Querying Geographically Dispersed, Heterogeneous Data Stores: The PPerfXchange Approach

Matthew Edward Colgrove
Portland State University

QUERYING GEOGRAPHICALLY DISPERSED, HETEROGENEOUS DATA STORES: THE PPERFXCHANGE APPROACH

by

MATHEW EDWARD COLGROVE

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2002

DEDICATION

This thesis is dedicated to my wife, Kathy, and son, Reed, whose patience and support made this thesis possible.

ACKNOWLEDGEMENTS

I gratefully acknowledge and thank the follow people for their help with this thesis.

Dr. Karen Karavanic for her advice, guidance, and entrusting PPerfXchange to me.

Christian Hansen for providing the example data and insight into PPerfDB.

Brian Kearns for his proof-reading expertise and moral support.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

A group of scientists meet at a conference and discover each is working on similar problems. They exchange ideas, discuss future needs, and find that each have data useful to each other's research. They decide to collaborate and return to their respective labs excited about the new prospects. However, when it comes time to actually exchange their data, they find each have very different methods for storing and organizing the data. The process of translating the data into each others respective format becomes a laborious chore. While the new information would greatly help, the translation quickly becomes too time consuming to be useful. Ultimately the collaboration wanes, and each scientist continues on separate paths.

What was needed for these scientists was a method by which they could quickly retrieve each other's data and translate it into their local format. This thesis offers a solution to this problem and describes an innovative tool, PPerfXchange, which allows remote users to query geographically dispersed data and return the data in a format which can be easily translated into the local format. Such a tool may be used by many areas of collaborative research; PPerfXchange focuses on the exchange of parallel computing performance data.

Computer applications which execute on high performance parallel computer architectures are often extremely difficult to optimize. A variety of tools have been created which gather performance information during the execution of these applications. A single run of an application can generate hundreds of megabytes of information, which can then be analyzed using various visualization tools. However, exchanging gathered information between collaborators can be very difficult due to the large amount of data gathered, the incompatible data formats used, and the time it takes to convert between formats.

The PPerfDB project [12] at Portland State University is creating an experiment management tool for parallel performance analysis. PPerfDB uses multiple sets of performance analysis data and is able to compare results even if the data was collected by different analysis tools. It would be advantageous to the user to compare data from more than the limited set of executions that are stored locally. Given that the community of developers that analyzes parallel performance is large, each developer would benefit from the exchange of performance data. To do so, one developer would need to allow another to transfer his or her data set to the local system where it could then be analyzed. This process would entail a lengthy download, translating the data into a format the local system could understand, and then extracting the meaningful data. In most cases, the cost of this process is too prohibitive to be beneficial. Ideally, a developer would query the remote location for the particular data set of interest and receive this data in a format that easily integrates into the local system. To accomplish this, several obstacles would need to be overcome.

The first obstacle concerns how performance data is formatted. PPerfDB uses several external preprocessing scripts to transform various data formats into a common representation. The user must run this script before the data may be analyzed. It may be unknown what format the remote data is in, requiring the local user to transfer the remote data to the local system and then apply the appropriate script. Instead, it would be beneficial if the user had the option to do this translation step at the remote site. Each site would only be responsible for translating their own data into the common representation while not needing to know how the other sites format their data.

The second obstacle is how the remote data is stored. Given the wealth of data storage mechanisms in use, i.e. databases, text files, XML documents, and binary files, the remote site may employ one or more means to store their data. For instance, a site may store archived data in a relational database and non-archived executions in a text file. The developer should not need to know how the data is stored, and should be able to query all of the potential data stores in a uniform manner.

The final obstacle is the volume of performance data. Each execution potentially generates hundreds of megabytes of data. If the developer is interested in only a particular performance measurement, only the data corresponding to this measurement should be returned. Hence, the developer needs the ability to query the remote data for this focus. In rare cases the volume of data may be small enough to not warrant a query. However, for the purposes of this thesis it is presumed that the volume of each data set is large and querying this data set would significantly reduce the amount of data needing to be transferred.

As part of the PPerfDB project, I have developed PPerfXchange to allow scientists to easily exchange performance data by solving the obstacles described above. To facilitate data exchange between the collaborating scientists, each scientist maps his or her data to a common naming convention described by a global XML schema. The data is published as a set of virtual XML documents, an XML interface to a local data set, based upon this global XML schema. The virtual XML document performs the mapping between the local data and the global format, and allows the local data to remain in whatever data store the local site uses. To retrieve data from a remote site, a scientist queries a virtual XML document using the XML query language, XQuery. When the XQuery arrives at the remote site, PPerfXchange queries the virtual XML documents which in turn translate and retrieve the local data. The resulting data set is returned to the scientist as an XML document with its form defined in the XQuery. Only the data of interest is retrieved, reducing the amount of data transferred.

This thesis details PPerfXchange's approach for querying geographically dispersed heterogeneous data stores. While elements of PPerfXchange's method have been implemented for other application areas, PPerfXchange shows how these elements can be applied to parallel performance analysis. The accomplishments of this thesis are:

- The design of an architecture for PPerfXchange, giving a uniform method to query heterogeneous data stores;
- A proof of concept prototype implementation of PPerfXchange including a partial implementation of an XQuery processor and a relational database virtual XML document; and
- Evaluation of PPerfXchange using example parallel performance analysis data.

Chapter 2 of this thesis gives some background for PPerfDB, XML, XML Schemas, and XQuery. Chapter 3 details the overall architecture of PPerfXchange. Chapter 4 describes the implementation of a PPerfXchange prototype. Chapter 5 gives an example global XML schema, details a database containing parallel performance data, describes how the local database schema is translated into the global XML schema, and concludes with several example XQuery use cases. Chapter 6 highlights work related to PPerfXchange. Chapter 7 concludes with future work.

2. Background

This chapter gives a brief introduction to parallel performance analysis using PPerfDB, XML, XML Schemas, and XQuery.   PPerfXchange makes extensive use of both XML and XQuery to retrieve data for geographically dispersed, heterogeneous parallel performance data stores. As such, a brief overview of these languages is needed. This chapter is not intended to give a complete guide to the languages; rather it intends to highlight the portions of the languages that PPerfXchange accepts.

2.1 PPerfDB

For a programmer writing an application for use on a parallel architecture, the process of optimizing code is often a difficult task. In order to aid the programmer, a variety of parallel analysis tools have been developed [4]. These tools aid in discovering bottlenecks and poor performance by using instrumentation, performance libraries, or both to measure particular aspects of the application. Some tools also include visualization utilities for displaying performance measurements. However, most tools analyze only a single run of the application at a time. For comparing multiple runs, it is up to the developer to determine the differences. In addition, if multiple performance tools are used, comparing the various results is made difficult by the different data formats and specific measurements taken.

The PPerfDB project [12] at Portland State University is creating an experiment management tool that uses multiple sets of performance analysis data. The developer is able to study particular aspects of the parallel application's performance. If a particular question cannot be resolved using the existing data, then a new run of the application is performed and dynamic instrumentation inserted to measure this aspect of the application. Dynamic instrumentation uses the Dyninst or DPCL libraries to insert code into a running process to gather the performance data. The overall goal of the PPerfDB project is to remove the developer from the analysis work and have PPerfDB self-tune the application. Figure 1 gives a diagram of PPerfDB's overall architecture.

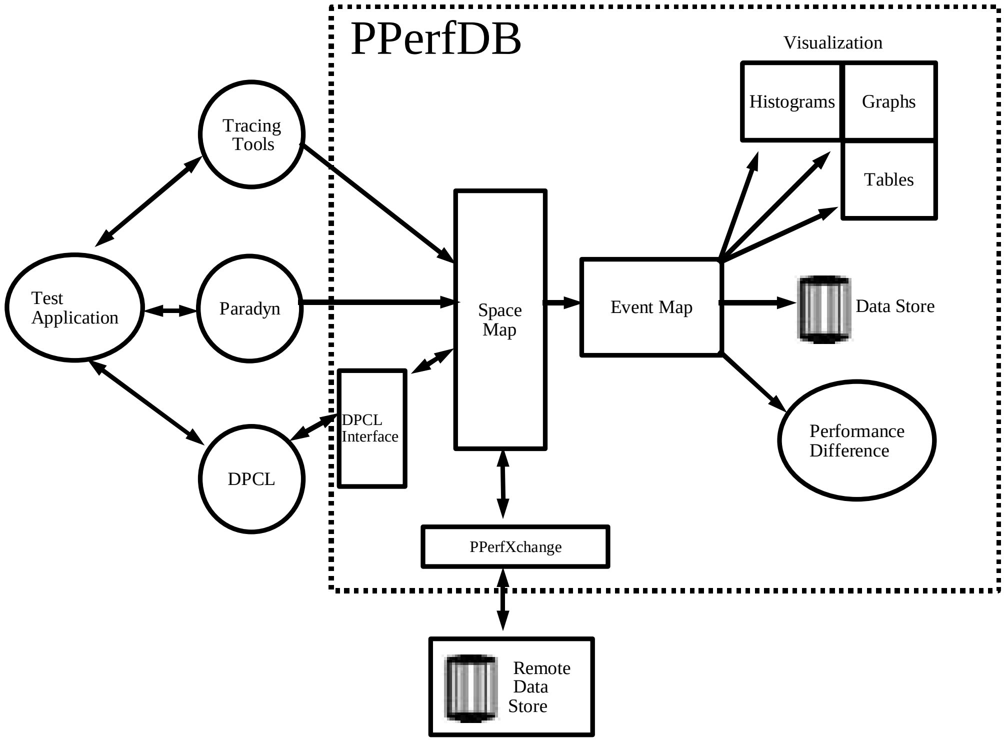
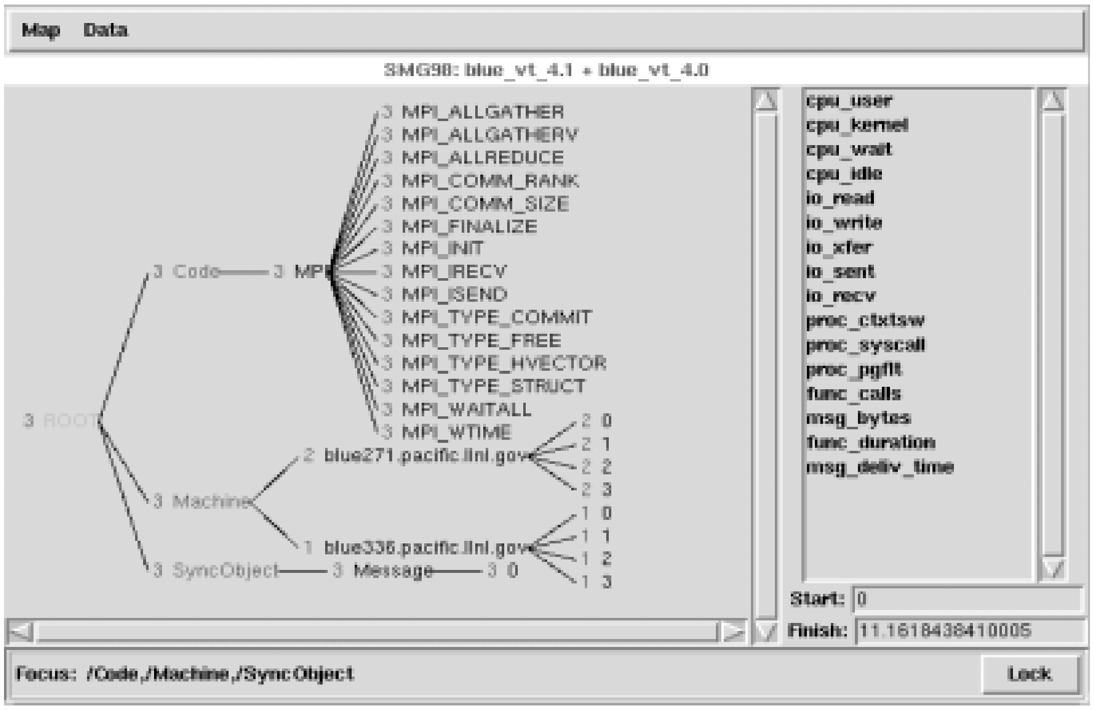


Figure 1: An Architectural Overview of PPerfDB

This figure gives an overview of the PPerfDB architecture. Shown is how PPerfDB gathers data from multiple performance tools, creates a Space Map of the gathered data, creates an Event Map, and then visualizes the results. A performance difference operation can be applied to the Event Map or the resulting data can be stored for future evaluation.

