use XML as a logical interface [55], and other XML datasets without indexing support.

The rest of this chapter is organized as follows. Section 3.1 motivates and defines GNL and presents an algorithm for constructing it. Algorithms for new optimizations are presented in Section 3.2. Parallelization of XQuery is considered in Section 7.3. We briefly argue how GNLs can enable optimizations that have been widely applied on database queries In Section 3.4, and summarize this chapter in Section 3.5.

### 3.1 Generalized Nested Loops

This section defines the GNL representation. We initially motivate the need for such a representation. An algorithm for generating GNL from XQuery syntax is presented at the end of this section.

We illustrate the basic idea of GNL by using a simple example. Consider the following query on a dataset with student records. The query involves computing average score for a particular class, CIS 100.

```xml
let $b = \text{document()}/\text{students/score[@c=cis100]}
return <average>
{sum(b) div count(b)}
</average>
```

When auxiliary structures such as index are not available for the above dataset, one can only retrieve the tuple stream by scanning the entire dataset. A naive processing of the above query would navigate the dataset at each step of the XPath expression, and the
resulting sequences will be merged and sorted according to the document order. Then, two aggregation functions will be applied to the sequence and the final result will be returned.

Once we recognize that the sequence on which \textit{sum} and \textit{count} operate is a subset of the sequence selected by the path expression, we can perform the aggregations together with the sequence selection process. Here, unnecessary sorting is also eliminated. The optimized processing of above query can be expressed like this:

```xml
foreach b in ("roster.xml")/students/score
  if ( score.c = cis100)
    { sum= sum + b.score;
      count = count + 1;
    }
end for;
return sum/count
```

To make such optimized processing possible, we need a semantic structure to specify the set of nodes of interest, along with the operations being performed on this set. Relational or set algebras are good at defining operations between sets and some simple aggregation operations such as sum and count. However, it is hard to capture more complex aggregations, such as variance, using them.

GNL is a semantic structure that explicitly defines:

- The iterative structure used for retrieving the data,
- The aggregation or \textit{reduction} operations to be performed on the qualified data.

By expanding XPath expressions explicitly as an iterative structure, selection and aggregation operations can be optimized together. Moreover, GNLs are also useful for parallelization, as well as optimizations on XPath expressions and nested queries.
3.1.1 Definition

Definition 1 A GNL \( N \) is a five tuple \((I, E_p, E_c, S, G)\) where,

1. \( I \) is the set of variables bound to \( N \),
2. \( E_p \) is the selection path of the corresponding XPath expression,
3. \( E_c \) is the qualifying expression of \( E_p \) (if any),
4. \( S \) is the loop body, which is an ordered sequence of operations performed on any variable \( i \in I \),
5. \( G \) defines other proprieties of the GNL, such as if it is ordered by document.

The semantic meaning of a GNL is similar to a for loop in an imperative programming language. For each element in the target XPath expression \( E_p \) predicated by \( E_c \), the element is bound to the variable \( I \), and each statement \( S_i \in S \) will be executed according to their order in \( S \).

As an example, the GNL for the above query will have the variable set \( I \) contain \( b \), the selection path \( E_p \) will be \( /student/score \), the qualifying expression \( E_c \) will be \( @c = cis100 \), and the loop body contains two statements, \( S_1 \) and \( S_2 \), where \( S_1 \) computes \textit{summary} and \( S_2 \) computes \textit{count}. The property set \( G \) is empty.

Here, a GNL changes the binding semantics of a let clause to those of a for clause. In this way, the selection process is explicitly defined to the compiler. GNLs are somewhat similar to the for loops resulting from expanding XPath expressions in the normalization process of the formal semantics approach [33]. During normalization, the original XPath expression is broken into several for loops, iterating over the result of several axes. The predicate expressions are mapped to an if expression.
However, there are two important differences between GNLs and the normalized for loops. First, a GNL uses a single loop structure and a single path expression, whereas, the formal semantics approach uses several nested for loops iterating over several axes. A representation with a single loop and a single path facilitates the analysis and optimizations we are interested in. Second, the formation of loop body $S$ of a GNL is guided by analysis across function boundaries (using interprocedural analysis or function inlining). The computations being performed in recursive reductions are also analyzed.

Since a GNL only uses a single path expression to represent the iterative operations, the selection of the path expression becomes crucial. Consider the following query.

```xml
let $b := document()/students/
let $c := $b/score
return sum($c)
```

One way to express the above query is by using two nested GNLs, as shown in Figure 3.1. This is similar to the normalized query that the formal semantics approach will create. However, if we remove the variable $b$ with a forward substitution, we can use just
one GNL to represent the query, as shown in Figure 3.1(b). Although both GNLs in Figure 3.1(a) and (b) have the same semantic meaning, GNLs in (b) can help simplify further optimizations.

With the above motivation, we introduce the concept of a critical path:

**Definition 2** A path expression \( \mathcal{E} \) is critical if the variable bound to it is referred to in the subsequent query (excluding XPath expressions), or if it is referred to in two or more XPath expressions.

Only the XPath expressions that are considered critical are mapped to a GNL structure. If an XPath expression is only used in one XPath expression, as shown in the above query, we use forward substitution. If, however, a defined XPath expression is referred to in more than one XPath expression, we will generate a GNL corresponding to the first XPath expression, and then the GNLs involving the reference to this XPath expression will be nested inside the first GNL. The XQuery code in Figure 3.2(a) is such a query, and the corresponding GNLs are shown in Figure 3.5(a).

Although the above GNL mapping method works for XPath expressions defined in a let clause, specific treatment is required for a for clause. In a for clause, the variable is bound to every tuple in the tuple stream, and the return clause is invoked for each tuple. Since such iterative invocation implicitly defines the structure of the result, we assume that each XPath expression defined in a for clause is finally referred. This results in the following lemma.

**Lemma 1** A path expression \( \mathcal{E} \) defined in a for clause is critical.
3.1.2 Generation of GNL

Our algorithm for generation of GNLs involves traversing the syntax tree two times. The first pass is a top-down traversal for generating a Path Dependence Graph (PDG). After generating the PDG, we traverse it to label each node as either critical or non-critical. We map the critical paths in the PDG to GNLs, initially keeping their loop bodies empty. The second pass is a bottom-up travel. Here, expressions from original syntax tree are extracted and inserted into the corresponding GNLs.

To define a PDG, we first define the notion of path dependence.

**Definition 3** A variable $v_2$ is path dependent on variable $v_1$ if $v_1$ is bound to XPath expression $E_1$, $v_2$ is bound to another XPath expression $E_2$, and $v_1$ is referred to in $E_2$.

As an example, in the query shown in Figure 3.2(a), $b$ is dependent on $a$.

**Definition 4** A Path Dependence Graph is a directed graph, $G = (V, E)$, where the vertex $V$ represent variables defined using XPath expressions, and edges $E$ represent path dependence between these variables. There is an outgoing edge $e$ from $v_1$ to $v_2$ if $v_2$ is path dependent on $v_1$. 

---

Figure 3.2: Example of PDG
In this dissertation, we only consider queries for which PDG is a tree or a forest.

The PDG of query shown in Figure 3.2(a) is shown in Figure 3.2(b). Here, $b$ is dependent on $a$, and $c$ and $t$ are dependent on $b$. $t$ is an internal variable assigned to XPath expression $/student/score$, which has no variables associated with it in the original query.

The algorithm for generating PDG is shown in Figure 3.3. A node is labeled as critical during the first traversal, if it can be determined to be so at this time. The algorithm adds each variable defined in a $for$ or $let$ clause in the set $V$. If a variable $v_1$ is used in another XPath expression with associated variable $v_2$, an edge $(v_1, v_2)$ will be added in the set $E$. For an XPath expression without bounding variables, our algorithm assigns a unique internal variable.

To check reference of each variable in $V$, a recursive procedure $process$ is called for each subexpression in a $return$ clause, $where$ clause, and qualifiers of XPath expressions. If a variable $v$ is used in any such expression, the processing depends upon the nature of the expression. We consider three different types of expressions:

1. An XPath expression $e$: Here, $e$ has no variable bound with it. Our algorithm assigns a unique internal variable for each such expression. An edge between $v$ and the new variable is added in $E$.

2. A user defined function call expression $f$: We use the buildPDG algorithm to build a separate PDG by traversing the function definition. We then merge the new PDG with the original PDG by adding an edge between $v$ and the root of the new PDG into $E$.

3. Other expressions: The node corresponding to $v$ is labeled as critical, since it will be used for aggregation.
buildPDG
Input: a XQuery Query Q with FLWR expression
Output: the Path Dependence Graph of Q
\[ V = \emptyset ; E = \emptyset ; \]
Let Exp = FLWR expression of Q
\[ \text{foreach } \text{(for } v_i \text{ in } e_i \text{)} \text{ do } \{
\]
\[ \text{if not } v_i \in S \]
\[ V = V \cup \{v_i\} \]
\[ \text{label } v_i \text{ as critical} \]
\[ \text{buildEdge}(v_i, E_i) \} \]
\[ \text{foreach } (\text{let } \_ = \) \text{do } \{
\]
\[ \text{if not } S \&\& E_i \text{ is XPath expression} \]
\[ V = V \cup \{v_i\} \]
\[ \text{buildEdge}(v_i, E_i) \}
\[ \text{else process}(e_i) \}\]
\[ \text{foreach } \text{other expression } e_i \]
\[ \text{process}(e_i) \]
end

buildEdge(v,e)
Input: Variable v; XPath Expression e
Construct edges of PDG
\[ \text{if } e \text{ start with } v_2 \]
\[ \text{if } v_i \in V \]
\[ E = E \cup \{v_i, v_i\}; \]
\[ \text{let } e_f = \text{filter expression of } e \]
\[ \text{process}(e_f) \]
end

process(e)
Input: Expression e
\[ \text{if } e \text{ is an XPath expression } \{ \]
\[ \text{assign a new variable } v_i \text{ in } V \text{ for } e \]
\[ \text{if } e \text{ start with } v_i \]
\[ E = E \cup \{v_i, v_i\}; \} \]
\[ \text{else if } e \text{ is function call expression } \{ \]
\[ \text{buildPDG}(e) \]
\[ \text{foreach } e_i \text{ as argument of } e \]
\[ \text{process}(e_i) \} \]
\[ \text{else if } e \text{ is variable } \]
\[ \text{if } e \in V \]
\[ \text{label } e \text{ as critical in } G = (V, E) \]
\[ \text{else foreach } \{e_i\} \text{ of } e \]
\[ \text{process}(e) \]
end

Figure 3.3: Algorithm to Generate Path Dependence Graph from XQuery
In the PDG we construct, a node $v$ is labeled as critical if it is referred to in the subsequent expression for processing or aggregating the tuples. Also, by Lemma 1, nodes corresponding to \textit{for} clauses are labeled as critical. However, some critical nodes are still left unlabeled. These are the internal nodes that are used by two or more XPath expressions. We use a simple algorithm which traverses the PDG from bottom-up and labels these nodes as critical. If a node has more than one child labeled as critical, then it is also labeled as critical.

After the PDG has been correctly labeled, we can map each node that represents a critical node to a GNL. The nesting of GNLs represents the path dependence relationship
between two nodes. If there is a dependence edge from \( v_2 \) to \( v_1 \), the GNL corresponding to \( v_2 \) will be nested inside the GNL of \( v_1 \). As shown in Figure 3.4, the GNLs of \( c \) and \( t \) are nested in GNLs of \( b \). They are treated as statements of the body of GNLs corresponding to \( b \). Nodes in PDG that are not labeled as critical are not translated into GNLs. For example, the \textit{let} clause defining the variable \( a \) no longer has effect in the final GNLs.

To use GNL for an entire program, other expressions in XQuery syntax also need to be placed inside the GNL. These expressions include the \textit{where} clause, \textit{return} clause and the \textit{let} clause that are not reflected on the PDG. We place each such expression inside the GNL that corresponds to the most recent \textit{for} clause prior to it. Furthermore, we create a dummy GNL that serves a similar role as the \textit{main} function in an imperative language like C. Expressions that do not have a \textit{for} clause prior to them are inserted in this GNL. Inside the body of a GNL, all expressions, including the nested GNLs, are arranged according to their order in the original query.

### 3.2 New Optimizations

In this section, we describe various optimization that can be applied on GNLs. Overall, we assume that auxiliary data structures for fast retrieval are not available, and each step of a XPath expression requires scanning the dataset one tuple at a time.

By definition, the tuple stream that a GNL operates on is specified by its path expression \( E_p \) and filter expression \( E_c \), and the aggregation operations should be specified by the statement sequence in its body. This provides two kinds of opportunities for query optimization: one involves rewriting the control structures between loops, and the other exploits local optimizations inside a loop body.
3.2.1 Aggregation Rewrite for GNLs

Each GNL built by the algorithm we presented in the previous section defines the tuple stream it selects, and the nested structure defines the relationship between each tuple stream. However, as we mentioned earlier, we change the binding semantics of a let clause, i.e., the variable is no longer bound to the tuple stream as a whole, but to each tuple instead.

For this reason, we need to discover aggregation operations in the body of the GNL and rewrite them into operations that can be applied a tuple at a time. The algorithm for this purpose is shown in Figure 3.6 and is applied to each expression in the body of a GNL. We assume that any aggregation operation on a tuple stream is performed through a function, which could be user-defined or internal. For each function call expression \( e \), if there is exactly one actual argument that corresponds to a critical node in the PDG, we move the function call inside that GNL by adding a new expression \( temp := e \) to its body. The original expression is replaced with a reference to \( temp \). Functions operating on more than one tuple streams can be processed by function inlining.

The algorithm traverse the tree in a bottom-up fashion. Before a function call is moved, its arguments are traversed first. Figure 3.5 shows the GNLs after code motion.

After code motion is done, each expression that invokes an aggregation function is moved into the body of a GNL retrieving the tuples to be aggregated. For internal functions such as sum, count, and average which have pre-defined meaning, we can easily rewrite them in an iterative way. For example, we can rewrite sum() as \( tmp = tmp + v \), where \( tmp \) is a temporary reduction variable and \( v \) is the tuple. For a user-defined function, including recursive function, we apply a static analysis technique to extract the reduction operation from the definition of the function [56], which will be described in Chapter 4. The basic idea is to examine the syntax tree from leaves and apply tree pattern matching.
to retrieve the desired sub-tree. The final GNLs with aggregation operations is shown in Figure 3.5(a). Here, aggregation functions average() and sum() are moved inside the corresponding GNLs, and have been rewritten accordingly.

\[
\text{traverse} \\
\text{Input: XQuery Expression } e, \text{ PDG } =(V,E) \\
\text{if } e = "f \{e_i\}" \{ \\
\text{foreach } e_i \\
\text{traverse}(e_i) \\
\text{if } \bigcup \{e_i\} \cap V = e_j \{ \\
\text{let } e'_j = "temp_{p_j} = f\{e_i\}" \\
\text{add}(\text{GNL}(e_j), e'_j) \\
\text{replace } e_j \text{ with } temp_{p_j} \\
\} \\
\text{else foreach primary expression } e_i \text{ of } e \\
\text{traverse}(e_i) \\
\text{end}
\]

Figure 3.6: Algorithm to Perform Code Motion of Aggregation Functions

### 3.2.2 XPath Optimization by Loop Fusion

Consider the GNLs shown in Figure 3.5(a). The two innermost GNLs retrieve tuples from the same location, which is specified by /$b$/score. The filtering conditions involved are different. If we find out this information by examining $E_p$ of both GNLs, we can fuse the two GNLs into one a single GNL. In the new GNL, the original aggregation operations defined by both the original GNLs are invoked if the corresponding filter condition $E_c$ is qualified. By this transformation, only one navigation of the path involved is required.

For supporting such a transformation, we introduce the notion of conformable for GNLs:
**Definition 5** Two GNLs $A$ and $B$, where $A$ is executed before $B$, are conformable for fusion if

1. The path expressions $E_p^A$ and $E_p^B$ are equivalent.
2. $B$ does not refer to, either directly or indirectly, any variable that is defined inside $A$.

The two nested GNLs, involving $c$ and $d$, respectively, in Figure 3.5(a) are conformable. This is because they are both defined on `//students/score`, and the GNL $d$ does not use any variable returned by GNL $c$. By definition, they can be fused into one GNL, and the query result will not change. The GNLs after performing loop fusion is shown in Figure 3.5(b).

![Diagram](image)

(a) Formation 1  
(b) Formation 2  

Figure 3.7: Example of Greedy Loop Fusion

Although two conformable GNLs can be merged into one by loop fusion, loop fusion does not always improve performance. Consider the GNLs shown in Figure 3.7(a). In this example, GNL $b$ and GNL $c$ are conformable, since they both work on the same path, and there is no dependence between them. However, if we merge these two GNLs, as shown in Figure 3.7(b), the computation of `count` is now nested inside GNL $a$. Thus, this loop fusion
reduces the number of times a path is navigated, but introduces redundant computations. To address this issue, we develop two schemes for loop fusion, referred to as conservative fusion and greedy fusion, respectively.

**Conservative Fusion**

In conservative fusion, our goal is to not introduce any redundant computation, even if some opportunities for reducing path navigations are lost.

Therefore, in conservative fusion, two conformable GNLs are fused if none of them is nested inside other GNLs, or they are nested inside the same GNL with the same nesting level.

**Lemma 2** *Conservative Fusion will not introduce redundant computation.*

During a loop fusion, redundant computations are generated by merging two GNLs at different nesting levels. Therefore, computations inside the GNL with fewer nested parents are invoked more times than necessary. By applying the above restriction in conservative fusion, two conformable GNLs cannot be merged unless they are at the nested inside same GNL. The two GNLs shown in Figure 3.5 can be fused in conservative fusion.

The detailed algorithm for conservative fusion is shown in Figure 3.8. Loop fusion is performed from top to down, i.e., GNLs with the nesting level zero are checked and conformable GNLs are fused. We use a new GNL to replace the two merged GNLs. After no more fusion can be performed at a given nesting level, for each remaining GNL, we apply the same algorithm for its nested GNLs.