To correctly compare an application's performance data using multiple performance tools, a common data representation is used. PPerfDB gathers performance tool data formatted in the common representation into a Space Map. The Space Map contains the data from multiple executions having some common parameter that the experiment will compare. For example, the experiment may determine how varying the number of processors affects an application's performance when using a common platform. Each execution's data is stored as a hierarchy of identifiable resources, such as code modules or process identifiers, with each execution assigned a unique power of two as an identifier. An Event Map gives a view of this hierarchy. Each node contains a label with the resource name and an execution identifier. A structural merge operation combines the various execution trees into a unified Event Map view (Figure 2). The execution identifier of the merged tree is the sum of the individual execution identifiers. For example, if execution one and two both were run on processor one, the resulting execution identifier for processor one is three. Once an Event Map is constructed, the user may select a focus, i.e. one resource from each path, a metric, and time interval, to compare the executions' performance. A metric might be the number of function calls performed, the duration of a function call, or the percentage of CPU utilization. PPerfDB uses a visualization tool, such as a graph or histogram, to display the results.



SMG98: blue_vt_4.1 + blue_vt_4.0

Start: 0

Finish: 11.1618438410005

Focus: /Code,/Machine,/SyncObject

Figure 2: An Example Merged Event Map

This screen shot from PPerfDB gives an example of a merged Event Map and associated metrics.

Figure 2 gives an example of a merged Event Map using two executions, labeled 1 and 2. The main frame shows the available resources along three paths. The first path contains the MPI function calls performed during the execution of the application. The second path indicates the particular machine that each execution was run on and the processor identifier of the machine. Blue 271 and 336 are nodes on Lawrence Livermore National Laboratory's Blue Pacific supercomputer. The final path indicates the MPI message tags. The right-hand window gives the user a list of available metrics. The bottom window indicates the focus selected by the user. The start and end fields give the time range the user is interested in examining. Figure 3 gives the resulting visualization for the comparison of the CPU idle time for processor 3, obtained by selecting the focus "/Code,/Machine/blue.pacific.llnl.gov/3,/SyncObject."

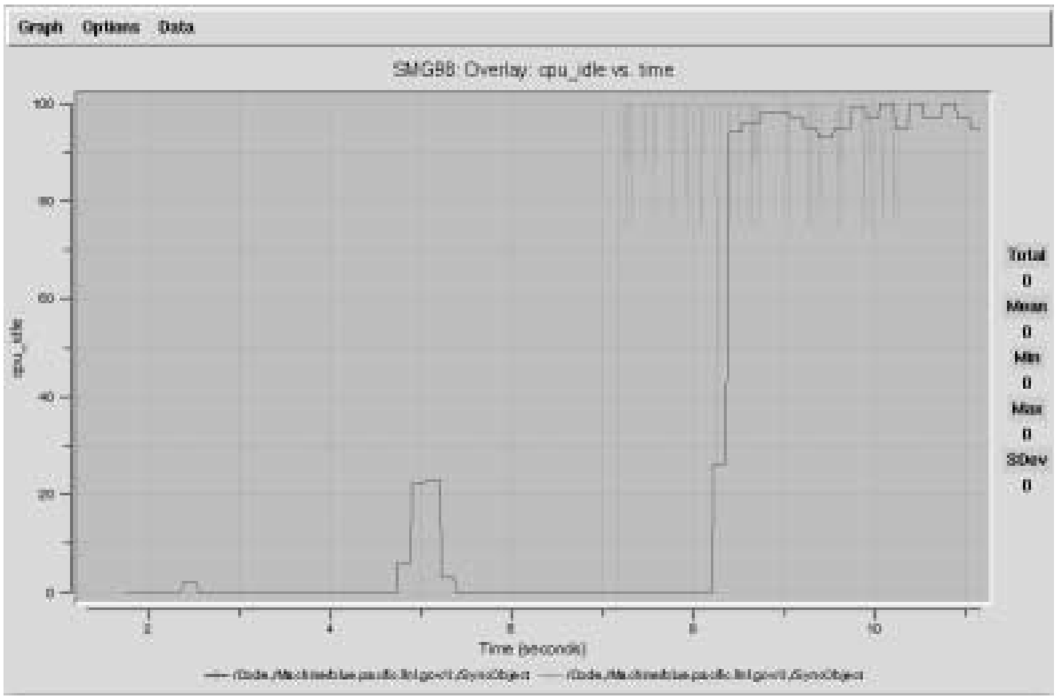


Figure 3: Comparing CPU Idle Time for Two Executions

Shown is a graph comparing the CPU idle times from two executions of SMG98 for the selected focus "/Code,/Machine/blue.pacific.llnl.gov/3,/SyncObject."

2.2 XML

XML (Extensible Markup Language), developed by the World Wide Web Consortium (W3C), presents a standard method for formatting data and documents. XML enforces a rigid structure for its documents, allowing multiple parties to easily exchange data since any XML aware client can read any XML document. While rigid in structure, XML can be customized for a particular application. XML uses user-defined labels, also called tags or element constructors, to define the various elements of a particular document or data set. XML is similar to HTML in that both are mark-up languages. However, instead of using tags to define the formatting of text, XML uses tags to define the semantics of individual elements. XML formats documents through other means, such as cascading style sheets.

The structure of a well-formed XML document is a hierarchy of elements with leaves containing the actual content of the XML document. Elements may have attributes indicating additional information about the element. This structure can represent a wide variety of data including a relational database table, or a book. The table would have a flat hierarchy with many sibling nodes, while the book would have an extensive hierarchy, i.e. book, chapter, section, paragraph, sentence, word, but fewer siblings.

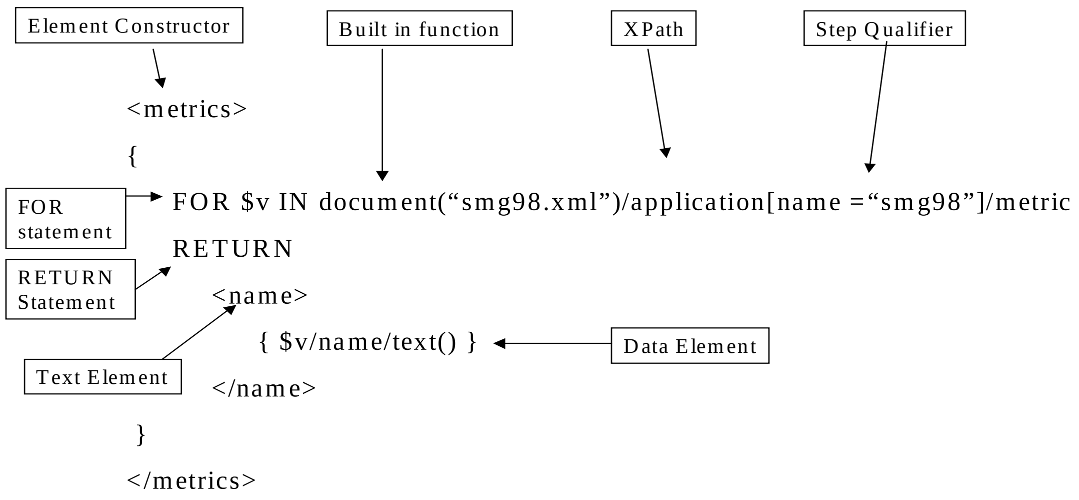
While the XML family is extensive, the core of XML is quite simple. To export a particular data set, label each element of the set and arrange the elements in a hierarchy to define relationships. However, just because a third party can parse the XML document and read its tags, it does not mean that it can be interpreted. To give meaning and a specific structure to XML documents, the W3C XML Schema recommendation [27] gives a framework for multiple parties to create a common vocabulary and rule set governing the form of their XML documents. This allows each to publish XML documents that others can not only read, but also interpret.

2.3 XQuery

To query XML documents, PPerfXchange uses the W3C’s working draft XML query language, XQuery [28]. XQuery is the result of many years of collaborative effort and merges ideas from other XML query languages such as XQL, XML-QL, and Quilt into a single XML Query language. XQuery requests particular elements of XML documents and transforms them into other well-formed XML documents.

The FLWR (FOR LET WHERE RETURN) statement allows for iteration, aggregation, and joins. The RETURN statement lists a set of element constructors or other FLWR statements and defines the structure of the resulting XML document. The client can define any structure and tagging they need to easily integrate the returned data into their system. Hence, the data should need little additional transformation when it is returned. FOR statements iterate through each element of the XML document and apply the RETURN section to each element. LET statements assign a statement to a variable. Whenever the variable is encountered in the RETURN statement, the LET statement is evaluated. LET statements may be text or an element. If it is an element, the entire data set is evaluated. The WHERE statement allows for the selection of particular elements in the XML document and gives the join properties for multiple XML documents.

XQuery accomplishes projection using XPath syntax. XPath gives a path, similar to a file path, to the element of interest within the XML document. In addition to projection, XPath allows selection through the use of step qualifiers that place constraints on the particular element. XQuery has an extensive set of built in functions such as count, min, max, document, etc., which aid in the discovery and transformation of elements. Users may also define their own functions. Figure 4 gives an example XQuery that selects all metric data from the "smg98.xml" document where the application is "smg98". The FOR statement iterates through each metric element and returns the metric's name.



```
<metrics>
{
  FOR $v IN document("smg98.xml")/application[name ="smg98"]/metric
  RETURN
    <name>
      { $v/name/text() }
    </name>
 }
</metrics>
```

Figure 4: Components of an XQuery

Shown is an example of an XQuery. The labels indicate the various components of the XQuery. The element constructor defines elements of the resulting XML document. The FOR statement iterates through an XML document, retrieved by the built-in document function. The XPath defines the path within the XML document to the desired target element. A step qualifier selects only elements matching the given criteria. RETURN statements define the element constructors which are applied to each of the found target elements. A text element returns a literal tag while a data element returns an element of the queried XML document.

3 The PPerfXchange Architecture

The following chapter begins by showing how PPerfXchange can be used to retrieve heterogeneous data from geographically dispersed locations. An overview of the major components of PPerfXchange are then detailed. This chapter gives a high level description of PPerfXchange with some components not having been implemented in the PPerfXchange prototype. Please refer to chapter 4, the PPerfXchange prototype, for specific information about the implemented components.

3.1 Using PPerfXchange

The first step in using PPerfXchange is for a group of collaborating scientists to decide that exchanging data would help further their collective research. If the amount of data to be exchanged is small and the data exchange is infrequent, then the use of PPerfXchange would not be necessary. However, if the amount of data to be exchanged is large, and it is updated often requiring frequent exchanges, then PPerfXchange offers an innovative method to allow these scientists to uniformly exchange data. The scientists begin by creating a global XML schema to represent a common naming convention and format for their collective data. PPerfXchange makes no assumptions as to the specific details of the global XML schema.

Next, each site publishes the data they wish to share as a set of virtual XML documents based upon this global XML schema. A virtual XML document is an XML interface to a site's local data set. The site's data resides in what ever data store they choose with the virtual XML document performing the translation between the data store and the global representation. (A complete discussion of virtual XML documents is given in section 3.3.) To aid in publishing data, a graphical virtual XML document configuration tool, similar to Microsoft's XML View Mapper 1.0 schema mapping utility [25], is used. This tool would give a visual representation of the global XML schema and the data store's schema, and allow the scientist to map the schemas as well as place constraints as to the specific data set published.

With the global XML schema created and a site's data published as if it were an XML document, the other scientist in the group can begin asking queries using XQuery. An XQuery is formed by PPerfDB whenever a scientist requests data from a remote site. Multiple XQueries may be used. An initial query might request meta-data about the remote site's published data and store this information in PPerfDB's Space Map. Subsequent queries would be asked in order to retrieve data for visualization once the scientist has selected a particular focus from an Event Map. While PPerfXchange is designed for use with PPerfDB, it will respond to any client, such as a web browser, able to speak XQuery. Once an XQuery is sent to a remote site, the local PPerfXchange server processes the XQuery and returns the resulting data in an XML document. The resulting XML document's structure is defined in the XQuery itself allowing the client to define the format of the resulting data.

The remainder of this chapter discusses the internal components of the PPerfXchange architecture. Figure 5 (next page) gives an overview of this architecture. Section 3.2 describes the XQuery processor and section 3.3 discusses the virtual XML document construct.

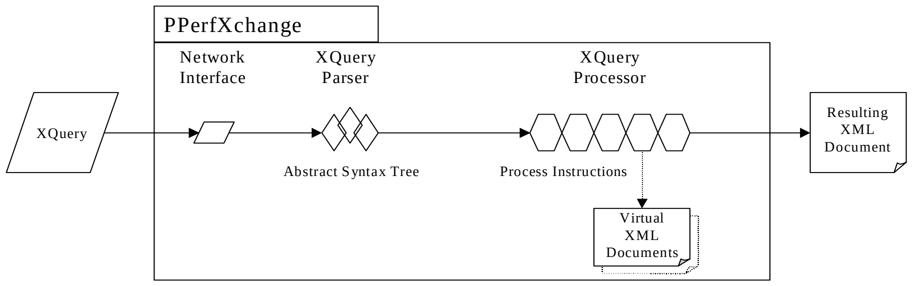


Figure 5: An Architectural Overview of PPerfXchange

Shown is an overview of PPerfXchange’s architecture. An XQuery is sent to the remote site where it is received by PPerfXchange’s network interface. The query is then parsed, creating an abstract syntax tree (AST). The AST is transformed into a series of process instructions that create a resulting XML document by retrieving data from a set of virtual XML documents.