**Greedy Fusion**

Unlike conservation fusion, greedy fusion attempts fusing all possible GNLs, irrespective of their nesting level. An example of greedy fusion is shown in Figure 3.7(b).
conservative_fusion
Input: a GNL
\( G = \emptyset \)
do {
    foreach GNL \( g_i \) with level 0
        \( G = \bigcup \{g_i\} \)
    foreach \((g_j, g_k) \neq k \in G \times G \)
        if \( g_j \) and \( g_k \) are conformable
            \( g = \text{fuse}(g_j, g_k) \)
            \( G = G - \{g_j, g_k\} \)
            \( G = G \cup g \)
} until \( G \) does not change
foreach \( g_i \in G \)
    conservative_fusion(\( g_i \))
end

Figure 3.8: Algorithm for Conservative Loop Fusion

In greedy fusion, we are trying to fuse a GNL \( g_1 \) with another conformable GNL \( g_2 \), where \( g_2 \) can be nested inside other GNLs. Since \( g_1 \) and \( g_2 \) have different nesting levels, computations inside \( g_1 \) will be executed more times than necessary. However, if the tuple stream that \( g_1 \) and \( g_2 \) operate on is quite large or cannot be cached in memory, it is still quite likely that loop fusion will increase performance. Whether greedy fusion can achieve performance gain or not depends on the characteristics of the dataset and the query. For an application that involves very large datasets, we can often assume that reducing the navigation of datasets is the primary goal. In such cases, we can apply greedy fusion.

The detailed algorithm for greedy fusion is shown in Figure 3.9. It is different from conservative fusion in that for each GNL \( g_i \) with nesting level zero, we check all other GNLs, including the nested GNLs, to find conformable GNLs. If there are multiple candidates conformable with \( g_i \), we simple chose one with the lowest nesting level. Among candidates at the same nesting level, the choice can be arbitrary. Like conservative fusion,
this algorithm is also recursively invoked. A GNL is checked for possible fusion before GNLs in its body are checked.

**greedy_fusion**
Input: a GNL

\[ G' = \emptyset \]

do {  
  foreach GNL \( g_i \) with level 0
  \[ G = \cup \{ g_i \} \]
  foreach \( g_i \in G \)
  \[ G_t = \emptyset \]
  foreach nested GNL \( g_j^k \) of \( g_j \in (G - \{ g_i \}) \)
  if \( g_i \) is conformable with \( g_j^k \)
  \[ G_t = G_t \cup \{ g_j^k \} \]
  \[ g = \text{fuse}(g_j, g_i) \]
  \[ G = G - \{ g_j \} \]
  let \( g_k = \min(G_t) \)
  until G does not change
  foreach \( g_k \in G \)
  greedy_fusion(\( g_k \))
end

Figure 3.9: Algorithm for Greedy Loop Fusion

### 3.2.3 Interchange of GNLs

Consider the following query that computes the total score of all students in each course in computer science.

```xml
unordered(
  for $a$ in document() / course/cis
  let $b$: =document() / student/score[@c=$a]
  return sum($b))
```

The GNLs generated for the above query are shown in Figure 3.10(a). It is easy to see that the resulting code can be quite inefficient, particularly when the record of students are
much larger than that of the courses. This is because each execution of the *let* expression will involve a complete scan over the tuples of students. However, if we swap the nesting orders of the two GNLs, as shown in Figure 3.10(b), we can navigate the record of students only once to compute the total score.

In general, GNL interchange can improve the performance of a query if the size of inner tuple stream is either much larger than the outer one, or cannot be held in memory. For GNLs generated from XQuery, the interchange of two nested GNLs in valid if

- Maintaining of document order is not required, and

- There is no data dependence that prevents GNL interchange.

To interchange two nested GNLs, the inner GNLs $g_2$ is swapped to the outside with outer GNL $g_1$ moved to the inside. The body of $g_2$ now becomes body of $g_1$. Special treatment is needed for filter expressions. If the filter expression $E^2_p$ of $g_2$ does not depend on the variable bound to $g_1$, $E^2_p$ can be moved together with $g_2$, or $E^2_p$ must function as a *where* clause in the new GNLs, as shown in Figure 3.10(b).
3.2.4 Aggregation Remapping

The last optimization we describe in this section is aggregation remapping. For some queries with pre-defined iteration space of GNLs, further optimizations can be applied after GNL interchange. Consider the following simple query:

```xml
unordered(
  for $a$ in (85 to 100)
    let $b$: =//student/score[@grade=$a]
    return count($b))
```

![Diagram](image.png)

(a) Formation 1                                                                                     (b) Formation 2

Figure 3.11: Example of Aggregation Remapping

Like the previous example, we can perform a GNL interchange here to reduce the number of times the dataset needs to be scanned. The GNL after the interchange is shown in Figure 3.11. After the GNL interchange, because the boundary of the inner GNL can be easily decided by our compiler, we can perform another optimization here. This maps the iterative operations defined by the inner GNL to aggregations over the output. By doing this optimization, the inner GNL is collapsed and the final query will contain only one a single GNL.

To perform aggregation remapping, we first construct an abstract output space corresponding to the nested GNLs after the interchange. Then, for each data tuple read in the
outer GNL, regardless of document order, we extract a necessary and sufficient condition for which \( i \) in the abstract output space is updated. If such a condition cannot be found, or the abstract output space cannot be determined, we cannot perform the transformation. Otherwise, we try to remove the inner GNL and use the extracted condition to directly update the output. To remove the inner GNL with bounding variable \( j \), we must replace any reference of \( j \) in the extracted condition by analyzing and rewriting the conditional clauses. If the variable of the inner GNL cannot be completely removed or replaced, then we do not perform further transformations and leave the original GNLs unchanged. The GNL after aggregation remapping is shown in the Figure 3.11.

### 3.3 Parallelization of XQuery

As XML is being used for larger datasets, storing the datasets and parallelizing the execution of XQuery is going to be important. In this section we will discuss parallelization of XQuery on shared-nothing architectures.

To parallelize a XQuery query represented by GNLs, we need to 1) identify a GNL which can be executed in parallel, and 2) generate parallel code for local processing on each node, and a global combination across the nodes.

#### 3.3.1 Identifying the Optimal Parallelization Scheme

To try and achieve the best performance, the following two-step approach is used. First, we enumerate the different ways in which a given GNL can be parallelized. Next, we use a cost model that computes disk access, memory access, and communication costs associated with each scheme.

The first step involves deciding which GNLs can be executed in parallel. Our compiler uses techniques from the existing literature on parallel compilation. We consider two
types of parallel loops, one in which the iterations are independent, and the one in which associative and commutative operations are involved.

We focus on parallel GNLs which are not nested inside another GNL, or the GNL they are nested inside is not parallel. If such a GNL has other GNL nested inside, there could be multiple ways of parallelizing this GNL. To do this, we enumerate all possible parallelization schemes and estimate the overheads on each node by a cost model. The parallelization scheme with the least overall cost is chosen. We define total cost of parallel execution on each node to be:

\[
T_{\text{total}} = n_d \times f_d(D) + n_m \times f_m(M) + n_c \times f_c(C)
\]

Here, \( n_d, n_m, \) and \( n_c \) denote total volumes of disk accesses, memory accesses, and interprocessor communication, respectively. \( f_d, f_m \) and \( f_c \) are feedback functions that compute the average unit latency of disk access, memory access, and communication, given a particular volume of these. These functions are generated by profiling and are dependent on system characteristics such as page size and size of total memory.

To calculate the cost, we first analyze the GNLs and calculate the number of memory and disk accesses. We use information on GNLs, the Schema information, and the information on the datasets. To simplify our cost model and analysis, we make some assumptions about data reuse. Instead of deciding whether each single data or memory access is a reuse or not, we use feedback functions to reflect the effects of data reuse as a whole. If the input parameters to feedback functions \( f_d \) or \( f_m \) are less than a certain threshold, the output of the function will be much smaller, indicating that cache or disk buffering is in effect.

We will use the example in Figure 3.12 to explain our approach. There are two loops that can be parallelized in this example. For each parallelizable loop, there are two ways to arrange the loop structure: the selected parallel GNL can be moved up as the outer loop
Figure 3.12: XQuery code and GNL for Satellite Processing

<table>
<thead>
<tr>
<th>Parallel Scheme</th>
<th>Disk</th>
<th>Cost per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(1) outer</td>
<td>$D$</td>
<td>$\frac{D}{P} \times f_e(D) + \left[ \frac{D}{P} \times f_m(1) \right] + \left[ \frac{D}{P} \times f_c(\frac{D}{P}) \right]$</td>
</tr>
<tr>
<td>L(1) inner</td>
<td>$\frac{D}{P}$</td>
<td>$\frac{D}{P} \times f_e(\frac{D}{P})$ + $\left[ \frac{D}{P} \times f_m(\frac{D}{P}) \right]$ + $\left[ \frac{D}{P} \times f_c(\frac{D}{P}) \right]$</td>
</tr>
<tr>
<td>L(2) outer</td>
<td>$\frac{D}{P}$</td>
<td>$\frac{D}{P} \times f_e(\frac{D}{P})$ + $\left[ \frac{D}{P} \times f_m(N) \right]$ + $\left[ \frac{D}{P} \times f_c(\frac{D}{P}) \right]$</td>
</tr>
<tr>
<td>L(2) inner</td>
<td>$\frac{D}{P}$</td>
<td>$\frac{D}{P} \times f_e(\frac{D}{P})$ + $\left[ \frac{D}{P} \times f_m(1) \right]$ + $\left[ \frac{D}{P} \times f_c(N) \right]$</td>
</tr>
</tbody>
</table>

$D$: size of dataset; $P$: number of nodes;

Table 3.1: Cost Estimation for Different Parallelization and Optimization Schemes

or pushed down as the inner loop. Thus, we have four possible schemes for the parallel execution, each with different costs associated with disk accesses, memory access, and communication. The costs are shown in Table 3.1.

For example, in the Scheme 3, GNL(2) is chosen for parallelization and moved to be the outer loop, using GNL interchange. In this scheme, the original dataset is distributed to $P$ nodes, with each node processing $1/P$ of the original data and aggregating on $N$ output values. The volume of communication is also $N$ since each node needs to broadcast...
its output to other nodes. In the Scheme 1, where GNL(1) is chosen for parallelization, the output buffer is distributed to P nodes while each node maintain a whole copy of the original dataset. As a result, the communication volume is reduced to $N/P$.

From Table 3.1, we can see that the Scheme 3 has the least disk space requirements and the minimum disk access cost. On the other hand, the Scheme 1 has the least memory usage with minimum memory cost. The best scheme that may be chosen depends upon the datasets. For example, the Scheme 3 may be ideal if the dataset size is very large.

To efficiently support processing on large disk-resident flat-file datasets and on a shared-nothing architecture, our compiler uses a runtime system called Active Data Repository (ADR) [17, 16]. The details of parallel code generation is described at the next chapter.

### 3.4 Other Optimizations

In this section, we briefly argue how GNL serves as a convenient basis for a number of optimizations that have been considered on database queries.

During the translation from XQuery to GNLs, only critical paths are mapped to GNLs, and aggregation operations with inlined functions are moved inside GNLs and rewritten to imperative statements. This results in a very simple intermediate representation, as compared to, for example, the normalized XQuery in the formal semantics approach [33]. Since each GNL can be viewed as a tuple-stream with explicitly defined aggregations, joins are represented as nested GNLs, and can be processed directly in tuple-at-a-time fashion as specified by its nesting structure. If document order is not required, we can treat each GNL as *set loops*. Thus, joins can be optimized by the same techniques used in join optimizations for nested loops in OO database programming languages [57, 21, 30] and transforming nested query blocks in SQL [52, 64]. The basic idea is to break up a GNL into two or
more nested GNLs by *loop fission*, where each performs one step of the joins and stores the temporary values in a set. For join operations, we can estimate the size of the intermediate result and perform loop fission on GNLs to perform the smallest one first.

### 3.5 Summary

This chapter has focused on a number of optimizations and transformations that are very useful when XQuery is used for aggregations and reductions on large datasets, which do not have any index associated with them. The techniques are based on a new representation, which we refer to as a Generalized Nested Loop (GNL). We have presented algorithms for a number of optimizations, including aggregation rewriting, loop fusion, GNL interchange, and aggregation remapping. Moreover, we have also argued how GNLs can be used for optimizations like join reordering and unnesting that have been widely applied on database queries.

As XML and XQuery are being used for large datasets, executing queries in parallel can provide faster responses. Using GNLs, we have shown how XQuery can be parallelized. We have presented techniques for enumerating parallelization schemes, cost-models to choose the best scheme, and an algorithm for parallel code generation.

In the next chapter, we will introduce supporting analysis techniques to enable optimization and parallelization techniques based on GNL. Detailed code generation algorithms as well as experiment results will also be presented.
CHAPTER 4

DETAILED ANALYSIS ALGORITHMS AND IMPLEMENTATION

We have described our overall framework and various techniques for optimizations and parallelization of XQuery in the previous chapter. While optimizations and parallelization can be conveniently performed once queries are expressed by using GNL, this detailed implementation process involves a number of challenges. Because XQuery is a functional language, the only practical way for specifying reduction computations is using recursion. This, and some of the other related features of XQuery, lead to the following compiler analysis and restructuring challenges:

- Analysis of recursive functions to identify reduction computations involving only associative and commutative operations.

- Replacement of recursive functions with iterative constructs, which is required to perform advanced transformations such as aggregation rewriting.

- Parallelization of generalized reduction functions, which particularly requires the synthesis of global reduction functions.

- Translation of XQuery processing to an imperative language like C/C++, which is required for using a middleware that offers low-level functionality. Particularly, the
challenge is to deduce the types of data-structures to be used in the imperative language.

We report our solutions towards the above problems in this chapter, which is organized as follows. The various analysis methods that we use for implementation are described in Section 4.1. We evaluate our compiler and the techniques in Section 4.2 and conclude in Section 4.3.

4.1 Detailed Analysis Methods for Implementation

In this section, we describe the various analysis, transformation, and code generation issues that are handled by our compiler. Initially, we summarize the challenges involved in the compilation process.

4.1.1 Overview of the Implementation Problems

Consider the codes shown in Figures 2.4 and 2.5. The computations of these two applications are specified in recursive functions, which in most cases are the only way to defined general reductions in a functional language.

Once we can recognize that the computation in the recursive loop is a reduction operation involving associative and commutative operators only, we can transform the recursive function into iterative loops. A direct benefit of it is that by replacing recursion with iteration, we can reduces the overhead of function calls. However, a more significant advantage is that the iterations of the resulting loop can be executed in any order. Since such a loop is inside an unordered nested for loop, powerful restructuring transformations can be applied. Particularly, the code resulting after applying aggregation remapping, which is described in the previous chapter, will only require a single pass on the entire dataset. Thus, the first
define function accumulate ($p$)
    return element
    {
        if (empty($p)) then 0
        else if (item-at($p,1) \geq 0)
            max (item-at($p,1),acumulate(subsequence($p,2)))
        else
            max(0,acumulate(subsequence($p,2)))
    }

(a)

define function accumulate ($p$)
    return element
    {
        if (empty($p)) then 0
        else if (item-at($p,1) \geq 0)
            max (item-at($p,1),abs(acumulate(subsequence($p,2)))))
        else
            acumulate(subsequence($p,2))
    }

(b)

define function accumulate ($p$)
    return double
    {
        if (empty($p$) ) then 0
        else
            let $val := accumulate( subsequence($p,2) )$
            let $q := item-at($p,1)$
            return
                if ($q < $val) then $val$
                else $q$
    }

(c)

Figure 4.1: Examples To Illustrate Reduction Analysis Algorithm

key compiler analysis and transformation tasks is recognizing that the recursive function involves a reduction computation and transforming such a recursive function into a \textit{foreach loop}, i.e., a loop whose iterations can be executed in any order.
For datasets that are stored as flat files, there are two ways in which XQuery is likely to be compiled. The first will be to translate it into an imperative language like C/C++, for which efficient compilers are available. The second will be to generate code for a middleware or runtime system, which offers data handling capabilities. Such systems also often offer interfaces based upon imperative languages. Thus, XQuery codes will likely need to be translated into codes in an imperative language. This leads to a new challenge, which is type inferencing. We need to deduce the types of the data-structures used in the target languages. The type system used in XQuery is significantly different from that in popular imperative languages, which makes the type inferencing problem a non-trivial one.

One of our goals is to parallelize XQuery codes. For the class of applications that we are targeting, this particularly requires the synthesis of global reduction functions. A global reduction function combines the results obtained from applying reductions independently on subsets of data elements on different processors. As we will show later, the analysis required for this is closely related to the tasks listed above.

4.1.2 Analysis of Recursive Functions

We now focus on analyzing recursive function with the following three goals: 1) identifying reduction computations involving associative and commutative operators, 2) replacing them with a foreach loop, and 3) extracting a global reduction function.

Our analysis requires that the recursive function has the following canonical form:

```latex
define function F($t) \{ 
    \text{if (p1) then } F1($t) 
    \text{else } F2(F3($t), F4(F(F5($t)))) 
\}
```
where,

- The input $t$ must be a sequence,
- $F_5$ must be a subsequence of $t$, including all but the first element of $t$ (i.e. subsequence($t,2$) in XQuery),
- Each of $F_1(t)$ and $F_3(t)$ must return constant values or should be functions of only the first element of the sequence $t$.

We use a series of examples to explain the algorithm we have developed. These examples are presented in Figure 4.1. Our algorithm is presented in Figure 4.2.