3.2 Parsing and Processing an XQuery

The XQuery parser parses the XQuery into an abstract syntax tree (AST) and determines if the query is well-formed. If the XQuery is ill-formed, the parser returns an error message to the user as to the source of the parse error. If the XQuery parser is successful, the AST passes to the XQuery processor for transformation into a series of processing instructions, and a set of native and virtual XML documents.

The processing instructions are then executed in order. An instruction may either create a static text node, create a node using queried data, or execute a set of instructions such as a FLWR statement. The created data and text elements are returned sequentially to the user as a resulting XML document with its form defined in the user’s XQuery. By sending the resulting elements incrementally, PPerfXchange reduces the amount of memory needed locally and ensures the successful retrieval of even the largest data sets.

3.3 Unified, Virtual, and Native XML Documents

The processing instructions retrieve data from a unified XML document. A unified XML document represents a single view over the data set of interest, which may span multiple data stores. It combines all requested virtual and native XML documents into a single virtual XML document, and applies qualifiers and aggregate functions to this document. Figure 6 illustrates the document hierarchy of a unified XML document.

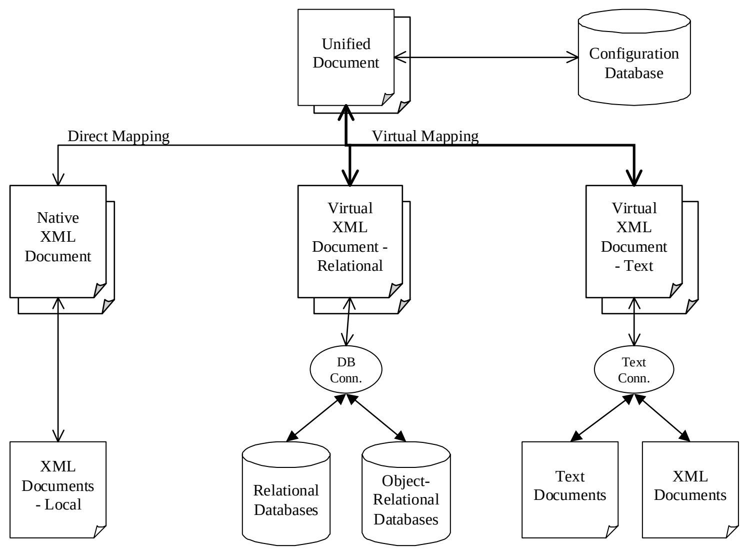


Figure 6: The Unified XML Document Hierarchy

Shown are the components of a unified XML document. The unified XML document represents an XML document based upon the global XML schema. Virtual XML documents transform a particular data model into the global representation while connection objects represent the actual link to a particular data store. Native XML documents are XML documents written in the global XML schema and thus require no mapping.

A unified XML document is formed when an XQuery “document” function is encountered in the FOR clause of a FLWR statement. The content of the XML document is determined by the path stated in the FOR clause’s XPath expression. If a query segment consists of a single XML document without a WHERE statement qualifier or an aggregate function being applied, the unified XML document abstraction is bypassed and only a single virtual or native XML document is used.

The virtual XML documents model the structure of an XML document based upon the global XML schema. It can be thought of an XML interface to a data set in that it maps the underlying local data store’s schema to the common XML representation. Each class of virtual XML document represents a common data model such as a relational database or text file. Additional modules can be written to be able to support a wider variety of data stores. A connection object performs the connection to a specific data store such as a MySQL database, and performs the actual data retrieval. Multiple connection modules can be written to support the various data stores. Native XML documents are XML documents whose format and content are based upon the global XML schema. Since a native XML document’s structure matches the global XML schema, schema mapping is not needed.

The configuration database contains the information needed for PPerfXchange to access and model the remote data. This includes the names of the published XML documents as well as the XML document’s data store type, location, connection method, and description. Other entities describe the structural mapping between the local virtual XML document and global XML schema. The configuration database’s meta-data is published as a virtual XML document and can be queried in the same manner as the site's published data.

4. The PPerfXchange Prototype Implementation

In this chapter, the implementation of the PPerfXchange prototype is detailed. The prototype implements partial or complete versions of all PPerfXchange components. Rather than focusing on a complete implementation of certain components, it was decided to create limited versions for all components. This allows for the evaluation of the PPerfXchange approach as a whole rather than focusing on a single aspect of the approach.

Section 4.1 begins by examining how an XQuery is created by the client and sent to the PPerfXchange prototype. Section 4.2 discusses the parsing of the XQuery and some of the implementation details of the XQuery parser. Section 4.3 details how the XQuery is processed using processing nodes and process instructions. Finally, section 4.4 shows how virtual XML documents are used to map schemas and retrieve data from relational databases.

4.1 Sending an XQuery

The first step to any query is the XQuery formulation by the client. While PPerfXchange has been developed for use with PPerfDB, the actual client could be any program with the ability to send an XQuery to the PPerfXchange prototype, such as a web client. The PPerfXchange prototype does not currently use any headers or authorization methods. Rather, it simply takes an XQuery and returns the results. The PPerfXchange prototype supports portions of the XQuery language and allows the client to formulate a wide range of queries. The PPerfXchange prototype allows queries to retrieve and manipulate the data from a single virtual XML document at a time. Multiple virtual XML documents may be queried in succession. Qualifiers are used to limit the size of the returned data. Users may define their own return element tags, or use the default global tags for the returned XML document.

An example question a client might ask is “For the application SMG98’s fourth execution, what was the CPU idle time for the focus /Code/MPI/MPI_ALLGATHER,/Process/4,/SyncObject/Communicator/0.” If the question cannot be satisfied locally, the client may retrieve this information from a remote site by reformulating the question into an XQuery. Figure 7 shows an example XQuery to answer this question.

```
<smg98>
{  FOR $x IN document(“smg98.xml”)
      /Application[name=”SMG98”]
      /execution[id=4]
      /metric[name=”cpu_idle”]
      /focus[path1=”/Code/MPI/MPI_ALLGATHER”]
            [path2=”/Process/4”]
            [path3=”/SyncObject/Communicator/0”]
      /data
   RETURN
      <data>
         {$x/time}
         {$x/value}
      </data>
}
</smg98>
```

Figure 7: An Example XQuery

The shown XQuery requests all data from the fourth execution of the SMG98 application with the metric CPU idle time and the focus “/Code/MPI/MPI_ALLGATHER./Process/4./SyncObject/Communicator/0.”

The query is created to retrieve data from one or more published XML documents located at the remote site. The client may obtain a list of the available published XML documents, as well as the documents' schemas, by first sending an XQuery requesting a list of documents from the remote site's configuration database. Once the client has formed and sent the query to the PPerfXchange prototype, the network interface receives the request. Each request is placed on a queue to be serviced by an XQuery processor. The PPerfXchange prototype is multi-threaded; the exact number of threads is a command line argument. Each thread contains an XQuery processor and processes a single request at a time from start to finish. Once finished, a thread will take the next request off the queue. If no requests are pending, it will wait until one becomes available. When an XQuery processor receives a request, it stores the query temporarily on disk. Storing the query in an intermediary file allows the queries to be arbitrary sizes and helps reduce the risk of running out of memory. Each new query an XQuery processor receives, overwrites the previous query. When the program terminates, the temporary files are removed.

4.2 Parsing the Query

After the XQuery is received, the XQuery parser reads the query from disk and transforms it into an in-memory abstract syntax tree (AST). If the XQuery is ill-formed, the XQuery parser aborts and the XQuery processor sends an error message to the client. Otherwise, the XQuery parser returns the root of the AST to the XQuery processor. The XQuery parser was created using GNU's Flex [15] and Bison [6] utilities. The

XQuery grammar rules [28], an LL(1) grammar, were translated into Bison compliant grammar rules, LALR(1). Figure 8 gives an example of an XQuery grammar rule and its equivalent Bison grammar rule.

```
XQuery Grammar Rule:

      FLWRExpr := (ForClause | LetClause)+ WhereClause? “return” Expr
      ForClause := “for” Variable “in” Expr (“,” Variable “in” Expr)*
      LetClause := “let” Variable “:=” Expr (“,” Variable “:=” Expr)*

Bison Grammar Rule:

      FLWRExpr:
               ForLetList WhereClause RETURN Expr
            |  ForLetList RETURN Expr
      ForLetList:
               ForLetList
            |  ForLetList LetClause
            |  ForLetList ',' LetNextClause
            |  ForLetList ',' ForNextClause
            |  ForClause
            |  LetClause
      ForClause:
               FORVariable IN Expr
      ForNextClause:
               Variable IN Expr
      LetClause:
               LET Variable ASSIGN Expr
      LetNextClause:
               Variable ASSIGN Expr
```

Figure 8: Bison Equivalent Grammar for an XQuery Grammar Rule

Shown is an XQuery grammar rule for the FLWR statement and the equivalent Bison grammar rule.

Each FLWRExpr rule is composed of one or more FOR or LET clauses, zero or one WHERE clauses, and a RETURN expression. The FOR and LET clause rules contain a variable, an expression, and zero or more additional clauses without the FOR or LET identifier. In order to mimic the one or more constructs, new list elements were added to build a left recursive list. While Bison does allow rules to have empty sentences, doing so causes multiple reduce/reduce conflict errors. To create zero occurrences of a clause, a rule is given in its parent rule without the corresponding child rule. In order to mimic the property that multiple FOR or LET clauses can be put together without a new FOR or LET identifier, the “ForNextClause” and “LetNextClause” rules were added.

The resulting AST is comprised of nodes, called xnodes, each corresponding to the major rules. For example, the above rules are reduced to two xnodes, “FLWR” and ”ForLetList”. The parser was tested using the W3C’s published use cases [31]. All queries from this list were accepted. However, user defined functions, context declarations (namespaces), and data types do not have a corresponding xnode. These grammar rules are ignored by inserting an “unsupported” xnode into the AST.

4.3 Process Nodes and Process Instructions

Once the XQuery has been represented as a tree of xnodes, the xnodes are returned to the XQuery processor. The XQuery processor creates a series of process nodes and process instructions to fulfill the client’s request. The process nodes represent an element in the resulting XML document. There are three types of process nodes: text process nodes contain only labels or literal value, data processing nodes contain data from the resulting query, and a document node indicates the start of the resulting XML document and contains header information. Process instructions represent control structures and functions.

The PPerfXchange prototype implements a single process instruction, the FLWR instruction. While additional instructions would be needed to create a fully functional XQuery processor, the FLWR instruction illustrates how instructions can be accomplished. The FLWR instruction contains a variable table, a document list, and a set of process instructions and processing nodes. The variable table is used for retrieving the appropriate virtual XML document when forming data nodes. The documents list contains a linked list of unified XML documents. However, the current implementation is limited to a single virtual XML document per FLWR instruction and the unified XML document construct is bypassed. The scope for the FLWR instruction's data process nodes is limited to this virtual XML document and the virtual XML documents declared in any higher order FLWR statements. While nesting of FLWR statements is allowed in the PPerfXchange prototype, joining documents is not. As such, nested FLWR statements are not recommend since they result in the cross product of the two documents.