Our algorithm analyzes the abstract syntax tree (AST) of the function. Initially, it processes all leaf nodes of the tree. We focus on nodes of two types, the nodes that are a recursive call to the function $F$ and the nodes that denote a variable defined by such a recursive call. In either of these cases, if the node is used as part of a return value, the node is inserted into the set $S$. Consider the examples in Figure 4.1. For each of the cases (a) and (b), our algorithm will insert two nodes in the set $S$. For the case (c), our algorithm initially find three nodes that are of interest, the recursive call node, the use of $\text{val}$ inside the conditional, and the use of $\text{val}$ as a return value. However, since only the last of these three is part of a returned value, it is the only one added to the set $S$.

Note that a recursive reduction function may compute multiple simple types, each through a different reduction operation. For example, an averaging function may compute a sum field and a count field, applying reduction operations $\text{add}$ and $\text{add by one}$, respectively. Our algorithm separates the recursive calls used for computing each distinct field. Thus,
Analyze($AST \ T$) \{ \\
$S = \phi$ \\
\text{for each leaf node } n \text{ in } T \\
\begin{align*} 
& \text{if } n \text{ is a recursive function call node} \\
& \quad \begin{cases} 
& \text{if } n \text{ is used as part of a returned value} \\
& \quad S = S \cup \{ n \}
\end{cases} \\
& \quad \text{else if } n \text{ is a variable defined by a recursive call} \\
& \quad \begin{cases} 
& \text{if } n \text{ is used as part of a returned value} \\
& \quad S = S \cup \{ n \}
\end{cases}
\end{align*}
\}

Partition nodes in $S$ based upon the field computed 
\text{let } S_1, \ldots, S_k \text{ denote the } k \text{ partitions} 
\begin{align*} 
& \text{for } i = 1, \ldots, k \\
& \quad R = \phi \\
& \text{for each node } n \text{ in } S_i \\
& \quad R = R \cup \{ \text{Findnode}(n) \} \\
& \text{if } S_i \text{ and } R \text{ are singleton sets} \\
& \quad \text{let } R = \{ t \} \\
& \quad \text{if } t \text{ is an associative and commutative operation} \\
& \quad \quad \text{mark the sub-tree with } t \text{ as the root as the reduction operation} \\
& \quad \text{else mark the function as non-transformable} \\
& \text{else} \\
& \quad \text{if each node in } S_i \text{ is in different control path and} \\
& \quad \quad \text{each node in } R \text{ is the same associative and commutative operation} \\
& \quad \quad \text{let } t' \text{ be the least common ancestor of all nodes in } R \\
& \quad \quad \text{mark the sub-tree with } t' \text{ as the root as the reduction operation} \\
& \quad \text{else mark the function as non-transformable} \\
& \}

\text{Findnode(node } n) \{ \\
& \text{let } p = \text{parent of } n \\
& \quad \text{if } p \text{ has at least two children including } n \\
& \quad \quad \text{return } p \\
& \quad \quad \text{else return Findnode}(p) \\
& \}

**Figure 4.2: Algorithm for Analysis of Recursive Functions**

the set $S$ is partitioned into $k$ subsets, $S_1, \ldots, S_k$. For each of three examples in Figure 4.1, 
the value of $k$ is 1.

Our algorithm subsequently processes each of these sets independently. For each node 
$n$ in a set $S_i$, we apply the function $\text{Findnode}(n)$. This function finds the closest ancestor
of \( n \) that has more than one child. The node thus obtained combines the value at \( n \) with another value. In Figure 4.1, case (a), the application of the function \( \text{Findnode} \) to the two nodes in the set \( S_i \) finds the two occurrences of the \( \text{max} \) function. In case (b), the results are the \( \text{max} \) function and the \( \text{else if} \) clause. In case (c), the result is the \( \text{if} \) statement.

Next, we need to consider several different cases. If the set \( S_i \) has only one element, let \( t \) be the result of \( \text{Findnode} \) to this element. If \( t \) denotes an associative and commutative operation, and if the function matches the canonical structure shown above, then we know that the recursive function performs a reduction computation. Here, we mark the sub-tree with \( t \) as the root as the reduction function. This function is used for transforming the recursive function to a \( \text{foreach} \) loop and for performing the global reduction.

In the example (c) in Figure 4.1, the node \( t \) such determined is an \( \text{if} \) statement, and cannot automatically be determined by the compiler to be an associative and commutative operation. However, if we have additional information that the recursive function uses associative and commutative operations, we could still use the sub-tree rooted at \( t \) as the reduction operation. Several languages use annotations for marking reduction computations, including HPF-2 [43] and a data parallel dialect of Java we have used in our earlier work [35].

If the set \( S_i \) includes more than one element, we need to check for several conditions. We need to ensure that each node in \( S_i \) is in a different or \textit{mutually exclusive} control path, i.e., invocation of the function results in at most one recursive call. If no language annotation is available for the compiler, we require that the results of \( \text{Findnode}(n) \) for each \( n \) in \( S_i \), i.e. the nodes in the set \( R \), denote the same associative and commutative operation. In example (a), the two nodes in the set \( R \) are \( \text{max} \) functions. Therefore, our algorithm can determine that the recursive function performs reduction computations. In this case, we
find the least common ancestor of the two \( \text{max} \) function nodes. The sub-tree rooted at this node is used as the reduction operation.

If a language annotation is available to determine that the function involves a reduction computation, we do not need to check if each node in the set \( R \) is the same function, or an associative and commutative function. We can simply use the sub-tree rooted at the least common ancestor of the nodes in the set \( R \) as the reduction operation.

If the function performs reduction computations, either as determined by our compiler or through language annotations, we use the sub-tree identified by our algorithm for creating equivalent \textit{foreach} loops. In Figure 4.3, we show the three \textit{foreach} loops corresponding to the three recursive functions shown earlier in Figure 4.1. These examples use constructs that are internal to the compiler, and do not correspond to valid XQuery codes. The procedure used for doing this transformation is as follows. The \textit{output} element is initialized with the value of the function \( F1 \) in our canonical form. We iterate over all elements in the sequence and apply the reduction function extracted from the abstract syntax tree. All occurrences of \texttt{subsequence}(s_p, 2) are replaced by the output element and the result is assigned to the output element.

The \textit{foreach} loop thus obtained can also be executed in parallel, by initializing the \textit{output} element on each processor. The reduction function extracted by our algorithm can then be used for combining the values of \textit{output} created on each processor.

### 4.1.3 Code Generation for ADR

To efficiently support processing on large disk-resident datasets and on a cluster architecture, our compiler generates code for a runtime system called Active Data Repository.
function accumulate ($p)
{
    $Output := 0
    foreach element $e in $p {
        if ($e ≥ 0)
            $Output := max($e,$Output)
        else
            $Output := max(0,$Output)
    }
    return $Output
}

(a)

function accumulate ($p)
{
    $Output := 0
    foreach element $e in $p {
        if ($e ≥ 0)
            $Output := max ($e,abs($Output))
    }
    return $Output
}

(b)

function accumulate ($p)
{
    $Output := 0
    foreach element $e in $p {
        if ($e < $Output) then
            $Output := $Output
        else
            $Output := $e
    }
    return $Output
}

(c)

Figure 4.3: Replacing Recursion By Iteration

(ADR) [17, 16]. We now briefly describe the functionality of ADR and the issues in generating code for ADR.

ADR targets processing of datasets that are stored as flat files. In ADR, the processing of a loop that involves a generalized reduction progress through the following three phases:
(1) *Initialization* – output elements (possibly replicated on all processors) are allocated space in memory and initialized, (2) *Local Reduction* – input disk blocks on the local disks of each processor are retrieved and aggregated into the output disk blocks, and (3) *Global Combine* – if necessary, results computed in each processor in phase 2 are combined across all processors to compute final results. To facilitate these, a *query planning* phase is used, in which a schedule for processing the query is created. The schedule is particularly used for minimizing disk seek time, and for aggressively using asynchronous I/O operations.

ADR run-time support has been developed as a set of modular services implemented in C++. The processing associated with an application is specified by providing a set of functions. The key functions that our compiler needs to generate are: 1) a *range function* that specifies the part of the multi-dimensional dataset that is processed, 2) a *mapping function* that maps an element to a loop iteration (or an output element), and 3) a *reduction function* for aggregating the values.

### 4.1.4 Parallel Code Generation

We briefly summarize our solution to one problem that arises in generating parallel code. This relates to computations inside GNL that involve associative and commutative operations. Here, after local processing on each node, a *global combination* needs to be performed. In this step, the copies of output on all nodes are combined. Although the global combination function performs the same associative and commutative operation that is defined in a GNL body, we cannot use the body of a GNL simply as the global combination function. This is because the input to a global combination function is copies of output from other nodes, and not the data tuples from the original dataset.
Our compiler uses an algorithm for this purpose which we summarize below. The basic idea is to extract a part of the code that impacts the output, either directly or indirectly, from the GNL. This is done by a static analysis technique called program slicing [78]. After a program slice is extracted, certain statements are either removed or modified. These include

- Statements that create control dependence on the input data. Because input data is not visible at the global phase, control dependence on input data tuples can be discarded.

- Statements that create data dependence on the input data. For the same reason, we can replace each instance of the input data (bound to the GNL variable) with references to remote copies of output from other nodes.

The remaining statements are used in the body of the global combination function. The detailed algorithm is shown in Chapter 7, where parallelization of XQuery based on general parallel libraries such as MPI is presented.

### 4.1.5 Type Analysis and Conversion

As stated previously, our compiler generates code for Active Data Repository (ADR) [17, 16], which uses C++ virtual functions to specialize the processing for a particular application. To generate C++ code for a XQuery program, one challenging task is the correct and efficient conversion of various types in XQuery to corresponding C++ types. This problem needs to be addressed by any compiler for XQuery whose target code is in an imperative or object-oriented language.
XQuery is a statically typed functional language. The type annotations of each operand, argument, and function are always either explicitly declared or can be generated while validating against a Schema. This static feature of the type system provides various possibilities for compiler optimizations and analysis, such as the well studied method for static type checking [13], and in our case, static type analysis for translation to an imperative language.

Although type systems of both C++ and XQuery are static, the mapping between these two is not straightforward. There are several issues that make the translation problem challenging.

```
typeswitch ($pixel) {
  case element double_pixel
    return max( $pixel/d_value,0)
  case element integer_pixel
    return max($pixel/i_value,0)
  default 0
}
```

Figure 4.4: XQuery expression with different types

First, unlike variables in an imperative language, an expression in XQuery may be associated with values of several different types. An example of such an expression is shown in Figure 4.4. The expression here may return either a double or an integer, depending upon the type of the variable `pixel`. To perform the translation, our compiler needs to collect the static types of all possible branches and compute a union of these types. For union of simple types, we will generate a corresponding union type in the target code. For union of complex types, we use the polymorphism of C++ language.
The second issue is related to the parametric polymorphism of XQuery. An actual argument of a function is only required to be of a subtype of the declared type of the corresponding formal function parameter. For example, the \textit{max} function in Figure 4.4 can be invoked with an argument whose type may be either integer or double. Moreover, some parameters may be declared as \textit{AnyT}\textit{ype}, implying that their type can only be known by validating against an XML Schema. Therefore, to infer the type of a formal parameter, we need to gather information about the actual arguments from call sites, and if necessary, from Schema definitions. In cases where such parametric polymorphism is used, we use \textit{function cloning}, i.e., we generate a copy of the function for each distinct type we infer. We believe this gives better performance than using polymorphism of C++, considering the limited occurrences of such functions in the actual cases we studied.

Overall, our static type analysis algorithm is based on constraint-based type inference algorithm [44, 3], and the static type checking algorithm listed as part of the specification of XQuery formal semantics [28]. The goal of this algorithm is to infer the collection of all possible static types for a given expression, which will be used to guide the code generation in our compiler. The algorithm is top-down and recursive, since the static type of an expression depends only on the types of its sub-expressions.

The first step in our algorithm is to initialize the type of each expression, if the type is explicitly defined in the XQuery program or in an XML Schema. After initialization of type variables, we will analyze possible types for an expression by propagating type variables from subexpressions according to possible run-time data flow. Specially, if the target expression has only one possible value, the outcome is just propagated from its operands. For operands with different but compatible types, a \textit{least upper bound} for these compatible
Figure 4.5: Algorithm for Static Type Analysis

types is returned as the result. If the target expression is a function call expression, information about the types of the actual arguments needs to be gathered from the call sites. This is done by applying the algorithm for each actual argument. The resulted types are assigned to each formal parameter, and the function body of the corresponding function definition is processed by the algorithm. For any other expression whose value is defined by subexpressions in more than one branch, we simply apply the algorithm for all sub-expressions in each branch and compute a union for all possible results. The detailed algorithm is listed in Figure 4.5.
In our static algorithm, special consideration should be given to recursive functions. If the return type of a recursive function is declared as $\textbf{AnyType}$, it is generally hard to infer specific return types from actual parameters [29]. This is because the inference algorithm may never terminate. For this reason, we require that all recursive functions be first transformed into iterative constructs, and then our algorithm be applied.

---

class pixel
{
    ...
    ...
    struct tmp_result_1 {
        union {
            double r1;
            int r2;
        };
    };
    ...
    tmp_result_1 t1;
    if (pixel.tag $\text{double}_{\text{pixel}}$
        t1.r1 = max_1(pixel.d_value,0);
    else if (pixel.tag $\text{integer}_{\text{pixel}}$
        t1.r2 = max_2(pixel.i_value,0);
    else
        t1.r2 = 0;

    Figure 4.6: Compiler Generated C++ code

After finishing type analysis for each expression in a XQuery program, we can generate C++ code that correctly implements the type system of XQuery. The C++ code generated after type analysis and code generation for the code in Figure 4.4 is shown in Figure 4.6. Because the $\text{types}_{\text{switch}}$ expression may return either a double or an integer type, a $\text{union}$ type is declared to keep the result of this expression. The variable $\$\text{pixel}$ may be bound to
two complex types when validated against a Schema. Therefore, we declare a superclass
for $\textit{pixel}$, from which two subclasses can be derived. Also in this example, because the
actual arguments of the function $\textit{max}$ can be either double or integer, a clone of the function
is generated for each type.

4.2 Experimental Results

To evaluate our techniques and current prototype XQuery compiler, we have conducted
two sets of experiments. In the first set, we evaluate the impact of the various transforma-
tions we have developed. In the second set, we measure parallel performance for a set of
queries.
Figure 4.9: Evaluating the Benefit from Transformations, satellite application

Figure 4.10: Evaluating the Benefit from Transformations, vscope application

4.2.1 Queries for Our Experiments

We used eight different queries for our experiments. Four of these are taken from the XMark benchmark [73]. Specifically, we used the queries 5, 11, 12, and 20 from this set, as they were suitable for the transformations we are targeting.

The last four queries involve processing of scientific datasets. sum is a very simple sum reduction, where values associated with pixels having the same $x$ and $y$ coordinates are added together. irrreg is a simple irregular reduction, where the values associated with edges are used to increment the values associated with corresponding nodes. satellite involves processing of data collected from satellites and creating composite images [17]. vscope is an application to support interactive viewing and processing of digitized data arising from tissue specimens [34].
While the techniques we have presented are general, our implementation supported processing of flat-file datasets. For this purpose, some additional techniques described in the next chapter were used. The datasets for execution of XMark queries, as well as sum and irreg were synthetically generated. The experiments with satellite and vscope involved using real satellite-generated and medical images, respectively.

### 4.2.2 Impact of Transformations on Sequential Execution

This subsection evaluates the performance gains from the transformations we have implemented. Our experiments were conducted on a 933 MHz Pentium III workstation, with 256 MB of RAM, and running Linux version 7.1.

We compared sequential performance between two versions, opt and naive. The naive version is used as a baseline for our comparison, but already includes several optimizations from the original semantics of XQuery. Specifically, path navigation, sort, and
merge are not performed in this version. The opt version includes the transformations using GNLs we described in Section 3.2 and 3.4. Particularly, the queries 11 and 12 from XMark benefit from join optimization and the query 20 benefits from loop fusion. satellite, vscope, sum and irreg benefit from GNL interchange, followed by aggregation remapping and recursion analysis. Note that in the case of query 20 from XMark and satellite and vscope queries, our comparison between opt and naive shows the performance gains resulting from the optimizations we have introduced.

Figure 4.7 shows the benefits of loop fusion on the query 20 from the XMark set. Experiments were conducted on four different datasets, with the sizes of 150 MB, 300 MB, 600 MB, and 1.2 GB. Loop fusion improves the execution rate by 50% for the 150 MB dataset (which can be cached in memory), and by a factor of three for the larger datasets.
Table 4.1: Benefit of Optimizations on Queries 11 and 12

<table>
<thead>
<tr>
<th></th>
<th>Opt. (sec.)</th>
<th>Naive (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 11</td>
<td>0.47</td>
<td>12.92</td>
</tr>
<tr>
<td>Query 12</td>
<td>0.42</td>
<td>6.74</td>
</tr>
</tbody>
</table>

The gains from transformations on queries 11 and 12 are shown in Table 4.1. Only one datasets, with the size of 1.3 MB was used. The performance improvement is a factor of 28 for the query 11, and a factor of 15 for the query 12.

The results from `sum` are presented in Figure 4.8. The output produced by this kernel is a $2 \times 2$ array. Thus, the `naive` version requires 4 passes over the entire data, whereas, the `opt` version requires only a single pass. The difference between the `opt` and `naive` versions is consistently a factor of 4. Thus, it appears that the execution time is dominated by memory and disk access times.

The results from `irreg` kernel are presented in Figure 4.11. The output of this application is a $12 \times 12$ array. The difference between the two versions is a factor of 4 for the 150 MB dataset, and a factor of 8 for each of the other three datasets. Note that the first dataset can be cached on memory, therefore, subsequent passes over the data only require memory accesses. In comparison, the other three dataset cannot be cached and multiple passes over the dataset result in more disk accesses. This accounts for higher difference in performance for the larger datasets.