Figure 9 (next page) shows the resulting processing nodes and instructions from the XQuery given in Figure 7.   The process nodes and process instructions are created by recursively descending the xnode tree. Semantic rules are applied during this creation process. If a semantic error occurs, the process stops and an error message is returned to the user. The PPerfXchange prototype implements a stricter set of semantics than the formal semantics defined by the W3C [28]. The stricter set is due to the lack of support for other XQuery instructions and functions. For example, a FOR clause's expression

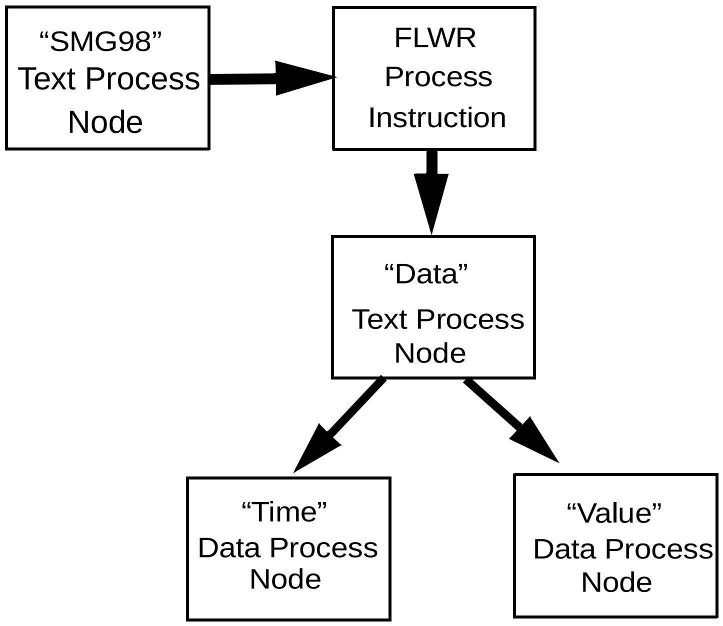


Figure 9: Example Flow of Process Instructions and Process Nodes

The shown process instruction and process nodes represent the processing of the sample XQuery given in Figure 7. The XQuery processor recursively descends through the process node and process instruction hierarchy. Text process nodes return a literal tag to the client while data process nodes return content retrieved from a virtual XML document. The FLWR instruction iterates through each item in the virtual XML document and applies its child process nodes to the virtual XML document.

must contain a document function or a variable. XQuery semantics would allow any document related expression, such as an aggregate function. After the process nodes and process instructions are created, the query is executed. Starting at the document node, the XQuery processor recursively descends through the process nodes and process instructions. The document node contains the header information about the return document. This information is contained in a document configuration object passed to the XQuery processor by the main program. The configuration information is stored in a text file and read by the main program. It contains such information as the site contact, XML document version number, and the character encoding used. The document header also contains a randomly generated document number for use in logging. The PPerfXchange prototype can be modified to store the tracking number and information about the requested XQuery for statistical use.

When encountering a process node, an opening tag is generated and sent to the client. After the subsequent process nodes and process instructions have been executed, the XQuery processor returns to this node, and the closing tag is sent. Data process nodes contain a pointer to a virtual XML document and a path in the virtual XML document to the elements of interest. The data process node has two options, return a complete XML element based on the path, or just the element's text. The type sent is determined by the client's XQuery. A text or data function call after the node's path, i.e. "element-Name/text()", indicates only the text of the node should be returned. Using just the element name indicates the complete XML element is returned.

Process instructions themselves do not return data to the client; rather they indicate that special processing is to be done. In the case of the FLWR process instruction, the XQuery processor will open a virtual XML document, apply qualifiers and ranges, and then move to the first data item in the result set. See section 4.4 for more information about how documents are formed and queried. Next, the FLWR process instruction will iterate through each data item and apply its process nodes and process instructions to each. Once all the data items have been processed, the FLWR process instruction will close the document and proceed with the next process node or process instruction.

4.4 Virtual XML Documents

This section details the methods used to create and retrieve data from virtual XML documents. Virtual XML documents may represent any number of different data stores, including structured text files or databases. The PPerfXchange prototype implements a single virtual XML document class used to model a relational database. Each virtual XML document is created from a common virtual base class and allows for the development of other future virtual XML document classes. The specific type of virtual XML document created is determined by information contained in the configuration database. Section 4.4.1 outlines the configuration process. Section 4.4.2 details how a relational database schema is represented as a global XML schema. Section 4.4.3 shows the process of forming a relational database virtual XML document and section 4.4.4 details how data is acquired by the XQuery processor.

4.4.1 Configuring a Virtual XML Document

When the XQuery processor encounters a document function while creating the process nodes and process instructions, a new virtual XML document object is instantiated. However, the specific virtual XML document employed is unknown by the XQuery processor. The document’s name and database configuration object (see below) are passed to a global function called “createDocument”, which determines which type of virtual XML document to create and returns an object of this type to the XQuery processor.

The “createDocument” function first connects to the local configuration database and issues an SQL query to the “documents” table requesting information about this virtual XML document. The document identifier, connection type and connection identifier are returned. If no document record exists, an error is returned to the client. Figure 10 gives the schema for the XML document configuration tables.

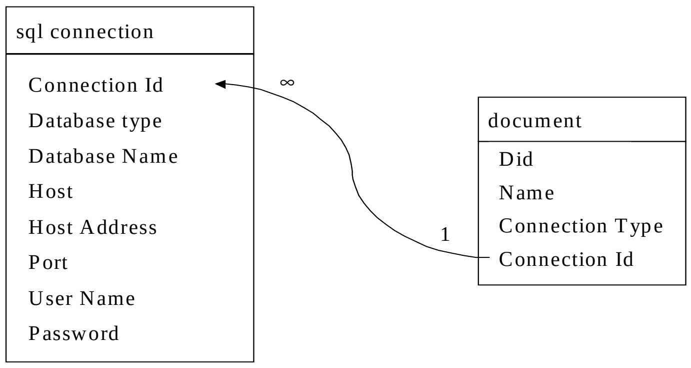


Figure 10: The Configuration Tables

Shown is the relational schema of the configuration database’s configuration tables. These tables are used by the PPerfXchange prototype to create the appropriate virtual XML document and connection object.

A second SQL query is made to the table containing the particular connection type for this document. Currently, there is a single connection type, a relational database (SQL). As additional connection types are supported, additional connection tables can be added. For relational database connections, the database type and name are the only required fields. Host, host address, and port can be used if the database is not located on the same location as the PPerfXchange prototype. User name and password are used to access the database as a different user than the user that started the PPerfXchange prototype.

For a relational database connection, a database configuration object is used to store the configuration information. This information is used by an object of the database connection class to create the actual connection to the database. The database connection class is a virtual base class allowing for uniform access to multiple relational databases. Each type of relational database has its own connection class based upon this common interface. The PPerfXchange prototype supports the PostgreSQL [19] object-relational database. Support for additional databases can be added by creating a corresponding database connection classes.

4.4.2 Representing Global XML Schemas in a Relational Database

The PPerfXchange prototype does not attempt to completely translate the XQuery into an SQL query. Rather, only a small portion, the XPath, is actually translated. From the example XQuery in Figure 7, the "/Application[name="smg98"]" part of the XPath is translated into the following generalized SQL statement:

```
SELECT * FROM Application WHERE name = "smg98"
```

To map the virtual XML document to the relational database's schema, each level in the virtual XML document's hierarchy is represented as an SQL statement. Retrieval of child elements is accomplished using additional subqueries. While this method is simple to implement and allows for easier schema mapping, performance suffers when the document has an extensive hierarchy. The multiple subqueries act as nested SQL queries with each subquery being re-evaluated each time a higher order item is moved. The translation process is accomplished using several tables in the configuration database. The tables are based on the structure of an XML document and allow the mapping of a particular XML document to a relational view. Figure 11 shows these tables and their relationships.

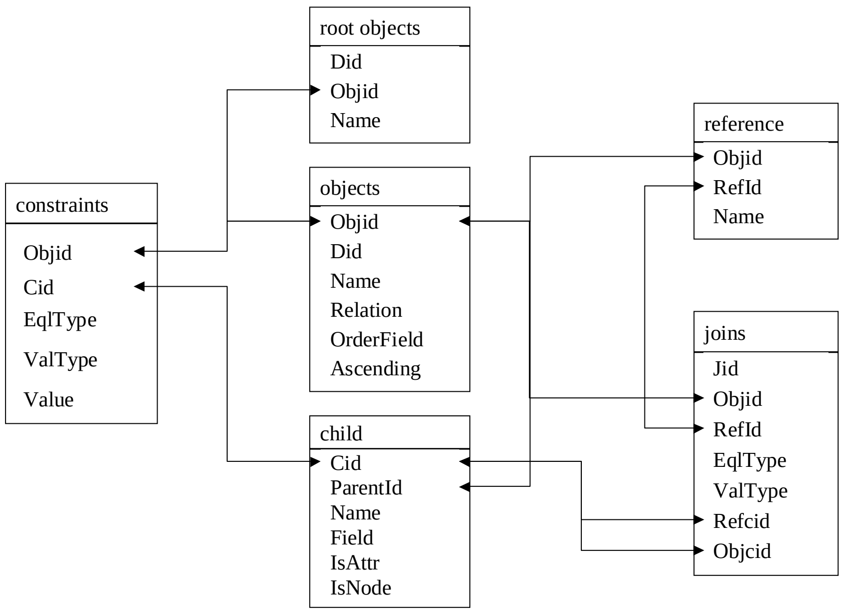


Figure 11: Translating A Relational Database Schema to a Virtual XML Document

Shown are the configuration database tables used to translate the local relational database's schema to a particular virtual XML document. The tables model the hierarchical structure of an XML document. The arrows indicate the relationships between the relational database's tables.

The "objects" table contains the identifier and name of each global XML schema's non-leaf elements, called objects, for all documents. Because each document can contain objects having the same name, the document identifier is given so only the correct objects for a particular virtual XML document are retrieved. The "name" field is the name of the object in the global XML schema and the "relation" field is the relational database’s entity that models this object. The “root objects” table defines which objects are document nodes, with one root object for each virtual XML document. Each object maps to an entity, table or view, in the relational database that the virtual XML document is modeling. When there is not a one-to-one mapping between a database table and a virtual XML document object, a view must be created in the database that correctly models this object. Entities can be shared among objects.

One key feature of XML is that its data is ordered. However, most relational databases do not inherently keep data in an ordered manner. To help maintain order, the “orderField” and “ascending” fields were added to the “objects” table. The “orderField” indicates which of the object's children determines the order of the data. The “ascending” field is a Boolean value indicating the order direction. The document order of the objects themselves is determined by the object identifier. Lower object ids will be traversed before higher object ids.

Objects may contain children, the leaf elements or attributes of the virtual XML document. To represent the mapping between the children and the relational database, the PPerfXchange prototype uses the "child" table. The “name” field is the global XML schema name for the child. "Field" indicates the field in the parent object's entity to which the child is mapped. The “isattr” field is a Boolean value used to indicate if this child is an attribute of the object. Certain fields in the relational database are sometimes needed to satisfy joins between the virtual XML document’s levels, or indicate ordering, but are not elements of the virtual XML document itself. The “isnode” field is a Boolean flag indicating if the contents of this child are actually part of the virtual XML document or are only used to aid in the processing of the virtual XML document.

In addition to children, objects may contain other objects. The "reference" table contains a list of child objects. The "joins" table is used to define the relationships between the parent and child objects’ relational database entities. For example, if a student object contains a address object, then a join is used the to retrieve an address for a particular student. The "reference" contains the object identifier of the parent, a reference identifier of the child object, and the global XML schema name for the child object. The “reference id” uniquely identifies which join to use for a particular child object. Within the "joins" table, the “object id” is the identifier of the child object. The “refcid” and “objcid” are the child identifiers from the "child" table that join the two objects. The “eqltype” field defines which equality type, i.e. equals, not equals, less than, greater, etc., the join is bounded by. The “valtype” field is the data type of the two child values.

The "constraints" table is used to add qualifiers to a particular object's data. If only certain records of a given entity define an object, adding a constraint eliminates the need to create a separate view. The “cid” is the child identifier of the object's children upon which the projection is based. The object identifier is the object applied to the constraint. The “eqltype” and “valtype” are the same as the "joins" table equality and value types. The “value” field is the literal value of the constraint.

4.4.3 Forming a Relational Database Virtual XML Document

During the execution phase of the XQuery processor, a virtual XML document is created when the “document” function is encountered. The query then applies selection and projection qualifiers to the virtual XML document using the subsequent XPath. Step qualifiers and range expressions are applied to achieve selection and the document path is applied to achieve projection. This path information is stored in the virtual XML document and is used to form the document when the FLWR instruction that contains it is first encountered. The formation of a document is the process of creating an in-memory data structure matching the virtual XML document whose structure is defined in the configuration database. This section first describes how XPath is applied to a document and then details how documents are formed.

Adding a new object is straightforward for simple path navigation, i.e. “/”. The object is simply added to the document’s path object. For recursive navigation, i.e. “//”, all object’s of a given name that are descendants of the current object are added. This could lead to multiple branches within the path. The PPerfXchange prototype does not currently allow a path to contain more than one branch. Hence, only the first descendant in document order is added. To find the path to the descendant, the descendant’s parent object is determined by generating a SQL query to the configuration database. If the parent object does not match the object indicated before the recursive navigation, a second query is made to find the parent's parent object. The process continues until the complete path is found.

Qualifiers are added by passing the virtual XML document the path of the qualified object, the name of the object's child that the qualifier is applied to, the equality type of qualifier, the value to apply, and finally the data type of the value. A range expression constrains the resulting data set from a particular minimum element to a maximum element, for example from element 2 to 10. Since applying a range expression implies the document is ordered, results will only be correct if the order field in the "child" table is specified.