The results from `satellite` and `vscope` are presented in Figures 4.9 and 4.10, respectively. Here, the use of GNL interchange followed by aggregation remapping reduces the number of passes on the input data to 1. The output of `satellite` kernel is a $6 \times 3$ array and the output of `vscope` kernel is a $3 \times 3$ array. For `satellite`, the difference
in performance between opt and naive versions is a factor of 7 for the 150 MB dataset, and between 16 and 18 for other datasets. This shows that when the data cannot be cached in main memory, the execution time is dominated by disk access time. Thus, increasing the number of passes on data result in almost linear increase in execution times. The results from vscope are very similar. The difference in performance is a factor of 4.5 for the 150 MB dataset, and then, nearly a factor of 9 for other datasets.

4.2.3 Results from Parallel Execution

In this subsection, we present results from executing parallel code that is generated by our compiler. We use the queries 5 and 20 from the XMark set, along with satellite and vscope. We ran our test programs on a cluster of 900 MHz Pentium III based nodes connected by a gigabit Ethernet. Each node has 256 MB of main memory and 18GB of local disk. We ran our experiments on 1, 2, 4, 8 nodes of the cluster.

The results from the parallelization of XMark queries is shown in Figure 4.13. Queries 5 and 20 were executed on datasets whose size was 4 GB and 6 GB, respectively. Both queries achieve linear speedups on up to 8 nodes.

For the satellite application, the dataset contains an image of $29,238 \times 28,800$ pixels collected at 5 different magnification levels, which corresponds to 3.3 GB of data. The query we use generates an image of $20,000 \times 20,000$. This requires reading nearly 3 GB, and generating an output of nearly 1.6 GB. The dataset for the satellite application contains data for the entire earth at a resolution of $1/128^{th}$ of a degree in latitude and longitude, over a period of time that covers nearly 15,000 time steps. The size of the dataset is 2.7 GB. The query we use corresponds to a region of $6,000 \times 20,000 \times 20,000 \times 40,000$ points. This involves reading nearly 1.7 GB data and generating a 400 MB output.
The results from vmscope are presented in Figure 4.14. The results from satelite are presented in Figure 4.12. We show two versions, referred to as comm and no-comm, for each of these cases. comm corresponds to complete parallel execution, including the communication required for performing the global reduction. no-comm excludes the communication required in the end.

In each of the cases, the no-comm version shows linear speedups. This implies that parallel versions show good load balance, and do not include any overheads besides communication. However, as the number of processors is increased, the cost of communication starts dominating. Though the performance of comm versions continues to improve, the overall speedups are not very high.

4.3 Summary

In this chapter, we described various analysis techniques that are required for optimizations and parallelization based on GNL. Particularly, we have shown how we can generate code for an existing runtime system, ADR, and translate XQuery for execution on disk-resident datasets and a parallel cluster. Results from parallelization of two real applications show that parallel versions achieve good load balance, and incur no significant overheads besides communication. In the next chapter, we will present technologies to further simplify the process of developing data intensive application by using high-level abstraction.
CHAPTER 5

HIGH-LEVEL ABSTRACTIONS THROUGH XML TECHNOLOGY

We have described various techniques for efficient optimization and parallelization of XQuery in previous chapters. However, when the datasets are disk-residents, understanding the layout and maintaining high locality in accessing them is crucial for obtaining a reasonable performance. While the traditional relational database technology supports high-level abstractions and standard interfaces, it is suitable more for storing and retrieving datasets, and not for complex analysis on such datasets [72]. NetCDF [71] and HDF [42] are among the frequently used data formats for scientific data. While such formats enable sharing of scientific data, they do not hide the complexities of data layout from the users.

In this chapter, we describe the use of XML technologies for supporting high-level programming methodologies for processing scientific datasets. We particularly show how XML Schemas can be used to give a high-level abstraction of a dataset to the application developers, who can use such a high-level Schema for developing the applications. A corresponding low-level Schema describes the actual layout of data, but is hidden from the programmers. The compiler can use the source code, the low-level Schema, and the mapping from the high-level Schema to the low-level Schema for code generation.

Two key compiler techniques are required for supporting such an approach. First, we need a systematic way to translate the high-level code to the low-level code. Second, we
need to transform the generated low-level code to achieve high locality and efficient execution. This chapter describes our approach to these two problems, which has been implemented in our compilation system. By using Active Data Repository [16, 17] as the underlying runtime system, we offer an XML based front-end for storing, retrieving, and processing flat-file based scientific datasets in a cluster environment.

The rest of the chapter is organized as follows. Our overall system architecture is described in Section 5.1. In Section 5.2, we give examples of high-level and low-level XML Schemas of satellite processing and Virtual Microscope, and also show how they can be expressed in XQuery. The various analysis methods that our compiler uses are described in Section 5.3. We evaluate our compiler and the techniques in Section 5.4. and conclude in Section 5.5.
5.1 System Overview

In this section, we briefly introduce the overall architecture of our system. This discussion forms the basis for our description of the various compilation phases. An overview of the system is shown in Figure 5.1.

Our target environment is a cluster of machines, each with an attached disk. To efficiently support processing on large disk-resident datasets and on a cluster architecture, our compiler generates code for a runtime system called Active Data Repository (ADR) [17, 16]. ADR run-time support has been developed as a set of modular services implemented in C++, which targets processing of datasets that are stored as flat files. Our system does not directly process XML datasets. As a physical lay-out standard, XML involves several-fold storage overheads. Therefore, for scientific applications that involve large datasets, XML is only beneficial as a logical lay-out standard. Here, the key advantage of XML technologies is that XML Schemas allow the users to view the data at a high-level. Consequently, an XML query language like XQuery can be used for specifying the processing a high-level, i.e., keeping it independent of the details of the low-level layout of data.

In our system, a XML schema is provided to users for developing applications using XQuery. This XML Schema is called the high-level XML schema, because it describes a high-level abstraction of the dataset and does not expose any details of the physical layout of the dataset. The original dataset, which is stored as a flat file, will be distributed to disks of a cluster architecture by using data distribution and indexing services provided by ADR. A low-level XML Schema file reflecting the physical layout and meta-data information will be provided. High-level XML Schemas are known to the programmers when developing XQuery code, and will be used by the compiler for XQuery type checking. Low-level
XML Schemas will guide the compiler in generating efficient codes executing on the disk-resident datasets. More details and examples of high-level and low-level Schemas will be given in the next section.

5.2 High-level and Low-level Schemas and XQuery Representation

This section focuses on the interface for the system. We use two motivating examples, satellite data processing [17] and the multi-grid virtual microscope [2], for describing the notion of high-level and low-level schemas and XQuery representation of the processing.

5.2.1 Satellite Data Processing

```xml
<xs:element name="pixel" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="x" type="xs:integer" />
      <xs:element name="y" type="xs:integer" />
      <xs:element name="date" type="xs:date" />
      <xs:element name="band0" type="xs:short" />
      <xs:element name="band1" type="xs:short" />
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 5.2: High-Level XML Schema for Satellite

The first application we focus on involves processing the data collected from satellites and creating composite images.
The XML Schema shown in Figure 5.2 provides a high-level abstraction of the satellite data. The pixels captured by the satellite can be viewed as a sparse three dimensional array, where time, latitude, and longitude are the three dimensions. Pixels for several, but not all, time values are available for any given latitude and longitude. Each pixel has 5 short integers to specify the sensor data. Also, latitude, longitude, and time is stored within each pixel. With this high-level XML Schema, a programmer can easily define computations processing the satellite data using XQuery.

XQuery specification of the data processing is shown in Figure 2.4. The code iterates over the two-dimensional space for which the output is desired. Since the order in which the points are processed is not important, we use the directive unordered. Within an iteration of the nested for loop, the let statement is used to create a sequence of all pixels that correspond to the those spatial coordinates. The desired result involves finding the pixel with the best NDVI value. In XQuery, such reduction can only be computed recursively.

5.2.2 Multi-Grid Virtual Microscope

As described earlier, the Virtual Microscope [34] is an application to support the need to interactively view and process digitized data arising from tissue specimens.

The digitized microscope slides can also be viewed as a three dimensional dataset. Each pixel has x and y coordinates and the resolution is the third dimension. The high-level XML Schema of virtual microscope is shown in Figure 5.3. For each pixel in a slide, three short integers are used to represent the RGB colors.

XQuery code for performing the computations is shown in Figure 2.5. We assume that the user is only interested in viewing the image at the highest possible resolution level, which means that averaging is never done to produce the output image. The structure of
this code is quite similar to our previous example. Inside an unordered for loop, we use the
\textit{let} statement to compute a sequence, and then apply a recursive reduction.

\subsection*{5.2.3 Low Level XML Schema and XQuery}

The above XQuery codes for multi-grid virtual microscope and satellite data processing
specify a query on a high-level abstraction of the actual datasets, which eases the develop-
ment of applications. However, storing XML data in such a high-level format will result in
unnecessary disk space usage as well as large overheads on query processing. For example,
storing \(x\) and \(y\) coordinates for each pixel in a regular digitized slide of virtual microscope
is not necessary, since these values can be easily computed from the meta-data and the
offset of a pixel.

In our system, pixels in each flat file are later partitioned and organized into chunks by
data distribution and indexing services of ADR. A low-level XML Schema file is provided
to the compiler after partitioning of the datasets to specify the actual data layout. The low-level XML Schema of multigrid virtual microscope is shown in Figure 5.4. As we can see from the example, pixels are divided into chunks. Each chunk is associated with a bounding box for all pixels it contains, which is specified by a lower bound and a higher bound.
Within a chunk, the values of pixels are stored consecutively, with each pixel occupying three bytes for RGB colors.

```xml
define function getData( $x, $y )
return element {
  let $p := //data/chunks[($x ≥ lowbound/x) and ($y ≥ lowbound/y) and ($x ≤ highbound/x) and ($y ≤ highbound/y)]
  let $seq := $p/vmpixel
  let $pixel := item-at($seq, ($y-$p/low/y)*
    (506 div exp(2, div($x-$p/lowbound/x)) + ($x - $p/lowbound/x))
  return
    <pixel>
      <latitude> {$x} </latitude>
      <longitude> {$y} </longitude>
      <scale> {$scale} </scale>
      <color1> {$pixel/color1} </color1>
      <color2> {$pixel/color2} </color2>
      <color3> {$pixel/color3} </color3>
    </pixel>
  }

(a)

define function getData( $x )
return element {
  ...
}

(b)
define function getData( $x, $y, $z )
return element {
  ...
}

(c)

Figure 5.5: getData functions for Multigrid Virtual Microscope
```

After data distribution and indexing services is done, we provide several library functions written in XQuery to perform data retrieval. These library functions have a common name, `getData`, but the function parameters are different. Each `getData` function implements a unique selection operation based on its parameters. The `getData` functions are
similar to physical operators of a SQL query engine. A physical operator of SQL engine takes as input one or more data streams and produces an output data stream. In our case, the default input data stream of a \textit{getData} function is the entire dataset, while the output data stream is result of filtering the input stream by parameters of the \textit{getData} function. For example, the \textit{getData} function shown in Figure 5.5 (a) returns pixels whose $x$ and $y$ coordinates are equal to those specified by the parameters. The detailed implementation is based on the meta-data of the dataset, which is specified by the low-level XML Schemas. The \textit{getData} function in Figure 5.5 (b) requires only one parameter, which retrieves pixels with specified $x$ coordinate. For space reason, the detailed implementation of only one \textit{getData} function is shown here.

The XQuery code for virtual microscope that calls a \textit{getData} function is shown in Figure 5.6. This query code is called \textit{low-level XQuery} and is typically generated automatically by our compiler. The XQuery codes described in the above section operate on high-level data abstractions and are called \textit{high-level XQuery}. The recursive functions used in both the low-level and high-level XQuery are the same.

The low-level XML Schemas and \textit{getData} functions are expected to be invisible to the programmer writing the processing code. The goal is to provide a simplified view of the dataset to the application programmers, thereby easing the development of correct data processing applications. The compiler translating XQuery codes obviously has the access to the source code of the \textit{getData} functions, which enables it to generate efficient code. However, an experienced programmer can still have access to \textit{getData} functions and low-level Schemas. They can modify the low-level XQuery generated by the compiler, or even write their own version of \textit{getData} functions and low-level XQuery codes. This is the
unordered(
  for $i$ in ($x1$ to $x2$)
    for $j$ in ($y1$ to $y2$)
      let $p := \text{getData} (\; $i, $j \; )$
        where (scale $\geq$ $z1$) and (scale $\geq$ $z2$ )
      return
        <pixel>
          <latitude> { $i$ } </latitude>
          <longitude> { $j$ } </longitude>
          <summary> { accumulate($p$) } </summary>
        </pixel>
)
if we can perform aggregation remapping described in earlier chapter, by which only one single pass on the dataset is required. However, because low-level XQuery code presents physical details of the datasets, aggregation remapping from the low-level code will be more complicated. Thus, the two key compiler analysis and transformation tasks are: 1) transforming high-level XQuery codes to efficient low-level query codes, and 2) performing aggregation remapping on the resulted low-level query, restructuring the nested unordered loops to require only a single pass on the dataset.

5.3.2 High Level XQuery Transformation

High-level XQuery provides an easy way to specify operations on high-level abstractions of dataset. If the low-level details of the dataset is hidden from a programmer, a correct application can be developed with ease. However, the performance of the code written in this fashion is likely to be poor, since a programmer has no idea how the data is stored and indexed.

To address this issue, our compiler needs to translate a program expressed in the high-level XQuery to low-level XQuery. As described earlier, a low-level XQuery program operates on the descriptions of the dataset specified by the low-level XML Schemas. Although the recursive functions defined in both high-level and low-level XQuery are almost the same, the low-level XQuery calls one or more getData functions defined externally. getData functions specify how to retrieve data streams according to meta-data of the dataset. A major task for the compiler is to choose a suitable getData function to rewrite the high-level query.

The challenges for this transformation are compatibility and performance of the resulting code. This requires the compiler to determine: 1) which of the getData functions can
be correctly integrated, i.e., if a getData function is compatible or not, and 2) which of the compatible functions can achieve the best performance.

We will use virtual microscope as an example to further describe the problem. As shown in Figure 2.5, in each iteration, the high-level XQuery code retrieves a desired set of elements from the dataset first, then, a recursive function is applied on this data stream to perform the reduction operation. There are three getData functions provided, each will retrieve an output data stream from the entire dataset. The issue is if and how the output stream from a getData functions can be used to construct the same data stream as used in the high-level query.

For a given getData function $G$ with actual arguments $x_1, x_2, \ldots, x_i$, we define the output stream of $G(x_1, x_2, \ldots, x_i)$ to be

$$O\left(\frac{x_1, x_2, \ldots, x_k}{G}\right)$$

Similarly, for a given query $Q$ with loop indices $I_1, I_2, \ldots, I_j$, we define the data stream that is processed in a given iteration to be

$$O\left(\frac{I_1, I_2, \ldots, I_k}{Q}\right)$$

Let the set of all possible iterations of $Q$ be $I_Q$. We say that a getData function $G$ is compatible with the query $Q$ if there exists an affine function $f(y_1, y_2, \ldots, y_j)$, such that

$$\forall I_1, I_2, \ldots, I_j \in I_Q, \exists x_1, x_2, \ldots, x_i$$

such that

$$f(I_1, I_2, \ldots, I_j) = (x_1, x_2, \ldots, x_i)$$

and

$$O\left(\frac{x_1, x_2, \ldots, x_i}{G}\right) \supseteq O\left(\frac{I_1, I_2, \ldots, I_j}{Q}\right)$$
If a `getData` function $G$ is compatible with $Q$, it means that in any iteration of the query, we can call this `getData` function to retrieve a data stream from the dataset. Since this data stream is a superset of the desired data stream, we can perform another selection on it to get the correct data stream. Here, the second selection can be easily performed in memory and without referring to the low-level disk layout of the dataset. For the three functions shown in Figure 5.5, it is easy to see that the first two functions are compatible. Their selection criteria is either less or equally restrictive to what is used in the high-level query.

Because of the similarities between physical operators of SQL engine and our `getData` functions, the technique we proposed for translation from high-level XQuery to low-level XQuery is based on relational algebra. Relational algebra is an unambiguous notation for expressing queries and manipulating relations and is widely used in the database community for query optimization.

We use the following three step approach. First, we compute the relational algebra of the high-level XQuery and `getData` functions. A typical high-level XQuery program retrieves desired tuples from an XML file and performs computations on these tuples. We focus on the data retrieval part. The relational algebras of XQuery and the `getData` functions are shown in Figure 5.7 (a). Here, we use $\sigma(f)E$ to represent selection from the entire dataset $E$ by applying restriction $f$.

In the second step, we formalize these relational algebras into an equivalent canonical form that is easier to compare and evaluate. The canonical form we choose is similar to the disjunctive normal form (DNF), where the relations are expressed as unions of one or more intersections. Figure 5.7 (b) shows the equivalent canonical forms transformed. The actual canonical forms are internally represented by trees in our compiler.
In the third step, we compare the canonical forms of the high-level query and `getData` functions. For a given `getData` function, if its canonical form is an isomorphic subtree of the canonical form of the query, we can say that the `getData` function is compatible with
the original query. This is because when replacing part of the relational algebra of the high-level query with a *getData* function, the query semantics are maintained. From Figure 5.7 (b) it is easy to see that the first two *getData* functions are compatible. *getData*(\$x, \$y, \$z) is not compatible, because the its selection restriction on \$z is *equal*, while the restriction of the query on \$z is \( \geq \) and \( \leq \).

The next task is to choose the *getData* function which will result in the best performance. The algorithm we currently use is quite simple. Because applying restrictions early in a selection can reduce the number of tuples to be scanned in the next operation, a compatible *getData* function with the most parameters is preferred here. Formally, we select the function whose relational algebra in the canonical form is the largest isomorphic subtree. As shown in Figure 5.7 (c), the final function we choose is *getData*(\$x, \$y). The resulting relational algebra for low-level XQuery is shown in Figure 5.7, part (c). Here, the pixels are retrieved by calling *getData*(\$x, \$y) and then performing another selection on the output stream by applying the restriction on *scale*.

### 5.3.3 Aggregation Remapping at Low-level

The low-level code is very inefficient because of the need for iterating over the entire dataset a large number of times. If the dataset is disk-resident, it can mean extremely high overhead because of the disk latencies. Even if the dataset is memory resident, this code will have poor locality, and therefore, poor performance.

Since the input dataset is never modified, it is clearly possible to perform aggregation remapping we described earlier to require single pass of the dataset. However, the major challenge here is how to compute the mapping function. It requires the knowledge of data access patterns and the low-level data layout information. Because a *getData* function used
in the low-level XQuery provides details of the the pixels or elements that are retrieved, our compiler performs static analysis on the \textit{getData} function codes.

Specifically, our algorithm uses the following two step mapping. 1) Mapping from output tuples of a \textit{getData} function to its arguments, i.e, given an arbitrary element \( E \), we determine the arguments that can be used by this \textit{getData} function to retrieve \( E \). 2) Mapping from arguments of \textit{getData} functions to the iteration space, i.e, for an arbitrary combination of arguments of a \textit{getData} function, determine in which iteration it is called.

The first mapping requires inverting the \textit{getData} function, which can generally be done by static analysis. We will show our analysis algorithm using the \textit{getData} function shown in Figure 5.5. First, we extract every restriction on the return value or XML path expressions defined by any function argument. By using slicing and forward substitution, we try to remove intermediate variables.

From the \textit{getData} function defined in Figure 5.5, we can extract the following restrictions.

\[
\begin{align*}
\$\text{pixel} &= \text{item} - \text{at}(/\text{data/chunks/vmpixel}, \nonumber \\
&\quad (\$y - /\text{data/chunks/lowbound/y}) * \nonumber \\
&\quad (506\text{div}(\exp(2, (4\text{div} /\text{file/scale})))) + \nonumber \\
&\quad (\$x - /\text{file/chunks/lowbound/x}) \nonumber \\
\$x &\geq /\text{data/chunks/lowbound/x} \nonumber \\
\$y &\geq /\text{data/chunks/lowbound/y} \nonumber \\
\$x &\leq /\text{data/chunks/highbound/x} \nonumber \\
\$y &\leq /\text{data/chunk/highbound/y}
\end{align*}
\]
In these equations, $\text{pixel}$ is the return value of the function, and $x$ and $y$ are the function parameters. These XML path expressions can be mapped to the meta-data defined by low-level XML Schema, and thus can be treated as constant. For example, $/\text{data/chunks/highbound}/x$ specifies the $x$ coordinate of the higher point of the bounding box of each chunk, which is available at runtime. The $\text{item-at}$ function in equation (1) specifies the offset of the pixel in a chunk, which can also be determined at runtime.

After extracting the above equations, we then use the Omega library [50] to solve these equations. For a given pixel, our goal is to determine how $x$ and $y$ are defined, using meta-data and the offset of the pixel in a chunk. The resulting code after applying the Omega library functions is the first mapping function we want to compute.