A new document is formed each time an FLWR instruction is entered. For nested FLWR statements, the document is re-formed each time it is encountered. This allows different qualifiers to be applied to each iteration of the FLWR instruction. Qualifiers added during the document's configuration are permanently added, while qualifiers added during the execution phase are only applied to the next formation of the document.

An XML object class represents an object as defined in the "object" table of the configuration database. It is formed by querying the configuration database for its relation, order field, and order direction. Next, it determines if it is the target object, i.e. the last

XPath node, and if the “descendants” flag is true. The descendants flag is used to enable or disable adding the target object’s descendants to the virtual XML document. Disabling descendants increases the efficiency of the PPerfXchange prototype since the subqueries for the descendants are not evaluated and only the target object’s children are used in the query. By default this flag is set to false, but may be set to true using the function call “descendants("true”)” within the user’s XQuery.

If this is not the target node, the XML object creates a new XML object for the next object in the path. If this is the target object and the descendants flag is true, a new XML object is created for each of the target object’s child objects. The parent object queries the configuration database for all joins between the parent object and child objects. Joins are stored in the parent object and applied when the child object is opened. Each child object is formed until all descendents on this path are created. After all descendent objects have been formed, the current XML object’s children are added. The configuration database’s “child” table is queried for a list of children for this object. Each child is added to the object’s children list, ordered by identifier. The XML object stores the child’s global XML schema name, the database field name, a Boolean flag determining whether the child is an attribute, and second Boolean flag determining whether the child is a node and should be part of the resulting XML document. The final query to the configuration database determines the constraints to apply to this object.

During the formation of each XML object, a helper class is employed to build and store this object’s SQL statement. The “sqlBuild” class enables the SQL statement to be built incrementally. The “FROM” clause is set using the object’s “relation” field. The “SELECT” clause is built by adding each child’s “field” value. Permanent “WHERE” statements are added using constraints while variant qualifiers are added just before execution of the SQL query. “ORDER BY” clauses are set using the object’s order field value.

4.4.4 Retrieving Data from a Relational Database Virtual XML Document

This section illustrates the process of retrieving data from a virtual XML document and how the data is returned to the user. The virtual XML document begins by opening its root object. The root object connects to the actual relational database containing the performance data and executes an SQL query created during the formation of the object. If a range expression is used with this object, the object will move to the first record in the range. Next the object’s children are assigned values from the return data set. The object then applies all joins to its child objects. Each child object is then opened. The process repeats for each child.

Each iteration of the FLWR instruction moves to the next record in the data set. The next record is defined as the next record in the target object. When the target object reaches the last item in its set, it is closed and its connection to the database is destroyed. Its parent object moves to its next record and the target object is re-opened, joining to the parent object’s new data item.   The process is the same for objects on all levels in the path.

The virtual XML document returns data to the XQuery processor using an xmlNode object from the GNU’s libxml2 library [33]. The first time a process node, data or text, is retrieved, a new xmlNode is created by recursively moving through each object in the virtual XML document. The object adds its own information and each of its children, attributes and content, to the xmlNode. Children with the “isNode” flag set to false, are ignored. The XQuery processor can retrieve the entire xmlNode or a particular sub-node at a given point in the path. The XQuery processor then either sends the entire retrieved portion of the xmlNode or just the text. Every subsequent retrieval for this record is applied to this current xmlNode. When the object is moved or closed, the xmlNode is destroyed.

5 Examples

This chapter gives an example of how PPerfXchange can be used to query remote parallel performance data. Section 5.1 details an example global schema for parallel performance data. Section 5.2 gives three example performance data databases created from data gathered by Christian Hansen [12] for use with PPerfDB. Section 5.3 illustrates the mapping between the global schema and the performance data databases. Finally, section 5.4 gives some example use cases for retrieving data from the performance data databases using PPerfXchange.

5.1 Example Parallel Performance Global Schema

The initial step in using PPerfXchange is to develop a global schema. The specifics of the schema is left to the participants. So long as each party agrees to the schema, any schema can be used. Figure 12 (next page) gives an example global schema hierarchy for parallel performance data. Appendix A contains the corresponding schema file.

The root element of the global XML schema is the application for which the performance analysis data was generated. Each application contains the name of the application, any general information the author wishes to publish, and one or more executions. Each execution contains an unique identifier, various configuration information, the start and end times of the performance data, and one or more observed metrics. The author can also add zero or more “other” elements. These are meant to allow additional information about a specific execution but cannot be queried. Metrics have a name and zero or more foci. Each focus contains one or more paths and zero or more data elements. Data elements contain a time value as well as the data value observed.

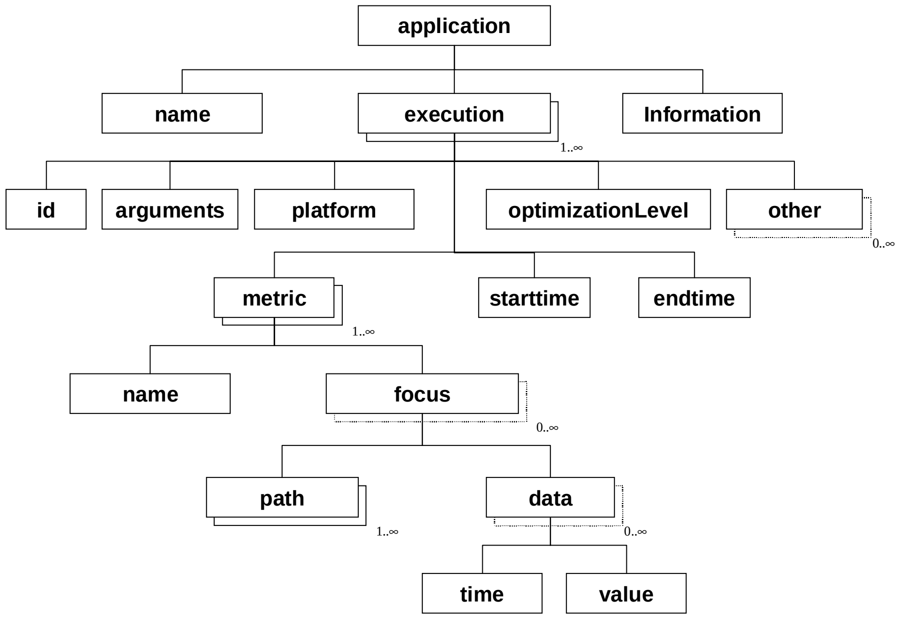


Figure 12: An Example Parallel Performance Global XML Schema

Shown is an example global schema hierarchy for parallel performance data.

5.2 Parallel Performance Database for SMG98

The example data used to show PPerfXchange's features is taken from a set of parallel performance analysis data gathered by Christian Hansen [12]. The data was gathered using the Vampir tracing tool for the application SMG98. SMG98 is a semicoarsing multigrid solver used to solve systems of linear equations that compute finite difference, finite volume, or finite element discrete diffusion equations on distributed memory architectures [12]. The data for each execution of SMG98 was transformed from

Vampir’s data format into a set of text files. One file contains a list of available metrics, a second contains the available foci, and a third contains a listing of metric-focus pairs, the start and end times of the execution, and the name of the file containing the gathered data for this pair. Each focus contains three paths, “/Code”, “/Process”, and “/SyncObject.” Depending upon the specific execution, the number of metric-focus pair entries, and hence the number of data files, ranges from 2 to 2199.

Three executions were used, SMG98_4, SMG98_8, and SMG98_27. The amount of data generated is dependent on the number of metric-focus pairs observed, the amount of time the application was allowed to run, and the number of observable events. Table 1 lists the number of metric-focus pairs, run time, and total size of the data.

Table 1: Test Executions

Execution	Number of Pairs	Run time (sec.)	Total Size of all files (MB)
SMG98_4	2	12.7	20
SMG98_8	1057	5.6	257
SMG98_27	2199	11	248

A PostgreSQL database was created to store the execution’s data. Initially, all executions for SMG were to be stored in a single database. However, the number of data value tuples exceed 20 million for just these three executions. SQL queries of the data values had unsatisfactory execution times due to the large number of tuples. As a result, a separate database was created for each execution. Note that the size of each database could not be determined due to the lack of administration support in the PostgreSQL database used. The databases share the same schema as shown in Figure 13. A utility program was created, paraImport, which populated the three databases from the text data files.

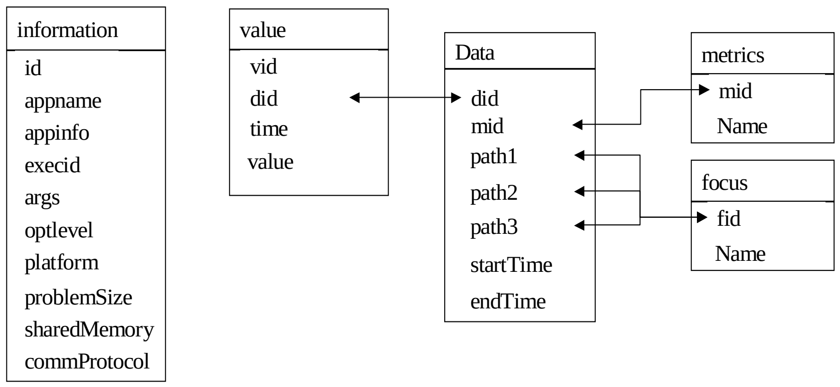


Figure 13: SMG98 Performance Data Database Schema

Shown is the database schema used to store the SMG98 performance data. The data table contains the metric-focus pairs and the start and end times. The metrics and focus tables store the metric and focus names. The value table contains the observed performance data. The information table contains information about the execution and application.

5.3 Configuration of XML to SQL

With the SMG98 databases populated, the configuration database must be configured to allow XQueries to the PostgreSQL databases. The initial step is to determine the mapping between the global XML schema and the database schema. Each level of the global XML schema's hierarchy must be mapped to an entity in the database. Figures 14 (next page) and 15 (page 43) illustrate these mappings.

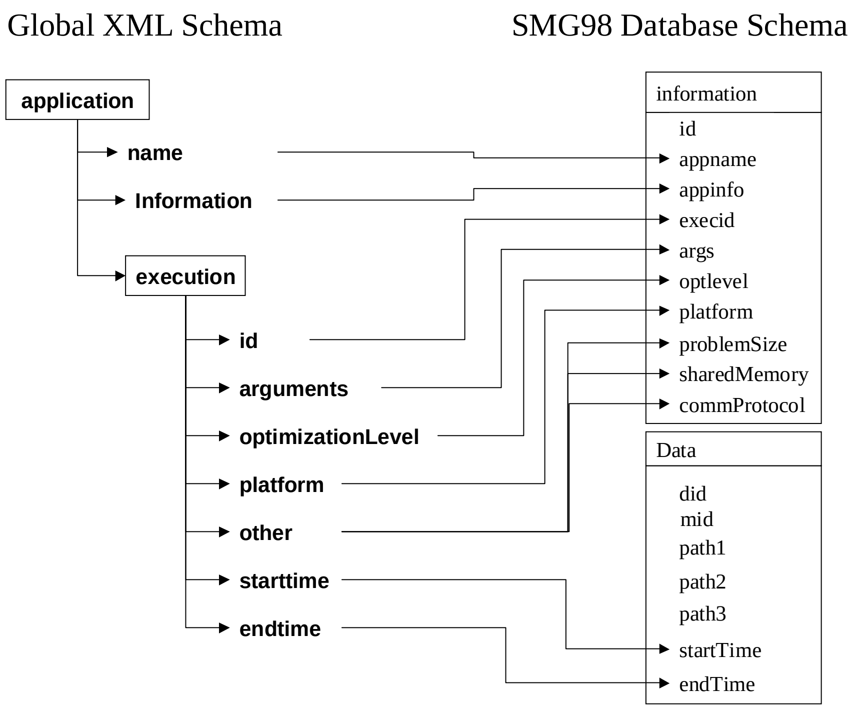


Figure 14: Mapping the Application and Execution Elements

Shown is the mapping between the application and execution elements in the global XML schema with the corresponding tables in the SMG98 database schema. Most elements map to a field in the information table with the exception of the start and end times, which map to the data table. Since all data items have the same start and end times, only one value from each is needed. Note that there are three "other" elements mapping to information specific to SMG98.

Once the mapping has been determined, the configuration database is populated. First each database is entered in the documents table with the SMG98_4 database entered as the smg98_4.xml document, SMG98_8 as smg98_8.xml, and SMG98_27 as smg98_27.xml. Next, an sql_connection entry is made for each database. Entries for the five objects in the XML schema, application, execution, metric, focus, and data, are made in the objects tables. For application, metric, and data, the entries are straightforward since each has a direct mapping to an entity, information, metrics, and value

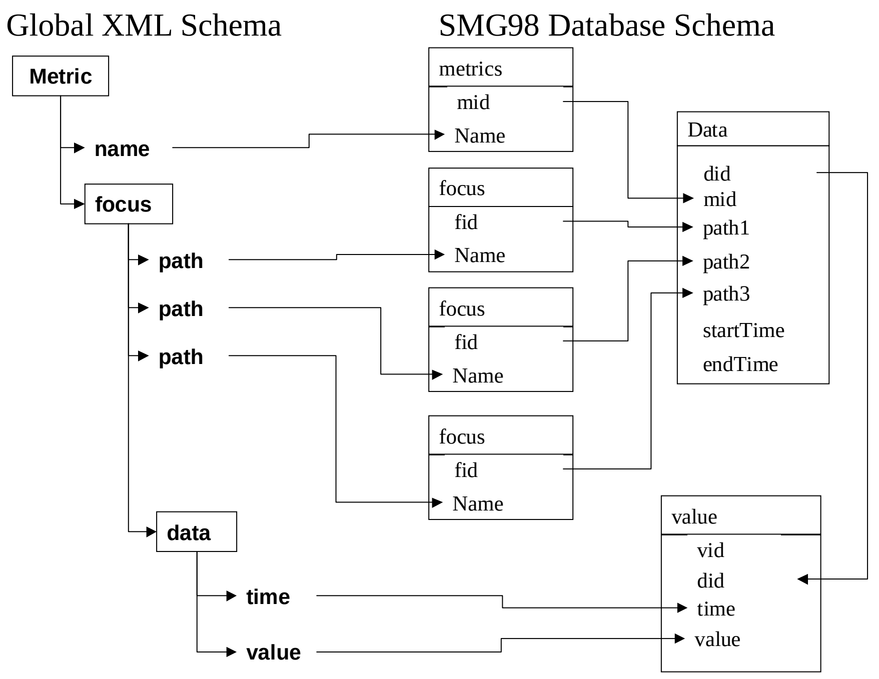


Figure 15: Mapping the Metric, Focus, and Data Elements

Shown is the mapping for the global XML schema's metric, focus, and data elements to the SMG98 database tables. The metric element directly maps to the metrics table while the focus element indirectly maps to the data table through three instances of the focus table. The data element maps directly to the value table. The combination of metrics identifier and the focus paths obtain the values comprising the selected metric-focus pair.

respectively. However, execution and focus are composites of multiple entities. Two views, execution and paths, were created in each of the SMG98 databases. Execution and focus are then mapped to the views. Each child element of the objects are entered in the child table and mapped to a specific field in the corresponding entity. Child objects are entered in the reference table and joins are added for the metric/focus and focus/data relationships. Additional children were added, i.e. the metric identifier and data identifier, to facilitate the joins. Finally, application was entered as the root object.

5.4 Use Cases

With the databases in place and the mapping configuration complete, PPerfXchange is now able to begin evaluating XQueries. This section gives a series of use cases (see Table 2) that PPerfDB may need to retrieve remote performance analysis for use in an experiment. The situation for each case is given as well as a corresponding XQuery. Each of use case's XQueries were sent to PPerfXchange and a resulting XML document retrieved. Performance measurements, such as the retrieval time, were not recorded since the test use cases are meant to only illustrate the usefulness of PPerfXchange. Once a full implementation of PPerfXchange is developed, the performance of PPerfXchange will be evaluated. Portions of the resulting XML documents are given in Appendix B.

Table 2: Use Cases

	Use	XQuery

1	To determine the remote site's published virtual XML documents, PPerfDB querys the configuration database for a list of virtual XML documents as well the corresponding global XML schema. The schema's tag is replaced with "template" and the schema element's text is inserted using the "text()" function.	`<documentList>`
		`{`
		`   FOR $x IN document("PPerfConf.xml")/documents`
		`   RETURN`
		`      <document>`
		`         { $x/name }`
		`         <template> { $x/schema/text() } </template>`
		`      </document>`
		`}`
		`</documentList>`

Table 2: Use Cases

	Use	XQuery

2	Once a list of documents is obtained, PPerfDB now determines what information is available about the applications in the "smg98_8.xml" document. While each document in the example corresponds to a single execution, a virtual XML document may span more than one execution.	`<applicationInformation>`
		`{`
		`  FOR $x IN document("smg98_8.xml")/application`
		`  RETURN`
		`    <application> { $x } </application>`
		`}`
		`</applicationInformation>`

3	Next, PPerfDB gathers a list of information about the available executions. Again, the example document has only a single execution but may span multiple runs.	`<executionInformation>`
		` {`
		`   FOR $x IN document("smg98_8.xml")/application`
		`        /execution`
		`   RETURN`
		`       <execution> { $x } </execution>`
		` }`
		`</executionInformation>`

4	The information about the available metrics is determined and the results are placed into PPerfDB's Event Map.	`<metricInformation>`
		` {`
		`   FOR $x IN document("smg98_8.xml")/application`
		`        /execution/metric`
		`   RETURN`
		`     <metric> { $x/name/text() } </metric>`
		`  }`
		`</metricInformation>`

5	Next, PPerfDB determines what foci are available for the metric "func_calls." PPerfDB now has a complete Event Map and the experiment may now begin retrieving data.	`<focusInformation>`
		` {`
		`    FOR $x IN document("smg98_8.xml")/application`
		`        /execution/metric[name="func_calls"]`
		`        /focus`
		`    RETURN`
		`       <func_calls> { $x } </func_calls>`
		`  }`
		`</focusInformation>`

Table 2: Use Cases

	Use	XQuery

6	The scientist has selected a focus from the Event Map and wishes to visualize the results. PPerfDB now retrieves all performance data for this focus from the remote site.	`<smg98_8data>`
		`  {`
		`    FOR $x IN document("smg98_8.xml")/application`
		`          /execution`
		`          /metric[name="func_calls"]`
		`          /focus[path1="/Code/MPI/MPI_Comm_size"]`
		`                [path2="/Process/4"]`
		`                [path3="/SyncObject/Communicator/0"]`
		`         /data`
		`    RETURN`
		`        <process4> { $x } </process4>`
		`  }`
		`</smg98_8data>`

7	The scientist wishes to only analyze the focus data between the time interval of 1 and 1.02 seconds. PPerfDB retrieves only a subset of data for the given focus.	`<smg98_8data>`
		`  {`
		`    FOR $x IN document("smg98_8.xml")/application`
		`          /execution`
		`          /metric[name="func_calls"]`
		`          /focus[path1="/Code/MPI/MPI_Comm_size"]`
		`                [path2="/Process/4"]`
		`                [path3="/SyncObject/Communicator/0"]`
		`         /data[time>1][time<1.02]`
		`    RETURN`
		`        <process4> { $x } </process4>`
		`  }`
		`</smg98_8data>`

Table 2: Use Cases

	Use	XQuery

8	The scientist wishes to compare the focus data from the “smg98_8.xml” document with the same focus from the “smg98_27.xml” document. Both documents are queried in succession. The results are returned as a single XML document. Note that the time intervals selected are meant to reduce the size of the resulting XML document. They are not meant to correspond to an actual performance analysis experiment.	`<smg98data>`
		`  <smg98_8data>`
		`   {`
		`       FOR $x IN document("smg98_8.xml")/application`
		`           /execution`
		`           /metric[name="func_calls"]`
		`           /focus[path1="/Code/MPI/MPI_Comm_size"]`
		`                  [path2="/Process/4"]`
		`                  [path3="/SyncObject/Communicator/0"]`
		`           /data[time>1][time<1.02]`
		`       RETURN`
		`           <process4> { $x } </process4>`
		`   }`
		`  </smg98_8data>`
		`  <smg98_27data>`
		`   {`
		`      FOR $x IN document("smg98_27.xml")/application`
		`           /execution`
		`           /metric[name="func_calls"]`
		`           /focus[path1="/Code/MPI/MPI_Comm_size"]`
		`                  [path2="/Process/4"]`
		`                  [path3="/SyncObject/Communicator/0"]`
		`           /data[time>7.7][time<7.74]`
		`      RETURN`
		`          <process4> { $x } </process4>`
		`   }`
		`  </smg98_27data>`
		`</smg98data>`

6 Related Work

The architecture and implementation of PPerfXchange presented in chapters 3 and 4 are the results of research in three major areas. Section 6.1 describes the mediator concept in which a unified process is created to access multiple databases using a global schema for semantic integration. Section 6.2 presents work related to representing XML and XML queries in relational databases. Section 6.3 describes some of the work surrounding the development of the XQuery language.

6.1 Mediators and Semantic Integration

The term mediator is used to describe a semi-autonomous module that manages partitioned, i.e. distributed, information systems. The mediator is placed between the user and the information system. The users see a single system and can make read-only queries against this unified view. The mediator receives this query and evaluates it by issuing sub-queries to particular data stores within the system. In Gio Wiederhold's 1991 paper *Mediators in the Architecture of Future Information Systems* [26], the author details an architecture that uses mediators to abstract the details of a distributed information system. The unified document hierarchy as described in section 4.3 is based upon this idea. A paper by Richard Hull, *Managing Semantic Heterogeneity in Databases: A Theoretical Perspective* [13], discusses how mediators can be used for read-only queries across data stores containing varying schemas.

Gio Wiederhold details two approaches for using mediators. The first is a materialized approach in which data from the various data stores is periodically updated into a single source. Data warehousing is an example of this approach. A second approach, which PPerfXchange adopts, is to use an integrated view, or virtual schema, across all data stores. The virtual approach has the advantage of being able to query active data stores but suffers from the complexity of needing to translate local schemas into the unified view, and the added costs associated with it. For the materialized approach, only historical data may be viewed. If large volumes of data are present, the cost of querying this large data store may outweigh the cost of mapping virtual schemas.

The concept of mediators has been studied for a variety of data stores. The paper *Object-Oriented Mediator Queries to XML Data* [16] discusses a method of using object-oriented mediator queries to retrieve data from a collection of XML documents. *Querying Heterogeneous Information Systems Using Source Descriptions* [10] describes a method of querying the world wide web (WWW) called the information manifold. The information manifold uses a “world view” to represent a virtual view across multiple WWW documents.

The concepts illustrated by these and other papers are presently used in commercial database systems. One such application, Nimble [7], extensively uses XML to achieve a unified model of the underlying data stores. Nimble uses the global-as-view approach [11] in which XML queries are definitions for how to query the local data stores.

While Nimble and PPerfXchange have similar architectures, PPerfXchange was developed without prior knowledge of Nimble. Nimble was released in February 2002 after PPerfXchange was developed and implemented. However, the co-founder of Nimble, Alon Halevy, has authored several papers on data integration including *Logic-Based Techniques in Data Integration* [11].

6.2 Representing and Querying a Relational Database Using XML

As the development of the PPerfXchange prototype began, the W3C published their working draft for the XQuery language [28]. While little work prior to this point had been done in the translation of XQuery to SQL, work had been done on related XML Query languages. In particular, *Pushing XML Queries Inside Relational Database* [17] by Ioana Manolescu, Daniela Florescu, and Donald Kossmann used the Quilt XML query language to show how such queries could be translated into SQL. The authors work uses a “local as view” (LAV) approach, purposed by Alon Halevy [11] in which the contents of a data store are described in terms of the global schema. The query is reformulated in terms of the local data store, executed, and then translated it back into the global schema. Manolescu, Florescu, and Kossmann define a set of relations that correspond to the structure of a virtual generic XML Schema. The local relations are then described as a virtual XML document based upon the virtual generic schema. The XML query is then normalized, translated into an SQL query, and evaluated by the relational database.

A second approach that influenced PPerfXchange’s design is given by SilkRoute [9]. SilkRoute offers two methods to represent an XML query as a relational database query. The first is a virtual view in which an XML query, in this case XML-QL, is applied against virtual XML views representing entities within the relational database. Each query is translated into SQL via an intermediately query language called “Relational to XML Transformation Language” (RXL). However, the translation process is complex and may create a slightly differing result then the intended query. The second method uses a materialized XML document for the entire database. While this means the query is not acting on live data, no translation step is needed.