```
for $i$ in ($x1$ to $x2$)
  for $j$ in ($y1$ to $y2$)
    Initialize output[$i$, $j$]
    foreach element $e$ in //data/chunks/vmpixal
      if ($/\text{data/scale} \geq z1$) and ($/\text{data/scale} \leq z2$)
        $i$ = $/\text{data/chunks/low}/x + (\text{offset} \div 512...))$
        $j$ = $/\text{data/chunks/low}/y + (\text{offset} \mod 512...))$
      if ($i \geq x1$) and ($i \leq x2$) and
        ($j \geq y1$) and ($j \leq y2$)
        Apply the reduction function and update output[$i$, $j$]
```

Figure 5.8: Data-Centric Transformations on Virtual Microscope Code

For our target class of applications, the second mapping is typically trivial. In general, we only handle affine mappings from arguments of a $\text{getData}$ function to the iteration
space. In virtual microscope, the invocation of $getData (x, y)$ can be mapped directly to the iteration $(x, y)$. By combining this information with the first mapping function, we can determine the mapping function from an element in the dataset to the iteration space. The results of performing data-centric transformation on the virtual microscope are shown in Figures 5.8. This code requires only one scan of the entire dataset.

5.4 Experimental Results

This section reports experimental data from our current compilation system. We used the two real applications, satellite and mg-vscope, discussed in chapter 2. The cluster we used had 700 MHz Pentium machines connected through Myrinet LANai 7.0. We ran our experiments on 1, 2, 4, 8 nodes of the cluster.
Table 5.1: Comparison of Lines of Code

<table>
<thead>
<tr>
<th>Application</th>
<th>XQuery</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mg-vscope</td>
<td>25</td>
<td>502</td>
</tr>
<tr>
<td>Satellite</td>
<td>21</td>
<td>488</td>
</tr>
</tbody>
</table>

The first goal of our experiment was to show the ease of programming provided by our system. For this purpose, the lines of codes for XQuery implementations of our two applications were compared against manual C++ implementations based on the ADR runtime system. Performance of these manually implemented applications were reported in earlier work [36].

The lines of code of the XQuery implementations of satellite and mg-vscope applications, as shown in Table 5.1, are 25, and 21 respectively. On the other hand, the C++ implementations of these two applications based on the ADR runtime system both have around 500 lines of codes. After carefully examining the manual implementations, we found that the majority of the codes were used for data retrieval and parallel communications. In the XQuery implementations, such operations are hidden to the programmers, and can be automatically generated by the our compiler. As a result, the development time and efforts spent on XQuery implementations can be reduced dramatically by our system.

Our second goal was to demonstrate that even with high-level abstractions and a high-level language like XQuery, our compiler is able to generate reasonably efficient code. Again, the compiler generated codes for our two applications were compared against manual implementations reported in [36]. These versions were generated by a compiler starting
from a data parallel dialect of Java, and were further manually optimized. For our discussion, the versions generated by our current compiler are referred to as comp and the baseline version is referred to as manual.

For the mg-vscope application, the dataset we used contains an image of $29,238 \times 28,800$ pixels collected at 5 different magnification levels, which corresponds to 3.3 GB of data. The query we used involves processes a region of $10,000 \times 10,000$ pixels, which corresponds to reading 627 MB and generating an output of 400 MB. The entire dataset for the satellite application contains data for the entire earth at a resolution of $1/128^{th}$ of a degree in latitude and longitude, over a period of time that covers nearly 15,000 time steps. The size of the dataset is 2.7 GB. The query we used traverses a region of $15,000 \times 10,000 \times 10,000$ which involves reading 446 MB to generate an output of 50 MB.

The results from satellite are presented in Figure 5.9. The results from mg-vscope are presented in Figure 5.10. For both the applications and on 1, 2, 4, and 8 nodes,
the comp versions are slower. However, the difference in performance is only between 5% and 8% for satellite and between 18% and 22% for mg-vscope. The speedups on 8 nodes is around 6 for both versions of satellite and around 4 for both versions of mg-vscope. The reason for limited speedups is the high communication volume.

To understand the differences in performance, we carefully compared the comp and manual versions. Our analysis shows that a number of additional simple optimizations can be implemented in the compiler to bridge the performance difference. These optimizations are, function inlining, loop invariant code motion, and elimination of unnecessary copying of buffers.

5.5 Conclusions

In this chapter, we have described a system that offers an XML based front-end for storing, retrieving, and processing flat-file based scientific datasets. With the use of aggressive compiler transformations, we support high-level abstractions for a dataset, and hide the complexities of the low-level layout from the application developers. Our preliminary experimental results from two applications have shown that despite using high-level abstractions and a high-level language like XQuery, the compiler can generate efficient code. In the next chapter, we will focus on how to efficiently process native XML datasets in a streaming environment.
CHAPTER 6

XQUERY COMPILATION IN A STREAMING ENVIRONMENT

6.1 Introduction

Recently, a new model of data processing has also emerged in the database community. In this data model, data arrives in the form of continuous streams, usually from a data collection instruments or a long running computer simulation. The data needs to be analyzed in real-time, and using only a single pass on the data. Many important applications classes, like protecting network security, monitoring critical infrastructure, analyzing stock and business data, monitoring climate and environment involve analysis of streaming data [41, 8, 83]. Because of the popularity of XML as a data exchanging format, processing and querying XML streams has become an important topic. We believe that there are two other important trends which also contribute to the need for processing XML streams. The first is related to distributed and grid-based processing. There have been rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Lambda Rail (NLR) effort. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster, and streaming model is gaining popularity. At the same time, XML has been widely adapted in web-based [37],
distributed [14], and grid computing [38]. The second development is the popularity of virtual XML, where XML is used as a logical view to low-level data formats, such as flat-file Bioinformatics data [70] or network data¹.

To query and process (virtual) XML data streams, XQuery designed by W3C [13] can be an ideal language, because of its declarative nature and powerful features. In previous chapters we have presented a framework to process XQuery over flat-file based dataset, where XML is used as an logical interface and the target environment is cluster of machines. In this chapter, we will describe how to extend GNL as a high-level representation for processing queries over native XML datasets in a stream fashion.

The rest of the chapter is organized as follows. First, we give an overview of ongoing of related research efforts and briefly introduce our contribution in this area. The overall problem of processing queries over XML streams is described in Section 6.2. Our high-level analysis, including the stream data flow graph, horizontal and vertical fusion techniques, and the technique to determine if the query can be executed correctly on streaming data are presented in Section 6.3. Low level analysis and code generation techniques are presented in Section 6.4. We present our experimental evaluation in Section 6.5 and summarize our research contributions on this area in Section 6.6.

6.1.1 Overview of Ongoing Efforts

Though there are a number of ongoing research projects on XQuery evaluation, relatively few of them have focused on streaming data. Currently, there is a limited work on query evaluation on XML streams, and most of this handles only XPath fragments [68, 66, 27]. Because of the regularity of XPath expressions, automaton based approaches have

¹PADS-GALAX effort, see http://www.galaxquery.org/slides/xsym2004.pdf
been used to process XML streams. XPath is only a small subset of XQuery, and in comparison, provides limited functionality. Most of the queries supported by XPath involve only navigation and filtering of the XML stream.

Compared to XPath, XQuery is significantly more expressive, and therefore, more challenging to handle. Currently, some techniques have been proposed for processing XQuery queries over streaming data [58, 53]. Transducer networks have been used in XSM [58] to handle a small subset of XQuery, in which only join and node creation operations are allowed. Flux [53], on the other hand, uses static analysis for optimize buffer size. There are two important limitations in both these efforts. First, neither of them can handle aggregation functions, which we believe can be critical in specifying the type of analysis that is often done on streaming data. Second, neither of them have presented query transformations techniques to reduce the number of traversals, which again can be important for enabling a larger number of queries to be executed correctly on streaming data.

As compared to the existing work on supporting XPath/XQuery over data streams, we make the following contributions:

1) In many cases, direct translation of a XQuery query requires multiple passes on the data, whereas the query can be transformed to correctly execute with only a single pass. We present techniques for enabling such transformations. We model the dependencies in the query using a representation we refer to as the stream data flow graph. We apply a series of high-level transformations, including horizontal and vertical fusion. These techniques enable a larger number of queries to be evaluated correctly on streaming data, and efficiently on any large dataset. Furthermore, such transformations reduce the workload of a
query programmer, who otherwise must rewrite the query manually to execute correctly on streaming data.

2) Based on our stream data flow graph, we present a methodology to determine if a query can be evaluated correctly in a single pass. This enables us to avoid generating a query evaluation plan that is going to fail, and instead, a user can be given feedback sooner.

3) We propose a new technique to generate efficient streaming code, using our GNL representation described earlier. We also introduce a new optimization, which we refer to as the control-aware optimization, to reduce the volume of data that needs to be buffered.

We have evaluated our implementation using several XMark benchmarks and three other XQuery queries driven by real applications. Our experimental results show that as compared to Qizx/Open, Saxon, and Galax, our system: 1) is at least 25% faster on XMark queries with small datasets; 2) is significantly faster on XMark queries with larger datasets; 3) at least one order of magnitude faster on the queries driven by real applications, as unlike other systems, we can transform them to execute with a single pass; and 4) executes queries efficiently on large datasets when other systems often have memory overflows.

6.2 Preliminaries

This section describes our data and evaluation model. We introduce the notion of progressive blocking operators, and describe the overall problem.

6.2.1 Evaluation Model

We assume that the length of the incoming XML stream exceeds our capability of storing it. We only investigate the possibility of obtaining exact query results in a single pass.
Approximate processing of queries using a single pass on streaming data has been extensively studied by many researchers, and we do not consider this possibility here. We limit the number of input streams to be one. Also, we assume that duplicate-preserving is always used for XPath expressions in the query.

When an incoming tuple is available, it is fetched for evaluation and a series of internal computations are performed. As a result of this computation, an output tuple may be dispatched. A limited amount of memory is available for internal buffering, which is much smaller than the entire length of the data stream.

The internal computations can be viewed as a series of linked operators. Each operator receives input from its parent(s), performs an operation on the input, and sends the output tuples to its children. An operator could be a pipeline operator or a blocking operator, as described by Babu and Widom [9].

**Pipeline Operator:** A pipeline operator can immediately dispatch the output tuple after processing one input tuple. In our system, assume that the input of the operator $f$ is

$$\text{Input}(f) = [x_1, x_2, \ldots, x_n]$$

and the output stream is

$$\text{Output}(f) = [y_1, y_2, \ldots, y_k]$$

A pipeline operator $f$ has the property:

$$y_i = g(x_{h(i)}, b)$$

where, $h$ is monotonically increasing and $b$ is a bounded size buffered synopsis of $x_1, x_2, \ldots, x_{h(i)-1}$. An example of a pipeline operator is the selection operation.
**Blocking Operator:** A blocking operator must receive all its input before generating the output. Using the above notation for input and output, for a blocking operator we have

\[ [y_1, y_2, \ldots, y_k] = g(x_1, x_2, \ldots, x_n) \]

An example of a blocking operator is the sort operation.

For our analysis, we introduce a special type of a blocking operator, which we refer to as the **progressive blocking operator**. This is based on the observation that not all blocking operators require buffering of the entire input before generating the output. If the following two conditions hold true, a blocking operator is a progressive blocking operator.

\[ |\text{Output}(f)| \ll |\text{Input}(f)| \quad (6.1) \]

\[ g(x_1, x_2, \ldots, x_n) = g_1(g(x_1, x_2, \ldots, x_{n-1}), x_n) \quad (6.2) \]

In such cases, the operator can be evaluated as follows. At each step, we only need to buffer the temporary results and can discard the input. This is because the Equation 6.2 ensures that the input is no longer necessary for the later computations. Equation 6.1 ensures that temporary results can actually be buffered in our evaluation model. An example of such an operator is the count operation.

### 6.2.2 Problem Overview

The analysis we perform in this chapter is based on the following key observation. In a system with limited memory, a query cannot be evaluated using a single pass on the entire data stream to obtain an exact answer if the following conditions holds true:

- A blocking operator with unbounded input is involved in the query, or
A progressive blocking operator with unbounded input is involved and its output is used by another pipeline or progressive blocking operator.

The first condition is straightforward. Let us consider the second condition. When the final output of a progressive blocking operator $f_1$ is referred by another operator $f_2$, which is either a pipeline or a progressive blocking operator, $f_2$ must wait until the computation of $f_1$ finishes. This blocks the pipeline or progressive blocking computation $f_2$ defines. Queries that satisfy this propriety are referred to as *correlated aggregates* [40], which in most cases can only be evaluated approximately with a single pass.

The dependence between blocking operators and pipeline or progressive blocking operators that prevents a query from being evaluated in a single pass can either be a *control dependence* or a *data dependence*. Control dependence between operators is involved in *correlated sub-queries*, where the result of a sub-query is used as a predicate to filter the tuple selection. Many research efforts have focused on *de-correlating* such queries using various *unnesting* techniques, in the context of both relational [52, 64] and object-oriented databases [57, 21, 30].

Data dependence occurs when an operator computes a value that uses the result of a previous operator as an operand. The following query, referred to as the *Query 1*, is an example where data dependence between operators is involved. Here, `pixel` contains two elements, `x` and `y`.

**Query 1:**