For PPerfXchange, some of the ideas expressed above have been implemented in the PPerfXchange prototype. However, aspects have been modified to better solve the objectives of PPerfXchange. One area that both solutions do not address is how other data stores can be queried and how virtual XML documents from multiple data stores can be joined. If the prototype followed the approach given by *Pushing XML Queries Inside Relational Database* [17], each class of data store, i.e. text, XML, object-oriented databases, would need its own SQL-like query mechanism and a translation process from an XML Query to this mechanism. Also, if multiple documents are used in a single query, and each document maps to a different data store, then an additional higher-order process will need to divide the query across the data stores and then join the resulting data sets. For native XML documents, XML documents written in the global schema, a complete XML query processor would need to be developed, passing the result set to a higher level XML query processor for additional processing. A second unsolved issue is that several features of a XML query are difficult, if not impossible, to translate into SQL. One example is when a query needs to be materialized for intermediate XML results. A second is support for user-defined functions.

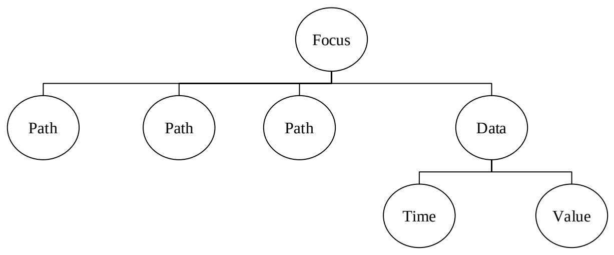
The approach the PPerfXchange prototype uses is to push only a small part of XQuery into the relational database, leaving the bulk of the processing to be done in the XQuery processor itself. Only the XPath portion of an XQuery is used by a virtual XML document to configure the mapping to the actual data set. The virtual XML document represents the entire object defined by the XPath, including children. The XQuery processor then processes the virtual XML documents as if they were native XML documents. While virtual XML documents would most likely be less efficient than an SQL processor and return larger than needed data sets for processing, this penalty is mitigated by benefit of applying a uniform process to all data stores.

In *The Table and the Tree: On-Line Access to Relational Data Through Virtual XML Documents* [2], the authors describe the ROLEX (Relational On-Line Exchange with XML) system architecture. ROLEX is a SAX and DOM interface to a relational database, specifically DataBlitz, in which relational data may be published as an XML document for use with a web server. DOM stands for “Document Object Model” and creates an in-memory tree data structure of an XML document. DOM allows for parsing of the XML document and retrieval of specific elements within the document tree.

SAX is an event driven XML document parser for use with large XML documents or used to model XML documents in a different object model than DOM. The authors determined that the techniques for translating XML queries to SQL as shown in SilkRoute [8,9] were too costly for the high demand of a web-server. Instead, they provide the interface for extracting relational data as an XML document and lets the XML query processor or web server handle any additional processing. PPerfXchange adopted this idea of only pushing part of the XML query into the relational database but stopped short of creating a SAX and DOM interface.

The paper *Efficiently Publishing Relational Data as XML Documents* [22] describes several methods for representing relational data as an XML document. This paper was not found until after PPerfXchange was designed and the prototype implemented. It is unfortunate because several of the methods discussed could have been implemented and might have greatly improved the performance of the relational database to virtual XML document translation process. The approach used in PPerfXchange is described by the authors as the “stored-procedure approach”. This approach uses nested queries to model the hierarchical XML model. For documents with large hierarchies, the nested queries result in a considerable performance cost. Other approaches are the Redundant Relation and Outer Union. Both join all relations into a single view. Markers keep track of the movement within the hierarchy, either in a column or row, and data is tagged appropriately. Figure 16 (next page) illustrates this approach.

The differences between the two is that the Redundant Relation approach uses left-outer joins to combine the relation while the Outer Union approach uses a combination of right and left outer joins. The Redundant Relation approach will return a tuple for all leaf nodes, even if that leaf node contains no data. The outer union approach returns only populated tuples thus reducing the result size and decreasing the amount of processing needed.



Path	Path	Path	Time	Value
/Code	/Process	/SyncObject	1	10
/Code	/Process	/SyncObject	2	5
/Code	/Process	/SyncObject	3	6
/Code	/Process	/SyncObject	4	3
/Code/MPI	/Process	/SyncObject	1	5
/Code/MPI	/Process	/SyncObject	2	2
/Code/MPI	/Process	/SyncObject	3	1
/Code/MPI	/Process	/SyncObject	4	3

Figure 16: Alternative XML Representation Approaches

Shown is how the Redundant Relation and Outer Union approaches would view an XML document in SQL.

6.3 XQuery

The direction of XQuery's development was heavily influenced by the document *Database Desiderata for an XML Query Language* written by Dr. David Maier [18]. The document outlines Dr. Maier's desired characteristics for XQuery. This document pushed the development of XQuery toward the “XML as data” instead of the “XML as document” [14]. This facilitated XQuery’s support for relational query operations and made it easier to translate XQuery syntax to SQL.

At the time the development of PPerfXchange began, XQuery 1.0 [28] had just been released by the W3C. At the time only two XQuery processor implementations had been created, one by Microsoft and one by FatDog Software. Both are commercial products and unavailable for use with PPerfXchange. This condition lead to the development of PPerfXchange’s XQuery processor. Nearly nine months later, several more implementations have become available. Notably, Galax [24] is in development at Bell labs and is currently in alpha release. Galax’s goal is to fully implement the entire XQuery family [27-32] as an open source application. The current alpha release is used as a module to Bell Labs DataBlitz database.

7 Conclusions and Future Work

This thesis has detailed the PPerfXchange approach for retrieval of parallel performance data from geographically dispersed, heterogeneous data stores. PPerfXchange uses global XML schemas to describe a common format for parallel performance data. For each set of data a site wishes to publish, a virtual XML document is created. A virtual XML document maps the published data set's schema to an XML document based on the global XML schema. PPerfXchange uses a configuration database to store information about this mapping. Once a set of virtual XML documents is published by a site, other sites may query these documents using the XML query language XQuery. Since the amount of parallel performance data is often large, the use of XQuery allows a user to retrieve a specific subset of performance data from the larger published set. XQuery also allows the user to define the format of the resulting data set for easier integration with the user's local format.

Chapter 5 gives an example of how PPerfXchange is used to retrieve parallel performance data from a remote data store. The data store used was a PostgreSQL object-relational database. The chapter shows how the global XML schema is mapped to the PostgreSQL database schema and gives several example use cases. The use cases illustrate the XQuery expressions, path expressions, element constructors, and FOR expressions, necessary to retrieve data from a relational database. Future versions of PPerfXchange should add the ability to join two or more documents, apply aggregate functions, and sort documents. Other XQuery expressions would only need to be implemented in order to fully support XQuery but are not necessary for the evaluation of the PPerfXchange approach. These unsupported expressions are: data types, most built-in functions, user defined functions, process instructions, references, arithmetic operation, comparison operations, logical operations, sequence-related operations, and conditional expressions. Future versions of PPerfXchange may consider integrating an open-source XQuery processor, such as GALAX [15], instead of completing the current XQuery processor.

The unified XML document hierarchy described in chapter 3, shows how multiple virtual XML documents can be combined into a single view. While each virtual XML document may map to a single data store, unified XML documents allow an integrated view of multiple data stores. Completion of the unified XML document is not absolutely necessary. As shown in the use cases, performance data can be retrieved without it. However, a unified XML document would allow for the abstracting of the SMG98 databases, each containing a single execution’s performance data, into a single view over all executions. This allows for simpler and more uniform queries and reduces the amount of site specific information needed by users to form queries.

Additional virtual XML document classes and connection classes are needed to support a wider variety of data models and data stores, including native XML documents and other structured text files. The supported structured text files would be those generated from parallel performance analysis tools such as the text files used to create the SMG98 database described in chapter 5. Since virtual XML documents map a specific data store's structure to the global XML schema, each type of structured text file may require a unique virtual XML document. Also needed is the creation of a schema-to-schema mapping tool to simplify the publishing of a data set as a virtual XML document.

Since each site's performance data is proprietary, future versions of PPerfXchange should give administrators the ability to authenticate and log XQuery requests. An administrator should be able to grant or deny access to each particular published data set depending upon privileges of the person sending the query. Also, logs should be kept of all requests for administrative information. The use of encryption for sending the XQuery and the resulting XML document is encouraged in future versions.

While the performance of PPerfXchange was not measured nor optimized, several performance issues should be addressed in future versions. The main performance issue is the time to transfer data. While additional processing by the XQuery processor may reduce the amount of data in some cases, the volume of data is still assumed to be large. Currently, the PPerfXchange prototype uses TCP to transport data across the network connection. While TCP is reliable, the use of UDP should be researched since UDP has less overhead than TCP and can be more efficient for bulk data transfer. A second performance issue is the time to process an XQuery. Specifically, the relational database virtual XML document class can be modified to use a more efficient method of modeling a relational database schema. The PPerfXchange prototype uses the "stored-procedure approach" but should investigate the use of the Outer Union approach [22] as described in section 6.2. Also, a reduction in the number of times the data is copied within the XQuery processor would increase efficiency. The virtual XML document first stores the data returned from the PostgreSQL database in a temporary data structure. A second copy is made when forming the resulting XML document and a final copy is created when the resulting XML document is buffered for network transport. The final two copies could be eliminated if the libxml2 library [33] was not used and if PPerfXchange contained its own method of creating and transferring the resulting XML document.

Although the prototype of PPerfXchange is not complete, it does allow for the evaluation of the PPerfXchange approach. The overall goal of PPerfXchange is to allow geographically dispersed collaborating scientists, specifically those scientists optimizing applications for use on a parallel architecture, to easily exchange heterogeneous data. To accomplish this goal, PPerfXchange needs to overcome three main obstacles. The first obstacle is the semantic integration of the heterogeneous data stores. To solve this, PPerfXchange uses a global XML schema. Each site maps their data's format and semantics to the common format. A site needs to only translate their own data into the global XML schema but does not need to have knowledge of the specific formats used by other sites. The second obstacle is each site may use different methods to store the data. Some sites may use a relational database, while others may use text files. In order to retrieve the data, the query writer should not need to know what data store is being used by the remote site. To solve this, PPerfXchange uses virtual XML document classes to represent various data models and connection classes to connect to a specific data store. A virtual XML document maps a specific data store's schema to the global XML schema and retrieves the data. The final obstacle is the volume of data that can be generated by parallel performance analysis tools. Since a single execution can generate hundreds of megabytes of data, the time to transfer the entire data set would be lengthy. To limit the amount data retrieved to the specific data set needed to aid in a particular performance analysis, PPerfXchange allows for the querying of remote performance data using XQuery. Also, an XQuery defines a resulting XML document's format allowing for easier integration at the local site.

8 References

[1] Fabio Arciniegas. *C++ XML*. Indianapolis, IN: New Riders Publishing, 2002.

[2] P. Bohannon, H.F. Korth, P.P.S. Narayan. *The Table and the Tree: On-Line Access to Relational Data Through Virtual XML Documents.* Bell Laboratories, Murry Hill, NJ. Proceedings of the WebDB 2001 Workshop on Databases and the Web. May 2001. Santa Barbara, CA

[3] Ronald Bourret. *XML and Databases*. http://www.rpbourret.com/xml/XMLAndDatabases.htm. February, 2002

[4] Rajkumar Buyya. *High Performance Cluster Computing: Volume 1 Architectures and Systems*. Prentice Hall, New Jersey, 1999. Pages 33-34.

[5] Justin Campbell, Daniel Grossman, Ana-Maria Popescu. *Quilt2Sql - An ML Storage Schema and Query Engine for the Quilt Query Language*. University of Washington CSE 544 Term Paper. http://www.cs.washington.edu/homes/grossman/projects/544project June 5, 2000.

[6] Charles Donnelly, Richard Stallman. *Bison: The YACC-compatible Parser* Generator. November 1995. Bison version 1.25 http://www.gnu.org/manual/bison-1.25/html_node/bison_toc.html

[7] Denise Draper, Alon Y. Halevy, Daniel S. Weld. *The Nimble XML Data Integration System*. Proceedings of ACM SIGMOD Conference on Management of Data 2001. http://www.cs.washington.edu/homes/alon/site/files/sigmod01-nimble.ps

[8] Mary Fernandez, Atsuyuki Morishima, Dan Suciu. *Efficient Evaluation of XML Middle-ware Queries*. 2001 SIGMOD Conference http://www.research.att.com/~mff/files/final.pdf

[9] Mary Fernandez, Atsuyuki Morishima, Dan Suciu, Wang-Chiew Tan. *Pushing Relational Data in XML: the SilkRoute Approach*. IEEE Data Engineering Bulletin , no. 24(2) , pp. 12--19 , 2001 http://www.research.att.com/~mff/files/_F292063957.pdf

[10] Alon Y. Halevy, Anand Rajaraman, Joann J. Ordille. *Querying Heterogeneous Information Sources Using Source Descriptions.* Proceedings of the 22nd VLDB Conference, Mumbai (Bombay), India, 1996. http://dbpubs.stanford.edu:8090/pub/1996-61

[11] Alon Y. Halevy. *Logic-Based Techniques in Data Integration.* University of Washington, May 1999 http://citeseer.nj.nec.com/391746.html

[12] Christian Hansen. *Towards Comparative Profiling of Parallel Applications with PPerfDB.* Portland State University Master's Thesis. October, 2001.

[13] Richard Hull. *Managing Semantic Heterogeneity in Databases: A Theoretical Perspective.* Proceedings ACM Symposium on Principles of Databases (Invited Tutorial) (1997), pp. 51--61. http://www-db-out.bell-labs.com/user/hull/pods97-tutorial.html

[14] Howard Katz. *An Introduction to XQuery.* http://www-106.ibm.com/developerworks/xml/library/x-xquery.html.

[15] John R. Levine, Tony Mason, Doug Brown. *lex & yacc.* (second edition) Sebastopol, CA: O'Reilly & Associates, Inc., 1992

[16] Hui Lin, Tore Risch, Timour Katchaounov. *Object-Oriented Mediator Queries to XML Data*. Proceedings of 1st International Conference on Web Information Systems Engineering, (Vol 2), Hong Kong, China, June 2000, pp 38-45. http://www.dis.uu.se/~udbl/publ/hui_xml.pdf

[17] Ioana Manolescu, Daniela Florescu, Donald Kossmann. *Pushing XML Queries Inside Relational Databases.* INRIA Domaine de Voluceau-Rocquencourt, Le Chesnay Cedex, France. http://www.inria.fr/rrrt/rr-4112.html. January 2001.

[18] David Maier. *Database Desiderata for an XML Query Language.* Query Languages 98. http://www.w3.org/TandS/QL/QL98/pp/maier.html

[19] *PostgreSQL 7.1 Documentation.* http://www.postgresql.org/idocs/ 2001.

[20] Erik T. Ray. *Learning XML*. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.

[21] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, Jeffrey Naughton. *Relational Database for Querying XML Documents: Limitations and Opportunities*. Proceedings of the 25th Very Large Database Conference, Edinburgh, Scotland, 1999.

[22] Jayavel Shanmugasundaram, Eugene Shekita, Rimon Barr, Micheal Carey, Bruce Lindsay, Hamid Pirahesh, Berthold Reinwald. *Efficiently Publishing Relational Data as XML Documents.* IBM Almaden Research Center, San Jose, CA Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000.

[23] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, I. Tatarinov. *A General Technique for Querying XML Documents Using a Relational Database System*. SIGMOD Record, September 2001

[24] Jerome Simeon. *GALAX*. Bell Labs, Lucent Technologies. http://db.bell_labs.com/galax

[25] Karli Watson, Brian Smith, Darshan Singh, Denise Gosnell, Carvin Wilson, Sam Ferguson, Warren Wiltsie, Paul Morris, Jan Narkiewicz, Jon Reid, Paul J Burke, J. Michael Palermo IV. *Professional SQL Server 2000 XML.* Appendix B pages 527-577. Wrox Press Ltd. Birmingham, UK. 2001.

[26] Gio Wiederhold. *Mediators in the Architecture of Future Information Systems*. IEEE Computer Magazine, March 1992. http://www-db.stanford.edu/LIC/mediator.html

[27] *XML Schema. Part 0: Primer, Part 1: Structures, Part 2: Data Types.* W3C Recommendation. May 2001 http://www.w3.org/XML/Schema#dev

[28] *XQuery 1.0: An XML Query Language.* W3C Working Draft. June 2001. http://www.w3.org/TR/xquery/

[29] *XQuery 1.0 Formal Semantics.* W3C Working Draft. June 2001 http://www.w3.org/TR/query-semantics/

[30] *XQuery 1.0 and XPath 2.0 Data Model.* W3C Working Draft June 2001. http://www.w3.org/TR/query-datamodel/

[31] *XML Query Use Cases.* W3C Working Draft. June 2001 http://www.w3.org/TR/xmlquery-use-cases

[32] *XML Syntax for XQuery 1.0 (XQueryX)*. W3C Working Draft June 2001. http://www.w3.org/TR/xqueryx

[33] *The XML C Library for Gnome*. http://www.xmlsoft.org

[34] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, Shunsuke Uemura. *XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases.* ACM TOIT, 1(1), 2001. http://db-www.aist-nara.ac.jp/members/Yoshikawa/paper/TOIT2001-authorCopy.pdf

Appendix A: Example Global XML Schema for Parallel Performance Data

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 (http://www.xmlspy.com)
     by Mat Colgrove  -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
attributeFormDefault="unqualified">
   <xs:element name="Application">
      <xs:complexType>
         <xs:complexContent>
            <xs:extension base="ApplicationType"/>
         </xs:complexContent>
      </xs:complexType>
   </xs:element>
   <xs:element name="metric" type="metricType"/>
   <xs:complexType name="ApplicationType">
      <xs:sequence>
         <xs:element name="Name" type="xs:string"/>
         <xs:element ref="execution" maxOccurs="unbounded"/>
         <xs:element name="Information" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="name" type="xs:string"/>
   <xs:complexType name="metricType">
      <xs:sequence>
         <xs:element name="name"/>
         <xs:element ref="focus" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="execution" type="executionType"/>
   <xs:complexType name="executionType">
      <xs:sequence>
         <xs:element name="id">
            <xs:simpleType>
               <xs:restriction base="xs:integer">
                  <xs:whiteSpace value="preserve"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:element>
         <xs:element name="arguments" type="xs:string"/>
         <xs:element name="optimizationLevel" type="xs:integer"/>
         <xs:element name="platform" type="xs:string"/>
         <xs:element ref="metric" maxOccurs="unbounded"/>
         <xs:element name="other" maxOccurs="unbounded"/>
         <xs:element name="startTime" type="xs:float"/>
         <xs:element name="endTime" type="xs:float"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="focus" type="focusType"/>
   <xs:element name="data" type="dataType"/>
   <xs:complexType name="dataType">
      <xs:sequence>
         <xs:element name="value" type="xs:double"/>
         <xs:element name="time" type="xs:long"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="focusType">
      <xs:sequence>
         <xs:element name="path" type="xs:string"
                     maxOccurs="unbounded"/>
         <xs:element ref="data" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Appendix B: Resulting XML Documents from the Use Cases

Given is the resulting XML documents, or portions of the resulting XML documents, for the corresponding use case. The use case is given along with the size of the resulting XML document.

Table 3: Resulting XML Documents

Use Case	Resulting XML Document

1) What documents are available at the remote site and what schemas do they use?
 	
Size: 1 KB
	<?xml version="1.0" encoding="UTF-8"?>
	<!-- PPerfXchange generated document from test site.
	    Contact: Mathew Colgrove Email: colgrove@cs.pdx.edu
	    Query id ="1893159416-1026" -->
	<documentList>
	  <document>
	    <name>PPerfConf.xml</name>
	    <schema> pperfconfig.xsd</schema>
	  </document>
	  <document>
	    <name>smg98_4.xml</name>
	    <schema>paradata.xsd</schema>
	  </document>
	  <document>
	    <name>smg98_8.xml</name>
	    <schema>paradata.xsd</schema>
	  </document>
	  <document>
	    <name>smg98_27.xml</name>
	    <schema>paradata.xsd</schema>
	  </document>
	</documentList>

2) What is the application information for SMG98_8.xml? Size: 1 KB	<?xml version="1.0" encoding="UTF-8"?> <!-- PPerfXchange generated document from test site. Contact: Mathew Colgrove Email: colgrove@cs.pdx.edu Query id ="1968639724-1026" --> <applicationInformation> <application> <name>smg98</name> <information>Data gathered by Christian Hansen for his thesis</ information> </application> </applicationInformation>

Table 3: Resulting XML Documents

Use Case	Resulting XML Document
3) What is the execution information for SMG98_8.xml? Size: 1 KB	<?xml version="1.0" encoding="UTF-8"?> <!-- PPerfXchange generated document from test site. Contact: Mathew Colgrove Email: colgrove@cs.pdx.edu Query id ="1929940265-1026" --> <executionInformation> <execution id="smg_8"> <optimizationlevel>blue</optimizationlevel> <platform>40x40x40</platform> <sharedMemory>yes</sharedMemory> <commProtocol>ip</commProtocol> <starttime>0</starttime> <endtime>5.600076</endtime> <arguments>basic</arguments> </execution> </executionInformation>
4) What metrics are available for the SMG98_8 execution? Size: 1 KB	<?xml version="1.0" encoding="UTF-8"?> <!-- PPerfXchange generated document from test site. Contact: Mathew Colgrove Email: colgrove@cs.pdx.edu Query id ="1633500571-1026" --> <metricInformation> <metric>func_calls</metric> <metric>msg_bytes</metric> <metric>func_duration</metric> <metric>msg_deliv_time</metric> </metricInformation>

Table 3: Resulting XML Documents

Use Case	Resulting XML Document
5) What foci are available for the metric “func_calls” and execution SMG98_8?	`<?xml version="1.0" encoding="UTF-8"?>`
	`<!-- PPerfXchange generated document from test site.`
	`    Contact: Mathew Colgrove Email: colgrove@cs.pdx.edu`
Size: 51 KB	`    Query id ="2097126748-1026" -->`
	`<focusInformation>`
	`  <func_calls>`
	`    <focus>`
	`      <path1>/Code/MPI</path1>`
	`      <path2>/Process</path2>`
	`      <path3>/SyncObject</path3>`
	`    </focus>`
	`  </func_calls>`
	`  <func_calls>`
	`    <focus>`
	`      <path1>/Code/MPI/MPI_Allgather</path1>`
	`      <path2>/Process</path2>`
	`      <path3>/SyncObject</path3>`
	`    </focus>`
	`  </func_calls>`
	`  <func_calls>`
	`    <focus>`
	`      <path1>/Code/MPI/MPI_Allgather</path1>`
	`      <path2>/Process/1</path2>`
	`      <path3>/SyncObject</path3>`
	`    </focus>`
	Results continue.

Table 3: Resulting XML Documents

Use Case	Resulting XML Document

6) Return all the data from SMG98_8.xml given the metric “func_calls” and the focus “/Code/MPI/ MPI_Comm_size, /Process/4, /SyncOb-ject/Communicator/0.”
 	
Size: 400 KB
	Header Omitted
	  <data>
	    <value>10</value>
	    <time>0.456719</time>
	  </data>
	 </process4>
	 <process4>
	  <data>
	    <value>11</value>
	    <time>0.457316</time>
	  </data>
	 </process4>
	 <process4>
	  <data>
	    <value>12</value>
	    <time>0.457363</time>
	  </data>
	 </process4>
	 <process4>
	  <data>
	    <value>13</value>
	    <time>0.467556</time>
	  </data>
	Results continue

Table 3: Resulting XML Documents

Use Case	Resulting XML Document

7) Return all the data between 1 and 1.02 seconds from SMG98_8.xml given the metric “func_calls” and the focus “/Code/MPI/MPI_Comm_size, /Process/4, /SyncObject/Communicator/0.”
 	
Size: 4 KB
	Header omitted
	  <data>
	    <value>611</value>
	    <time>1.007649</time>
	  </data>
	 </process4>
	 <process4>
	  <data>
	    <value>612</value>
	    <time>1.007709</time>
	  </data>
	 </process4>
	 <process4>
	  <data>
	    <value>613</value>
	    <time>1.00853</time>
	  </data>
	 </process4>
	 <process4>
	  <data>
	  <value>614</value>
	  <time>1.008592</time>
	  </data>
	Results continue

Table 3: Resulting XML Documents

Use Case	Resulting XML Document
8) Return all the data between 1 and 1.02 seconds from SMG98_8.xml and the data between 7.7 and 7.74 seconds from the SMG98_27.xml given the metric “func_calls” and the focus “/Code/ MPI/MPI_Comm_size, /Process/4, /SyncOb- ject/Communicator/0”. Size: 5 KB	`<?xml version="1.0" encoding="UTF-8"?>` `<!-- PPerfXchange generated document from test site.` `Contact: Mathew Colgrove Email: colgrove@cs.pdx.edu` `Query id ="929202754-1026" -->` `<smg98data>` `<smg98_8data>` `<process4>` `<data>` `<value>609</value>` `<time>1.006873</time>` `</data>` `</process4>` `<process4>` `<data>` `<value>610</value>` `<time>1.006954</time>` `</data>` `</process4>` `...` `</smg98_8data>` `<smg98_27data>` `<process4>` `<data>` `<value>5</value>` `<time>7.729737</time>` `</data>` `</process4>` `<process4>` `<data>` `<value>6</value>` `<time>7.729837</time>` `</data>` `</process4>` `<process4>` `<data>` `<value>7</value>` `<time>7.73039</time>` `</data>` Results continue