```plaintext
let $b = \text{count}(\text{stream/pixel}[x>0])

for $i$ in stream/pixel
    return $i/x idvi $b
```

93
Much of our analysis focuses on such dependencies. One possible approach for such analysis could be the use of algebras. However, due to the expressive power and flexibility of XQuery, this approach is unlikely to model the interdependencies between operators. This is especially true for user-defined aggregations and recursive functions, for which a simple yet complete algebra has not yet been proposed.

In the rest of this chapter, we propose to use static analysis at the expression level to model the data flow and dependence information for XQuery. Static analysis will be used for guiding query transformation, as well as efficient code generation for query evaluation. Static analysis techniques have extensively been used in the programming language community for optimization of imperative languages [4] and quite recently, have been successfully used for analyzing and optimizing XQuery [53, 60].

6.3 High-level Analysis

This section describes the high-level analysis done in our system. Our goal is to correctly transform the query so that it can be processed in a single pass, when it is possible, and also to recognize when single pass analysis is not possible. Low-level analysis to facilitate code generation for XQuery with user-defined aggregates and recursive functions is discussed in the next Section. Initially, we give an overview of our overall framework.

6.3.1 Overview

Analysis based on relational algebra has recently been proposed to characterize the memory usage of SQL queries over continuous streams [7]. Since our focus is on XQuery, we do not use the algebra approach for the following two reasons. First, as we had stated earlier, developing an algebra to fully exploit the expressive power of XQuery is hard. For example, most existing XQuery engines handle this language one expression at a time,
which does not allow aggressive optimizations. Second, unlike SQL, data dependence is frequently introduced in an XQuery code through the use of binary expressions for computations. Such dependence relationships are traditionally well represented by dependence graphs in optimizing compilers. Specifically, in the next subsection, we introduce a representation called the *stream data flow graph*.

As we had discussed in the previous section, there are two cases in which a query cannot be processed in a single pass. The first one involves a blocking operator with unbounded input. The second one involves a progressive blocking operator with unbounded input whose output is used by another pipeline or progressive blocking operator. The first case is simple to detect. Therefore, for our analysis in this section, we assume that we only have pipelined or progressive blocking operators in our query, i.e., we do not have a blocking operator which cannot be evaluated progressively.

Figure 6.1 shows the key phases in our system. First, we construct the stream data flow graph representing the data dependence information for the query. Then, we apply a series of high-level transformations to prune and merge the stream data flow graph. Such techniques not only simplify the later analysis, but most importantly, they can rewrite some queries to enable single pass processing. After pruning the graph, a *single pass analysis* algorithm will be applied to the resulting data flow graph to check if single pass evaluation is possible. If the answer is no, further processing will not be performed. Otherwise, we apply low-level transformations and our code generation algorithm, and efficient single pass execution code is generated.
6.3.2 Stream Data Flow Graph

We introduce the stream data flow graph to represent dependence information and enable high-level analysis and optimizations on XQuery.

**Definition 6** Given any pair of variables $v_1, v_2$, if the definition of $v_2$ uses the value of $v_1$, or if the value of $v_1$ impacts whether or not $v_2$ is evaluated, $v_2$ is considered dependent on $v_1$.

**Definition 7** A stream data flow graph is a directed graph in which each node represents a variable in the original query and the directed edges $e = (v_1, v_2)$ implies that $v_2$ is dependent on $v_1$.

We introduce nodes for the variables defined in the original query, such as those defined in Let and For clauses, as well as for output value of a function or an XPath expression that
is not explicitly defined in the original query. We distinguish between nodes that represent a sequence, and nodes which represent atomic values. This is because dependence relationships between sequences and atomic values are of particular importance. We represent nodes of sequence type (of unbounded length) with rectangles and nodes of atomic type (or sequences of bounded length) with circles.

The stream data flow graph for the Query 1 described in the previous section is shown in Figure 6.2. $S1$ is the implicit variable that represents the XPath expression \texttt{stream/pixel[x>0]}. Similarly, $S2$ is used to represent \texttt{stream/pixel}. The output of the aggregate function \texttt{count()} is represented by $v1$. Here $i$ in the \texttt{for} clause is treated as an atom variable to represent each item in the binding sequence.

**Lemma 3** The stream data flow graph for a valid XQuery query is acyclic.

**Proof:** The proof directly follows from the single assignment feature of XQuery [13]. Assume there is a cycle, then one of the following conditions must hold true: 1) a variable $v$ is defined more than once, or 2) a variable $v$ is referred to without definition.

Neither of the above are allowed in a valid XQuery query. ●

We distinguish between two types of dependence relationship among the nodes.

**Definition 8** Given two variables $v_1,v_2$, we say that $v_2$ is aggregate dependent on $v_1$ if: 1) $v_2$ is dependent on $v_1$, and 2) $v_1$ is a sequence variable, $v_2$ is an atomic variable, and moreover, $v_2$ is not used as the iterator variable for any \texttt{for} expression. In such a case, we denote $v_1 \succ v_2$.

Aggregate dependence typically exists between a progressive blocking operator and its output.
Definition 9: Given two variables $v_1, v_2$, we say that $v_2$ is flow dependent on $v_1$ if: 1) $v_2$ is dependent on $v_1$, and 2) $v_2$ is not aggregate dependent on $v_1$. In such a case, we denote $v_1 \rightarrow v_2$.

Let us reconsider the Figure 6.2. We have used dashed arrows to represent aggregate dependence, and solid arrows for flow dependence.

6.3.3 High-level Transformations

Let us consider a stream data flow graph. If this graph contains multiple rectangle nodes, the corresponding query cannot be evaluated in a single pass, if we strictly follow the original syntax and do not allow pipelined execution. This is because each rectangle node represents a sequence that may have an infinite length, which cannot be buffered in the main memory.

However, by applying our query transformation and graph pruning techniques, including horizontal and vertical fusion, many queries can still be evaluated in a single pass.
Graph Pruning with Horizontal Fusion

Consider a query that involves multiple traversals of a data stream. If these traversals share a common prefix in their corresponding XPath expressions, we can merge these traversals into one, and could enable processing in a single pass.

As an example, we consider the following query:

Query 2:

```
let $b = count(stream/pixel[x>0])
return sum(stream/pixel/y) idvi $b
```

The original query involves two traversals of the entire stream, and cannot be processed directly without buffering the stream. However, since the two XPath expressions share a common prefix `stream/pixel`, the computation of `count` and `sum` can be carried out in a single traversal of `stream/pixel`.

To fuse multiple traversals together, we first generate a new node representing their common prefix. Then, for each original sequence node representing the traversal, the label
will be changed to the subexpression obtained by removing the common prefix. A new edge will be added linking this node to the new node. If the subexpression obtained after removing the common prefix is empty, the corresponding node is deleted, and its children have an edge from the parent node.

The stream data flow graph for the Query 2 after horizontal fusion is shown in Figure 6.3. In this example, a new sequence node $S0$ is generated corresponding to the common prefix /stream/pixel. The label of the two original sequence node are changed to the remaining XPath expressions, which are $[x > 0]$ and $/y$, respectively. Each new node is linked to $S0$.

Sometimes horizontal fusion in a query may lead to incorrect results, because of interdependence among the traversal of sequences. As an example, consider the Query 1. The data flow graph after horizontal fusion is shown in Figure 6.4. When we combine the traversal to compute count and the final output together, in each iteration, the output will be computed using partial result of $b$, which is not correct. In our method, we just apply horizontal fusion irrespective of such inter-dependence. Later, during single pass analysis, such dependence will be detected and the query will be eliminated from further processing.

For nested queries with pre-defined iteration space, which are common in many scientific data processing applications, horizontal fusion can be applied after unrolling. Unrolling is a commonly used technique in traditional compilers. Consider the following simple query:

```xml
unordered(
    for $i$ in (white, red, yellow)
    let $b$: =//stream/pixel[color=$i]
    return count($b))
```

By unrolling the first `for` expression, we can generate the following intermediate query:

```xml
unordered(
    let $b1$: =//stream/pixel[color=white])
```
Figure 6.4: Horizontal and Vertical Fusion for Query 1

let $b2: =//stream/pixel[color=red]
let $b3: =//stream/pixel[color=yellow]
return count($b1), count($b2), count($b3)

Since the XPath expressions generated after unrolling share the same common prefix, horizontal fusion can be applied to all the sequence node corresponding to the different iterations.

**Graph Pruning with Vertical Fusion**

The stream data flow graph can be further pruned using a technique called vertical fusion. Vertical fusion exploits the benefits of the pipelined processing, which can remove unnecessary buffering and simplify the data flow graph.

Consider the following example.

Query 3:

let $b: = for $i in stream/pixel[x>0]

return $i
for $j in $b/y
return $j$

where $j = \text{count}(\$b)$

In this query, $b$ contains all tuples from the original stream with a positive value of the $x$ coordinate. In a pipelined fashion, we can further process each tuple in $b$ as soon as it is available without buffering the entire sequence of $b$, which is required for unbounded streams.

As described in 6.2.2, we only need to check dependence between a progressive blocking operator and a pipeline operator, while dependence among pipeline operators can be ignored. In vertical fusion, we try to merge multiple pipeline operations on each traversal path into a single cluster in the stream data flow graph. The cluster obtained after fusion is referred to as a super-node. A super-node is represented in the data flow graph with a dashed box enclosing all the merged nodes. By doing so, the pipeline operation and the progressive blocking operations can be separated, and the number of isolated nodes in the data flow graph is reduced. This significantly simplifies later analysis on their dependence relationships.

Our algorithm does a top-down traversal from each root node, following only the flow dependence edges. For each node visited during the traversal, it will be fused with the current super-node, if it is not already in another super-node. Note that not all sequence nodes can be merged by vertical fusion. If a sequence $B$ is flow dependent on both the sequence node $A$ and the sequence node $C$, which normally occurs when $B$ is the result of a join between $A$ and $C$, we will merge $B$ with either $A$ or $C$, but not both of them.

The details of the algorithm are shown in Figure 6.6. $R$ is the set of the nodes in the graph that do not have an incoming edge. $N$ denotes the set of nodes that have been inserted in any super-node. $\mathcal{N}$ denotes the compliment of $N$, i.e., the nodes in the graph that are not
in the set $\bar{N}$. The algorithm picks a sequence node $s_i$. It follows the flow dependence edges (denoted as $\rightarrow$) to find nodes that can be fused into a super-node with $s_i$. These nodes are put in the set $M$. Any node that has already been fused into a super-node, (i.e., is not in $\bar{N}$) is not inserted in $M$.

The data flow graph for the Query 1 after vertical fusion is shown in Figure 6.4(b). The data flow graph for the Query 3 after vertical fusion is shown in Figure 6.5 (b).

Vertical fusion simplifies the stream data flow graph for further analysis and optimization. After vertical fusion, most of the queries that can be processed in a single pass will have only one rectangle node in their data flow graph.

### 6.3.4 Single Pass Analysis

After horizontal and vertical fusion, analyzing whether a query can be evaluated in a single pass becomes simpler. For our discussion here, we treat all nodes in a super-node after vertical fusion as a single sequence node. With this, any stream data flow graph that contains more than one sequence node cannot be evaluated in a single pass. This is because
Vertical Fusion
Input: 1) data flow graph \( G = (V, E) \)
2) root set \( R \)

\[ N = \emptyset \]

\[ \text{foreach node } s_i \in R \{
    \text{if } s_i \text{ is a sequence node}
    M = \{s_i\}
    \text{do }
    N = N \cup M
    \text{Let } T = \{v | \exists x, (x \in M) \land (x \rightarrow v)\}
    M = M \cup (T \cap \bar{N})
    \text{until } (T \cap \bar{N} = \emptyset)
    \text{fuse } M \text{ into super-node}
    \}
\]

Figure 6.6: Algorithm for Vertical Fusion

each such node represents one traversal of a sequence of length \( \theta(\mathcal{N}) \). If two sequence nodes are not fused with vertical fusion to apply pipelined execution, two traversals must be used. Thus, we have the following theorem.

**Theorem 1** If a query \( Q \) with dependence graph \( G = (V, E) \) contains more than one sequence node after vertical fusion, \( Q \) may not be evaluated correctly in a single pass.

However, for queries whose stream data flow graph contains only one sequence node, a single pass evaluation may still not be possible. Two types of dependence relationship may prevent the query from being executed in a single pass. Examples of these two cases are shown in Figure 6.7.

**Theorem 2** Let \( S \) be the set of atomic nodes that are aggregate dependent on any sequence node in a stream data flow graph \( G \). For any given two elements \( s_1 \in S \) and \( s_2 \in S \), if
there is a path between $s_1$ and $s_2$, the query may not be evaluated correctly in a single pass.

**Proof:** For each $s_i \in S$, $s_i$ can only be computed after the sequence $V_i$ it depends on is fully scanned. Assume there is a path from $s_1$ to $s_2$, then the value of $s_2$ must be computed using $s_1$. Thus, the scan of $V_2$ must follow the scan of $V_2$. This implies that the query cannot be processed with a single pass. \(\blacksquare\)

In addition to the condition associated with the Theorem 2, there is another condition we need to check for.

**Lemma 4** If a stream data flow graph $G$ contains a cycle, it is formed after horizontal or vertical fusion.

**Proof:** From lemma 3 there is no cycle in the original stream data flow graph. Therefore, the cycle must be formed by either horizontal fusion or vertical fusion. \(\blacksquare\)

**Theorem 3** In there is a cycle in a stream data flow graph $G$, the corresponding query may not be evaluated correctly using a single pass.
Proof: From the lemma above, the cycle is formed after horizontal or vertical fusion. If the cycle is formed right after horizontal fusion of \( s_1 \) and \( s_2 \), there must be a path between \( s_1 \) and \( s_2 \), which implies dependence of \( s_2 \) on \( s_1 \). In this case, horizontal fusion will generate incorrect results, and single pass evaluation is impossible.

If the cycle is formed after vertical fusion, a super-node must be involved in the cycle. Assume the cycle is \( v_1, v_2, \ldots, v_k, v_1 \), and \( v_i \) is a super-node. Then, it is true that \( v_{i+1} \) is aggregate dependent on the node \( v_i \), otherwise, \( v_{i+1} \) will be fused with \( v_i \) during vertical fusion. Thus, the value of \( v_{i+1} \) can only be valid after the pipelined execution of \( v_i \) is completed. Because a cycle exists, the pipelined execution of \( v_i \) also requires the value of \( v_{i+1} \). As a result, pipelined execution of \( v_i \) is not possible, and the query cannot be evaluated in a single pass. ●

After vertical fusion, stream data flow graphs for both Query 1 and Query 3 contain cycles, and therefore, these queries cannot be executed with a single pass.

If the conditions corresponding to any of the above three theorems hold true for a query, we cannot further process the query using a single pass and ensure correct results. If the original graph has \( n \) vertex, the conditions corresponding to Theorems 1, 2, and 3 can be applied in \( O(n) \), \( O(n^2) \), and \( O(n) \) time, respectively.

The next theorem shows that if the conditions corresponding to the Theorems 1, 2, and 3 all hold false, the query can be processed correctly in a single pass.

**Theorem 4** If the results of a progressive blocking operator with an unbounded input are referred to by a pipeline operator or a progressive blocking operator with unbounded input, then for the stream data flow graph \( G = (V, E) \), at least one of the following three conditions holds true:

1. There are multiple sequence nodes.
2. There is a cycle involved.

3. \( \exists \text{ sequence node } s \in V, \exists \text{ atomic nodes } a_1 \in V, a_2 \in V, a_1 \text{ and } a_2 \text{ are aggregate dependent on } s, \text{ and there is a path from } a_1 \text{ to } a_2. \)

\textbf{Proof:} Assume that the progressive blocking operation is represented in \( G \) with a sequence node \( s \) and an atomic node \( a \), such that \( a \) is aggregate dependent on \( s \). Assume that there is no other sequence node in \( G \), otherwise the first condition holds true.

If the value of \( a \) is referred to by another progressive blocking operator to compute \( a' \), since \( s \) is the only sequence node in \( V \), \( a' \) must be aggregate dependent on \( s \). Because \( a' \) uses the value of \( a \), there must be a path \( a, v_1, \ldots, v_k, a', a \rightarrow v_1, \ldots, v_k \rightarrow a' \). Therefore, the third condition holds true.

Now, suppose the value of \( a \) is referred by a pipeline operator. Then, there must be a super-node in the graph, and there is a path \( a, v_1, \ldots, v_k, s \), such that \( a \rightarrow v_1, \ldots, v_k \rightarrow s \). Since \( a \) is aggregate dependent on \( s \), there will be a cycle \( a, v_1, \ldots, v_k, s, a \) in the graph. Then, the second condition holds true. 

Finally, it should be noted that as with all static analysis, our analysis is conservative in nature. There could be cases where a query can be processed in a single pass, but our analysis will determine that it cannot be. We consider the following example:

\begin{verbatim}
let $p: = \text{stream/pixel/x}
for $i in $p
    where $i <= \text{max($p$)}
    return $i
\end{verbatim}

This query has a \textit{redundant predicate} [7]. Though the predicate always returns true and does not impact the results from the query, it introduces a cycle in our graph, and disallows
processing with a single pass. Our analysis can be extended to recognize and remove such redundant predicates, but we do not expect them to arise frequently in real situations.

6.4 Low-level Analysis and Code Generation

This section focuses on the analysis and optimizations we perform for generating efficient streaming code for XQuery codes. As compared to the existing work on evaluating XQuery on streaming data, we make two significant contributions in this section. First, we show how we can generate process XQuery with user-defined aggregates. Second, we present a new optimization called the control-aware optimization, which can improve the efficiency of streaming code.

For achieving efficiency and handling a general class of XQuery codes, we generate executable for a query directly, instead of decomposing the query at the operator level and interpreting the query plan. This is similar in nature to the optimized codes that are generated by a compiler according to a specified underlying architecture. In comparison, interpreted codes generally suffer in efficiency, as has been shown for many languages, for example, Matlab [5]. Furthermore, operators cannot model some features of XQuery effectively, such as recursive functions.

6.4.1 GNL Representation

As described in the previous section, we generate a stream data flow graph from a given XQuery code. However, the stream data flow graph only represents a high-level view of the dependencies among variables and expressions, while details of the processing involved are not modeled. To facilitate generation of streaming code, we will use GNL as the intermediate representation for code generation, which helps exploit the imperative
nature of XML parsers such as SAX. Though our current implementation has been carried out on top of SAX, our code generation techniques are more general.

By definition, the tuple stream that a GNL operates on is specified by its path expression, and the aggregation operations are specified by the statement sequence in its body. With such a syntax structure, code generation for Java functions triggered by SAX events becomes easier. Furthermore, as we will describe later, low-level optimizations techniques on rewriting recursive functions and user-defined aggregates are also facilitated by the GNL representation.

**GNL Formation**

After the single pass analysis, only queries whose stream data flow graph have only one super-node or sequence node are left for further processing. If the query has only one sequence node, it is easy to map the query to the GNL representation. Since the sequence node represents a traversal on its path expression, we can directly map a sequence node to a GNL with an empty loop body $S$. This GNL is denoted as $G_1$. In addition, we introduce one GNL corresponding to the main query, which is denoted as $G_0$. The GNL $G_1$, as well as all atomic nodes in the stream data flow graph, are inserted as statements in the loop body of $G_0$. The order of these statements in loop body of $G_0$ needs to be consistent with their order in the original query.

If there is a super-node in the data flow graph, the GNL formation process is more complex. We create a GNL $G_0$ corresponding to the main query. We also create a GNL corresponding to each sequence node in the super-node that has an associated XPath expression. The GNL corresponding to the root sequence node in the super-node is inserted as a statement in the loop body of $G_0$. Consider any other node $n$ in the super-node. We

[^2]: http://www.saxproject.org
for $i_1$, stream/pixel, --
[ for $i_2$, /x, /x > 0
  [ for $i_3$, /y, --
    $v_1 = \text{count()}
    b = v_1
    v_2 = \text{sum()}
    \text{return } b \div v_2
  ]
]

(b) GNL after Aggregation Rewrite

Figure 6.8: Example of GNLs

find the closest ancestor of this node that has a GNL associated with it, and denote it as $CA$. If the node $n$ has a GNL associated with it (because it is a sequence node with an XPath expression), this GNL is inserted as a statement in the loop body of the GNL for $CA$. Otherwise, the statement corresponding to the node $n$ is inserted in the loop body of the GNL for $CA$. Finally, consider any atomic node that is not in the super-node. The statement corresponding to this node is inserted in the loop body of $G_0$. Whenever multiple statements are inserted in a loop body, their sequence must be consistent with their sequence in the original query.

GNL for the Query 3 is shown in Figure 6.8(a).

**Aggregation Rewrite for GNLs**

After the GNL representation is generated from the stream data flow graph, aggregation functions, including those defined by users, are typically placed outside the root GNL. This is because the formation of GNL does not follow any edge that represents aggregate dependence. Therefore, we need to recognize such aggregation functions, rewrite them...
into operations that can be applied a tuple at a time, and move the aggregation into the corresponding GNL. Here, we assume that any aggregation operation on a tuple stream is performed through a function, which could be user-defined or internal.

An aggregate function can be easily recognized by finding all atomic nodes that are aggregate dependent on a sequence node or a super-node. For each such node \( v \), we move the corresponding function call into the GNL by using function inlining. Internal functions, such as \( \text{sum}, \text{count}, \text{and average} \), can be easily rewritten in an iterative fashion. For example, we can rewrite \( \text{sum}() \) as
\[
\text{tmp} = \text{tmp} + v,
\]
where \( \text{tmp} \) is a temporary variable and \( v \) is the tuple. For a user-defined function, including recursive functions, we apply the static analysis technique described in Chapter 4 to extract an associative operation from the definition of the function. The basic idea is to examine the syntax tree from leaves and apply tree pattern matching to retrieve the desired sub-tree. Again, our algorithm can only deal with linear recursive functions.

The GNL of Query 3 after aggregation rewriting is shown in Figure 6.8(b).

### 6.4.2 Code Generation

We now discuss details of evaluation for a query on streaming XML. To achieve better performance, we generate executable code for a given query, instead of using a query evaluation engine to interpret the query at runtime. Specifically, we generate Java binary code using the XERCES SAX XML Parser, which is executed using the JDK 1.4 runtime system.

Similar to Peng and Chawathe [68], our processing assumes that an incoming data tuple is one of the following three types: 1) \( \text{startElement}(n, attr) \), which is the start event for the
node $n$ with attribute list $\text{attr}$, 2) $\text{character}(n)$, which is the content of the node $n$, and 3) $\text{endElement}(n)$, which is the end event for the node $n$.

**From GNL to SAX Event Handling**

The GNL generated from a query serves as a convenient intermediate representation for code generation. GNL uses a nested loop structure, which is commonly supported in imperative languages such as Java. Since the SAX parser internally supports streaming traversal, and generates a series of streaming events according to the document order, the explicit traversal defined in a GNL does not appear in the final code.

Specifically, we use the following strategy to evaluate a GNL:

- For a given GNL $E$ with $E_p = /x/y/z$, when the event $\text{endElement}(z)$ is triggered, the body of $E$ is executed once.
- For nested GNLs $E_1$ and $E_2$, with $E_2$ nested inside $E_1$, the processing for $E_2$ is always performed before the processing for $E_1$.

The code generated for evaluation of Query 2 is shown in Figure 6.9.

Though our goal is to process queries on the fly, certain XML elements may need to be buffered. For example, a node may be issued for output after a condition involving its children nodes is evaluated. Clearly, because buffering requires memory, we want to buffer as few elements as possible. We use a filtering and projection technique similar to the one described in [53]. However, our technique is simpler because we do not process blocking operators like $\text{join}$ and $\text{sort}$. Specifically, we buffer a node with its entire subtree if it is used as output. Also, we buffer any leaf node $v$ whose value is referred to in the query. In such a case, if $v$ is not in the subtree of a node used as output, the buffer of $v$ will be
immediately dispatched when reference of $v$ is finished. Using the GNL representation, we check the location path $E_p$ and filter $E_c$ of each GNL, and only mark a node for buffering if it is referred to in the body of that GNL.

The details of buffering for the Query 3 are shown in Figure 6.9. In this example, no node is used as output, and buffering of /x and /y is for using their values only. Therefore, these nodes are dispatched during the handling of endElement event.

```java
foreach startElement ($e_i$) {
    switch ($e_i$.node)
    x: buffer.add(x)
    y: buffer.add(y)
}

foreach endElement ($e_i$) {
    switch ($e_i$.node)
    x: if (buffer.dispatch(x) > 0)
        $v_1 = v_1 + 1$
    y: $v_2 = v_2 + buffer.dispatch(y)$
    root: { $b = v_1$
        return $b / v_2$
    }
}
```

Figure 6.9: Evaluation Code for Query 3

**Control-Aware Optimization**

Many queries involves checking the value or attribute of a node for a boolean condition before further processing its subtree or siblings. For such queries, we can improve the performance by not buffering the subtree, as soon as the condition has been evaluated to be false. To achieve this, we introduce a technique we refer to as control-aware optimization.
For example, consider the following query:

Query 4

for $i$ in stream/pixel[x>0])
    return $i$

We can generate code as shown in Figure 6.10, which will skip the processing of all sibling elements after $x$ turns out to be non-positive.

```java
flag_x = false;
foreach startElement ($e_i$) {
    switch( $e_i$.node)
        pixel: buffer.add(pixel)
        x: buffer.add(pixel.x)
        y: if (flag_x)
            buffer.add(pixel.y)
}

foreach endElement ($e_i$) {
    switch( $e_i$.node)
        x: if (buffer.get(pixel.x)<0)
            flag_x = true;
        pixel: if(flag_x)
            return buffer.dispatch(pixel);
}
```

Figure 6.10: Evaluation Code for Query 4

To perform such an optimization, we introduce the following definitions.

**Definition 10** For a boolean expression $b$, we define the Trigger Set, $T(b)$, to be all nodes whose names or attributes are referred in $b$.
Definition 11 Consider any boolean predicate $b$. Let $E$ be the set of expressions in the true branch of $b$, i.e., the expressions that are evaluated if $b$ is true. Then, the Dependent Set, $D(b)$, is the set of all node names that are referred in any expression $e \in E$.

Definition 12 Consider any boolean predicate $b$ and its dependent set, $D(b)$. The Dominated Set, $DD(b)$, is the set of nodes in $D(b)$ whose only references are in the true branch of $b$.

For performing the control-aware optimization, in addition to computing the trigger and dominated sets, we also assign an order number to any element defined in the XML Schema. For an element $n$, the order number is denoted as $ON(n)$. For two element names $n_1$ and $n_2$ defined in a XML schema, $ON(n_1) < ON(n_2)$ if the order of $n_1$ in the schema precedes that of $n_2$.

Theorem 5 Let $b$ be a boolean expression. Let $N$ be the maximum of the order number of any element in the trigger set $T(b)$. Buffering of an element $n_i$ is unnecessary if $n_i \in DD(b)$, $ON(n_i) > N$, and $b$ is evaluated to be false.

The reasoning for this is as follows. When an element with the order $N$ is received, we have all elements in the trigger set of $b$, and therefore, $b$ can now be evaluated. Since $ON(n_i) > N$, $n_i$ is still in the stream at this point. Since $n_i \in DD(b)$, $n_i$ is only needed when $b$ is true.

To perform control-aware optimization, we traverse the GNL and find all conditional expressions, including filter for XPath expressions and if and case-of expressions. We determine their trigger and dominated sets. Next, we check the XML schema and find all elements in each dominated set that satisfy the Theorem 5. In generating code for buffering
such elements, a flag is inserted to check the runtime value of the boolean expression that dominates the element.

6.5 Experimental Results

To evaluate our implementation of the framework and the techniques presented in this chapter, we conducted a series of experiments. We compared our implementation with other well known XQuery processors which are publically available. Specifically, we use Galax (Version 0.3.1) [32], Saxon (Version 8.0) [49] and Qizx/Open (Version 0.4.1) [1]. All these query processors are implemented using a SAX Parser, which we believe makes the comparison reasonable. Note that our transformations can enable many queries to be executed with a single pass, whereas other systems may require multiple passes for them. Thus, an advantage of our framework is that we may allow execution with a limited memory, or on unbounded streams, while the other systems may simply fail to execute the same query in such scenarios. For our comparison here, we only focus on execution times, and not the memory requirements or the ability of a system to process query on streaming data.

We used two sets of queries for our experiments. The first set comprised the queries 1, 5, 6, 7, and 20 from the XMark benchmark set [73]. These five queries were chosen because each of them could be processed in a single pass either directly, or after our transformations. We use datasets of different sizes, which were generated by the XMark data generator using factors 0.01, 0.05, 0.25, 1, and 2, respectively. The second set comprised three real applications which involve streaming data. Satellite data processing and Virtual Microscope have been described in Chapter 2, whose XPath expression are modified here to process streaming data. Frequent element counting is a well known data mining problem, here we use the one-pass algorithm by Karp et al. to find a superset of frequent items
### Evaluation Using XMark Benchmarks (All Execution Time in Seconds)

<table>
<thead>
<tr>
<th>Size</th>
<th>Ours</th>
<th>Qizx</th>
<th>Saxon</th>
<th>Galax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.16M</td>
<td>0.76</td>
<td>1.03</td>
<td>2.46</td>
<td>4.65</td>
</tr>
<tr>
<td>5.75M</td>
<td>2.26</td>
<td>3.2</td>
<td>5.57</td>
<td>24.59</td>
</tr>
<tr>
<td>30M</td>
<td>9.98</td>
<td>11.23</td>
<td>MO</td>
<td>173.85</td>
</tr>
<tr>
<td>120M</td>
<td>13.97</td>
<td>MO</td>
<td>MO</td>
<td>*</td>
</tr>
<tr>
<td>240M</td>
<td>27.59</td>
<td>MO</td>
<td>MO</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>Ours</th>
<th>Qizx</th>
<th>Saxon</th>
<th>Galax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.16M</td>
<td>0.74</td>
<td>1.09</td>
<td>2.46</td>
<td>4.93</td>
</tr>
<tr>
<td>5.75M</td>
<td>2.26</td>
<td>3.21</td>
<td>5.57</td>
<td>25.26</td>
</tr>
<tr>
<td>30M</td>
<td>9.94</td>
<td>13.68</td>
<td>MO</td>
<td>213.64</td>
</tr>
<tr>
<td>120M</td>
<td>13.87</td>
<td>MO</td>
<td>MO</td>
<td>*</td>
</tr>
<tr>
<td>240M</td>
<td>27.81</td>
<td>MO</td>
<td>MO</td>
<td>*</td>
</tr>
</tbody>
</table>

### Satellite Processing

<table>
<thead>
<tr>
<th>Size</th>
<th>Ours</th>
<th>Opt</th>
<th>Qizx</th>
<th>Saxon</th>
<th>Galax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05M</td>
<td>0.28</td>
<td>0.27</td>
<td>5.88</td>
<td>3.08</td>
<td>72.03</td>
</tr>
<tr>
<td>0.10M</td>
<td>0.33</td>
<td>0.33</td>
<td>20.48</td>
<td>4.45</td>
<td>136.7</td>
</tr>
<tr>
<td>0.66M</td>
<td>0.48</td>
<td>0.47</td>
<td>945.5</td>
<td>444.4</td>
<td></td>
</tr>
<tr>
<td>2.70M</td>
<td>1.60</td>
<td>1.52</td>
<td>4099.7</td>
<td>62.89</td>
<td></td>
</tr>
<tr>
<td>10.6M</td>
<td>3.47</td>
<td>2.98</td>
<td>*</td>
<td>MO</td>
<td>MO</td>
</tr>
<tr>
<td>100M</td>
<td>28.31</td>
<td>23.16</td>
<td>MO</td>
<td>MO</td>
<td>MO</td>
</tr>
</tbody>
</table>

### Virtual Microscope

<table>
<thead>
<tr>
<th>Size</th>
<th>Ours</th>
<th>Opt</th>
<th>Qizx</th>
<th>Saxon</th>
<th>Galax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05M</td>
<td>0.28</td>
<td>0.27</td>
<td>47.51</td>
<td>2.01</td>
<td>18.97</td>
</tr>
<tr>
<td>0.10M</td>
<td>0.32</td>
<td>0.31</td>
<td>*</td>
<td>24.7</td>
<td>38.98</td>
</tr>
<tr>
<td>0.66M</td>
<td>0.44</td>
<td>0.44</td>
<td>*</td>
<td>7.66</td>
<td>300.18</td>
</tr>
<tr>
<td>2.70M</td>
<td>1.54</td>
<td>1.52</td>
<td>*</td>
<td>24.56</td>
<td>MO</td>
</tr>
<tr>
<td>10.6M</td>
<td>3.29</td>
<td>2.89</td>
<td>*</td>
<td>MO</td>
<td>MO</td>
</tr>
<tr>
<td>100M</td>
<td>27.88</td>
<td>24.28</td>
<td>*</td>
<td>MO</td>
<td>MO</td>
</tr>
</tbody>
</table>

### Karp Frequent Item

<table>
<thead>
<tr>
<th>Size</th>
<th>Ours</th>
<th>Opt</th>
<th>Qizx</th>
<th>Saxon</th>
<th>Galax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05M</td>
<td>0.26</td>
<td>*</td>
<td>*</td>
<td>4.71</td>
<td>25.09</td>
</tr>
<tr>
<td>0.10M</td>
<td>0.32</td>
<td>*</td>
<td>10.66</td>
<td>122.63</td>
<td></td>
</tr>
<tr>
<td>0.66M</td>
<td>0.61</td>
<td>*</td>
<td>554.07</td>
<td>MO</td>
<td></td>
</tr>
<tr>
<td>2.70M</td>
<td>1.80</td>
<td>*</td>
<td>8302.7</td>
<td>MO</td>
<td></td>
</tr>
<tr>
<td>10.6M</td>
<td>5.61</td>
<td>*</td>
<td>MO</td>
<td>MO</td>
<td></td>
</tr>
<tr>
<td>100M</td>
<td>29.41</td>
<td>*</td>
<td>MO</td>
<td>MO</td>
<td></td>
</tr>
</tbody>
</table>

### Performance on Real Streaming Applications (All Execution Times in Seconds)

*Unable to produce result after 24 hours*  
*Out of memory*  

Figure 6.11: Experiments Results for XMark Queries and Real Streaming Applications
in a data stream [48]. Each of these three applications uses recursive functions to perform aggregations. After applying our techniques and optimizations, including analysis of recursive functions, aggregate rewriting, and horizontal and vertical fusion, each of these could be processed correctly using only a single pass on the entire data stream. We generated synthetic datasets of varying sizes to evaluate performance on these applications.

The results of our experiments are shown in Figure 6.11. Our experiments were conducted on a 933 MHz Pentium III workstation, with 256 MB of RAM, and running Linux version 7.1, with JDK V1.4.0. In addition to comparing our basic framework with the three other systems, we also evaluated the impact of the control-aware optimization we had described earlier. Among the queries we have used, this optimization is only applicable to satellite data processing and virtual microscope. In the tables in Figure 6.11, ours denotes our basic framework, and opt denotes the version with control-aware optimization.

Because we use compiled Java byte code, the running time shown in the tables excludes the compilation time for other XQuery systems. All available options for fast execution and optimization are turned on for each system. Specifically, for Galax, we disable sorting and duplicate removal on Path expressions, and set the option of projection to be on.

The results show that we consistently outperform other systems. For XMark queries with small datasets, Qizx is often quite close, but our system is at least 25% faster. There are at least two reasons for this. First, our static analysis based technique produced operations only on elements that are referred in the query. Second, we generate imperative code directly, which is more efficient compared with interpreted execution used by other engines.

For XMark queries with larger datasets, either our system was significantly faster, or other systems had a memory overflow. It should be noted that none of the other systems have been designed to deal with large datasets and/or streaming data. They often require
in-memory processing. For example, Saxon builds a DOM tree after retrieving all data in memory, and therefore, cannot process large datasets or streaming data.

For the three real streaming applications, our implementation outperforms other systems by at least one order of magnitude, and often, much more. None of the other systems was able to execute these applications with only a single pass on the data, whereas, our techniques and transformations enabled such execution. Finally, as the dataset size increases, a 10 to 20% performance improvement is observed with the control-aware optimization on Satellite and Virtual Microscope.

6.6 Summary

This chapter has presented a new framework and a set of techniques for processing XQuery over streaming data. As compared to the existing work on supporting XPath/XQuery over data streams, we have made three contributions. First, we have developed a series of optimizations which transform XQuery queries so that they can be correctly executed with a single pass on the dataset. Second, we have presented a methodology for determining when an XQuery query, possibly after the transformations we introduce, can be correctly executed with only a single pass on the dataset. Finally, we have developed a code generation approach which can handle XQuery queries with user-defined aggregates, including recursive functions.

We have evaluated our implementation using several XMark benchmarks and three other XQuery queries driven by real applications. Our experimental results show that as compared to Qizx/Open, Saxon, and Galax, our system: 1) is at least 25% faster on XMark queries with small datasets, 2) is significantly faster on XMark queries with larger datasets, 3) at least one order of magnitude faster on the queries driven by real applications, as unlike
other systems, we can transform them to execute with a single pass, and 4) executes queries efficiently on large datasets when other systems often have memory overflows. In the next chapter, we will discuss how to extend our code generation technique based on SAX XML parsing to parallelize XQuery applications involving native XML datasets.
CHAPTER 7

PARALLELIZATION OF XQUERY OVER NATIVE XML DATASETS

7.1 Introduction

As XQuery is being used for processing large datasets, and/or for compute-intensive applications, efficiency of XQuery implementations is becoming an important issue. Distributing the dataset on a set of computers in a cluster, and parallelizing the processing of XQuery can be an approach for speeding up the execution. In previous chapters, we have presented techniques to parallelize XQuery over flat-file dataset, with XML functions as a logical view and ADR as the middleware support. In this chapter, we report on a parallel implementation of XQuery over native XML datasets using MPI and SAX XML parsing libraries. Parallelization of this class of XQuery queries involves a number of challenges, which can be summarized as follows:

- Developing efficient and correct data distribution techniques for semi-structured datasets.

- Translation of the GNL representation of XQuery processing to low-level code enabling both XML parsing and parallel communications.

In this chapter, we report our solutions towards the above problems. By implementing the techniques in a compiler and generating code based on a C++ SAX parser and the
Message Passing Interface (MPI), we are able to achieve efficient parallel execution on a cluster of machines.

The rest of the chapter is organized as follows. We describe our overall system in Section 7.2. The various analysis methods that our compiler uses for code generation are presented in Section 7.3. Finally, we report the parallel performance from our system in Section 7.4.

### 7.2 System Overview and Data Distribution

In this section, we briefly introduce the overall architecture of our system, and the data distribution method we use.

Our target environment is a *shared-nothing* architecture, i.e., cluster of machines, each with an attached disk. The overall processing involves a local reduction phase and a global combination phase. During local reduction phase, each node processes its local XML document in a streaming fashion based on a SAX XML parser, and local output elements are updated. In the global combination phase, local output elements from each node are broadcast to all other nodes using MPI calls, and are merged to generate the final output.

To achieve efficient parallel execution and local balance on shared-nothing architectures, an important issue is distributing the XML data evenly among all processing nodes. The method we use is based upon the technique Marian and Simeon used for optimizing the memory in a serial environment [60]. Assume there are $N$ computers in the cluster. Our distribution algorithm will generate $N$ XML documents from the original document, each of which contains (approximately) the same number of elements.

Specifically, we first statically analyze the actual paths that will be referred to in the queries we are likely to execute on this dataset. To perform the data distribution, we use
the set of actual paths to filter the original document. Any node that is not on these paths is not included in the final documents. We use the common prefix among the set of paths as the distribution criteria. For example, the common path prefix for the Satellite processing application is /data/pixel. We distribute the pixel nodes evenly into $N$ documents, while the parent nodes are duplicated in each document.

Clearly, our approach has some limitations. We require that multiple occurrence must be defined in the Schema for the nodes corresponding to the common path prefix. Also, similar to [60], we can only handle forward axis. Although preprocessing and distribution may introduce extra costs for query evaluation, we believe such preprocessing strategy may still be beneficial for many queries.

- By removing unnecessary nodes and distributing useful nodes evenly, the memory requirement used to process the original query can be decreased dramatically.
- For queries that are compute-bound, the time saved by parallel processing may be much more than the time spent on preprocessing.
- Finally, one distribution scheme may be shared for multiple similar queries. Thus, the cost of preprocessing and distribution may be amortized over the execution of multiple queries.

### 7.3 Parallelization of XQuery

In this section, we discuss parallel code generation of XQuery on shared-nothing or cluster architectures that processes XML data. Similar to the parallelization strategy described in Chapter 3 and 4, for our target class of applications, parallelization can be done by dividing data and computation evenly across the nodes, and having each node execute
local reduction operations on the data items it owns. After this, each node broadcasts the result of its local reduction to all other nodes, and a global reduction phase is invoked to combine the local results. Depending on the application, we may need to iterate over local and global reduction phases. However, because native XML datasets are involved here, we can no longer rely on ADR as a middleware for parallelization.

As described in the previous section, we process SAX events at the local reduction phase and invoke MPI calls for message passing at the global reduction phase. To support SAX parsing and invoking MPI calls, we must translate the original XQuery code to an imperative language such as C/C++. The GNL representation can still be used to facilitate generation of such code, as it helps exploit the imperative nature of XML parsers such as SAX. Though our current implementation has been carried out on top of SAX and MPI, our code generation techniques are more general. For examples, besides MPI, we believe GNL can facilitate code generation for using parallel interfaces and middleware such as OpenMP [23], and Global Arrays [65].

7.3.1 GNL Formation and Aggregation Rewrite

We use the result from previous static analysis for data distribution to translate the original query into GNL representation. Specifically, we use the common prefix of path expressions as $E_p$ of the GNL, since this common prefix specifies the sequence of tuples that are processed in parallel. References to other path expressions are merged into the body of the GNL. For aggregation functions, we move them into the GNL by using function inlining. Internal functions, such as sum, count, and average, can be easily rewritten in an iterative fashion. For example, we can rewrite sum() as $tmp = tmp + v$, where $tmp$ is a temporary variable and $v$ is the tuple. For a user-defined function, including recursive functions, we
apply a previously developed static analysis technique to extract an associative operation from the definition of the function [56]. The basic idea is to examine the syntax tree from leaves and apply tree pattern matching to retrieve the desired sub-tree. Our algorithm can only deal with linear recursive functions.

The GNL generated for Satellite Data Processing is shown in Figure 7.1(a). Here, we use /data/pixel as the path expression for GNL, and rewrite the recursive function into an iterative reduction operation.

### 7.3.2 Parallel Code Generation

Consider a XQuery query represented by GNLs. To parallelize structure, we need to generate parallel code for local processing on each node, and a global combination across the nodes. We now describe how we generate code for each of these phases.

**Local Reduction**

The data instances owned by a processor are read. A local reduction function specifies how, after processing one data instance, a local output element is updated. The result of this processing must be independent of the order in which data instances are processed on each processor.

The GNL generated from a query serves as a convenient intermediate representation for code generation. GNL uses a nested loop structure, which is commonly supported in imperative languages. Since the SAX parser internally supports streaming traversal, and generates a series of streaming events according to the document order, the explicit traversal defined in a GNL does not appear in the final code.

Specifically, we use the following strategy to evaluate a GNL:
for \((i, \text{stream/data/pixel}, -)\) \{ 
  if\((y \geq \text{miny})\text{and}(y \leq \text{maxy})\text{and}
  (x \geq \text{minx})\text{and}(x \leq \text{maxx}))
  \begin{align*}
    \text{NVDI} &= ((\text{band1} - \text{band0})/ \\
    \quad & (\text{band1} + \text{band0}) + 1) \times 512 \\
    \text{if}(\text{v}[x][y] \leq \text{NVDI})
    \text{v}[x][y] &= \text{NVDI}
  \end{align*}
\}

(a) GNLs after Aggregation Rewrite

flag\_xy = false;
foreach startElement \((e_i)\) \{ 
  switch\((e_i.\text{node})\) 
  \begin{align*}
    \text{pixel: init();} \\
    \text{x: buffer.add(pixel.x)} \\
    \text{y: buffer.add(pixel.y)} \\
    \text{band0: if (flag\_xy)} \\
    \quad \text{buffer.add(pixel.band0)} \\
    \text{band1: if (flag\_xy)} \\
    \quad \text{buffer.add(pixel.band1)}
  \end{align*}
\}

(b) Local Reductions Based on SAX Events

foreach endElement \((e_i)\) \{ 
  switch\((e_i.\text{node})\) 
  \begin{align*}
    \text{y: if (pixel.x} \in [\text{minx}, \text{maxx}) \text{and}
    \quad (\text{pixel.y} \in [\text{miny}, \text{maxy})]
    \quad \text{flag\_xy = true}
    \text{pixel: if(flag\_xy) } \{ 
    \quad \text{NVDI = ...;}
    \quad \text{if (v[pixel.x][pixel.y] <= NVDI)}
    \quad \quad \text{v[pixel.x][pixel.y] = NVDI}
\}
  \}
  \text{dispatch(pixel)}
\}

(c) Local Reductions (Contd.)

Figure 7.1: GNLs and Optimized Code for Satellite Application
• For a given GNL $E$ with $E_p = /x/y/z$, when the event `endElement(/z)` is triggered, the body of $E$ is executed once.

• For any path expressions $p$ nested inside GNLs $E$, the processing for $p$ is always performed before the processing for $E$.

The code generated for local reduction of Satellite Processing is shown in Figure 7.1. Though our goal is to process queries on the fly, certain XML elements may need to be buffered. For example, a node may be issued for output after a condition involving its children nodes is evaluated. Clearly, because buffering requires memory, we want to buffer as few elements as possible. We use a filtering and projection technique similar to the one described in [53]. However, our technique is simpler because we do not process blocking operators like `join` and `sort`. Specifically, we buffer a node with its entire subtree if it is used as output. Also, we buffer any leaf node $v$ whose value is referred to in the query. In such a case, if $v$ is not in the subtree of a node used as output, the buffer of $v$ is immediately dispatched when reference of $v$ is finished. Using the GNL representation, we check the location path $E_p$ and filter $E_c$ of each GNL, and only mark a node for buffering if it is referred to in the body of that GNL.

**Global Combination**

The local output elements on all processors are combined using a global reduction function. The code generation for the local reduction module was relatively simple, since the operations to be performed locally are explicitly defined within the body of a GNL. Although the global combination function performs the same associative and commutative operation that is defined in a GNL body, we cannot use the body of a GNL simply as the global combination function. This is because the input to a global combination function is
copies of output from other nodes, and not the data tuples from the original dataset. Therefore, more sophisticated analysis is required for generating a global reduction function.

Our compiler uses an algorithm for this purpose which we summarize below. The basic idea is to extract a part of the code that impacts the output, either directly or indirectly, from the GNL. This is done by a static analysis technique called program slicing [78]. After a program slice is extracted, certain statements are either removed or modified. These include

- Statements that are only control dependent on the input data. Because input data is not visible at the global phase, control dependence on input data tuples can be discarded.
- Statements that create data dependence on the input data. For the same reason, we can replace each instance of the input data (bound to the GNL variable) with references to remote copies of output from other nodes.

The remaining statements are used in the body of the global combination function. The detailed algorithm is shown in Figure 7.2. The global reduction function for the Satellite Data Processing application is shown in Figure 7.3

### 7.4 Experimental Results

We used four different queries for our experiments. The first two queries involve processing of scientific datasets. satellite involves processing of data collected from satellites and creating composite images [17]. vscope is an application to support interactive viewing and processing of digitized data arising from tissue specimens [34]. The other two queries are taken from the XMark benchmark [73]. Specifically, we used the
input : GNL g
let E = \{ output variable v \}\n
s = last statement of S
let O = \varnothing ;
while (s ! = nil )
    \{ if \( LH S(s) \in E \)
       foreach variable \( v_i \) in RHS(s) \{
           E = E \cup \{ v_1 \} ;
           O = O \cup \{ s \} ;
       \}
    \}
    s = prev(s)
\}
Insert a loop \( l \) around \( v \) that ranges over all fields
foreach \( s_i \) \in O \{
   if \( s_i \) contains path expression \( p \) that refers the input
       replace \( p \) with \( tmp . v \)
\}
foreach \( s_i \) \in O \{
   if \( v \in RHS(s_i) \) and \( tmp_v \in RHS(s_i) \) \{
       insert \( s_i \) into loop \( l \)
       insert all child statement \( s_{ij} \) of \( s_i \) into \( l \)
\}\}

Figure 7.2: Algorithm for Synthesizing Global Reduction Function from GNL

queries 5 and 20 from this set, as they used aggregation operations and matched the processing structure we have targeted. Our experiments were conducted on a cluster of 900 MHz Pentium III machines, connected by a Gigabit Ethernet. Each node has 256 MB of main memory and 18GB of local disk. We ran our experiments on 1, 2, 4, 8 nodes of the cluster.

For the satellite and vscope applications, we uses XML datasets with sizes of 200 MB and 400 MB. Since all of the elements in these two applications are traversed to generate the results, projection does not alter the sizes of these two datasets significantly. The output of satellite is a \( 5 \times 5 \) array and the output of vscope is a \( 16 \times 16 \) array.
for (i = 1; i < N; i++) {
    rc = MPI_RECV(&tmp, size, ... &status)
    for (i = 0; i <= (maxx-minx); i++)
        for (j = 0; i <= (maxx-minx); j++)
            if (v[i][j] <= tmp_v[i][j])
                v[i][j] = tmp_v[i][j];
}

Figure 7.3: Generated Code for Global Reduction

Thus, the relative overhead of communication is quite small for these two applications. The results are shown in Figures 7.4 and 7.5. These speedups are almost linear in both the cases.

The results from the parallelization of XMark queries are shown in Figures 7.6 and 7.7. For these queries, we used the standard data generator with factors 5 and 10, which resulted
in datasets of size 500 MB and 1000 MB, respectively. For Query 5, the projection shrinks the two original datasets to 4.3 MB and 8.6 MB, respectively. For the Query 20, the sizes of the datasets after projection are 32 MB and 64 MB. The ratio of communication to local processing is somewhat higher for these queries. We observe good speedups for the Query 20 on both the datasets. Since the sizes of the final datasets for Query 5 are relatively small, communication and other overheads start to dominating on 4 and 8 nodes, which leads to smaller speedups.
Figure 7.6: Parallel Performance: XMARK Q20

Figure 7.7: Parallel Performance: XMARK Q5
CHAPTER 8

CONCLUSIONS

Development of applications that process large scientific datasets is often complicated by complex and specialized data storage formats. Recently, XML has rapidly emerged as a general medium for exchanging information between organizations. Similarly, researchers generating large data-sets from scientific simulations may make them available in XML format to other researchers needing them for further experiments.

With the rapidly increased popularity of XML, query and processing of XML data has become a very important topic. Most recent work in this area has been in the context of XQuery, which is the XML query language developed by the World Wide Web Consortium (W3C).

The goal of this dissertation is to provide a practical framework for the efficient compilation of XQuery queries to facilitate the development of data intensive applications. We have concentrated on two processing models that are essential to data intensive computing: distributed-memory processing and stream processing. We reiterate our thesis statement: *By designing new techniques and modifying existing techniques from both compiler and database communities, we show that advanced program analysis as well as loop transformation and parallelization techniques can allow efficient execution of XQuery.*

This section summarizes the contributions of the work described in this dissertation.
8.1 Contributions

Currently, for distributed-memory processing, our work focuses on optimizing and parallelizing scientific applications expressed with XQuery in a cluster environment. So far, in the area of optimization, we have developed a new set of optimization and transformation algorithms for XQuery, which are based on a new internal representation referred to as Generalized Nested Loop (GNL). These optimization techniques include aggregation rewrite, loop fusion, loop interchange, and aggregation remapping. Since XQuery is a very powerful and complex functional language, to enable the above optimization techniques, we also provide new algorithms to handle arbitrary recursive function and the type system in XQuery. As XML and XQuery are being used for larger datasets, parallelizing XQuery execution can enable faster response. In the area of parallelization, GNL offers a convenient basis for parallelization of XQuery. We present techniques for enumerating parallelization strategies, cost-models for choosing the optimal one, and algorithms for parallel code generation.

For stream processing of XQuery, we have proposed to use Data Flow Graph and apply a series high-level transformation based on it. The goal of these transformation techniques is to enable a single-pass evaluation strategy for the original query. Further, we have presented a methodology for determining when an XQuery query, possibly after the transformations we introduce, can be correctly executed with only a single pass on the dataset. Based on a SAX parsing engine, we have developed a new technique to generate efficient code for stream processing.

To further simplify the application development over scientific datasets we provide a solution to use XML Schemas as a high-level abstraction of a dataset to developers. A corresponding low-level Schema describes the actual layout of data is used by the compiler.
for code generation. A systematic way for translating the high-level code to a low-level code that achieves high locality and efficient execution is also provided.

We have implemented and evaluated the above techniques. Results from several XMark queries and scientific data processing queries show large improvements from new optimizations and good parallel speedups.

8.2 Future Work

In today’s Internet era, XML has been widely accepted and become increasing popular in almost every domain. Accordingly, there arises the urgent need for language and compilation support for using XML in different domains. Our work presented in this dissertation can be expanded in the following ways to serve the above purpose:

Support General Database Queries for Parallel and Stream Processing

As a flexible declarative language integrating XPath and FLWR expressions, XQuery presents challenges for efficient optimization and processing. Our current work on compilation and parallelization of XQuery focuses only on queries involving reduction operations. These reduction operations are normally specified through user-defined recursive functions or internal functions, and are limited to aggregate on single datasets. Because of the large volume of data involved in today’s database, it is also desirable to execute general database queries on parallel platform or in a stream processing fashion. Processing of general query expressed in XQuery, especially those involving sequences of joins, requires further extension of our current framework.

To parallelize general database queries, we need to extend our earlier cost model to estimate memory, disk and communication cost when multiple datasets are involved. Because transformation of joins involves loop unnesting, where intermediate results must be
buffered, we must also consider characteristics that affect the size of intermediate results, such as value distribution. Further, we need to consider how to maintain and distribute multiple datasets in our current system. In a stream processing environment, when multiple data streams are used in the query, the stream data flow graph must be extended. Also, we need a new single-pass analysis algorithm for multiple input data streams.

**Compilation of XQuery for Distributed and Grid Computing**

As XML has been widely adapted in web-based [37], distributed [14], and grid computing [38], expanding our current work to such areas seems necessary and natural. Since the GNL we proposed in this dissertation uses a nested loop structure commonly supported in imperative languages, it can be easily extended to serve as an intermediate representation for code generation on these new platforms. However, some open issues still need to be resolved, including task scheduling, resource allocation and domain-specific optimizations based on GNL.


