# Supporting High-level Abstractions through XML Technology

Xiaogang Li   Gagan Agrawal

Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{xgli,agrawal}@cis.ohio-state.edu

**Abstract.** Development of applications that process large scientific datasets is often complicated by complex and specialized data storage formats. In this paper, we describe the use of XML technologies for supporting high-level programming methodologies for processing scientific datasets. We show how XML Schemas can be used to give a high-level abstraction of a dataset to an application developer. A corresponding *low-level* Schema describes the actual layout of data and is used by the compiler for code generation. The compiler needs a systematic way for translating the high-level code to a low-level code. Then, it needs to transform the generated low-level code to achieve high locality and efficient execution. This paper describes our approach to these two problems. By using Active Data Repository as the underlying runtime system, we offer an XML based front-end for storing, retrieving, and processing flat-file based scientific datasets in a cluster environment.

## 1 Introduction

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

Development of applications that process large scientific datasets is often complicated by complex and specialized data storage formats. When the datsets are disk-residents, understanding the layout and maintaining high locality in accessing them is crucial for obtaining a reasonable performance. While the traditional relational database technology supports high-level abstractions and standard interfaces, it is suitable more for storing and retrieving datasets, and not for complex analyses on such datasets [12].

Recently, there has been a lot of interest in XML and other related technologies developed by the W3C consortium [5]. XML is a flexible exchange format that can represent many classes of data, including structured documents, heterogeneous and semi-structured records, data from scientific experiments and simulations, and digitized images. One of the key features of XML is XML Schemas, which serve as a standard basis for describing the contents and structure of a dataset.

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

Two key compiler techniques are required for supporting such an approach. First, we need a systematic way to translate the high-level code to the low-level code. Second, we need to transform the generated low-level code to achieve high locality and efficient execution. This paper describes our approach to these two problems. Our techniques have been implemented in a compilation system. By using Active Data Repository [6, 7] as the underlying runtime system, we offer an XML based front-end for storing, retrieving, and processing flat-file based scientific datasets in a cluster environment.

As part of our system, we use the XML query language XQuery [4] for writing queries using high-level abstractions. XQuery is derived from declarative, database, as well as functional languages. Though XQuery significantly simplifies the specification of processing, compiling it to achieve efficient execution involves a number of new challenges. Our recent related paper has addressed two key issues, i.e, replacing recursive reductions by iterative constructs and type-inferencing to translate from XQuery to an imperative language [11].

## 2 Background: XML, XML Schemas, and XQuery

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

### 2.1 XML and XML Schemas

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

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

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

Applications that operate on XML data often need guarantees on the structure and content of data. XML Schema proposals [2, 3] give facilities for describing the structure and constraining the contents of XML documents. The example in Figure (a) shows an XML document containing records of students. The XML Schema describing the XML document is shown in Figure (b). For each student tuple in the XML file, it contains two string elements to specify the last and first names, one date element to specify the date of birth, and one element of float type for the student's GPA.

### 2.2 XML Query Language: XQuery

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

XQuery is a functional language. The basic building block is an *expression*. Several types of expressions are possible. The two types of expressions important for our discussion are:

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

```
< student >
  < firstname > Darin < / firstname >
  < lastname > Sundstrom < /lastname >
  <DOB > 1974-01-06 < / DOB >
  < GPA > 3.73 < / GPA >
< / student >
...
```

*(a) XML example*

```
    Schema Declaration
< xs:element name="student" >
  < xs:complexType >
    < xs:sequence >
        < xs:element name="lastname" type="xs:string"/ >
        < xs:element name="firstname" type="xs:string"/ >
        < xs:element name="DOB" type="xs:date"/¿
        < xs:element name= "GPA" type="xs:float"/ >
    < /xs:sequence >
  < /xs:complexType >
< /xs:element >
```

*(b) XML Schema*

**Fig. 1.** XML and XML Schema

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

We illustrate the XQuery language and the *for*, *let*, *where*, and *return* expressions by an example, shown in Figure 2. In this example, two XML documents, *depts.xml* and *emps.xml* are processed to create a new document, which lists all departments with ten or more employees, and also lists the average salary of employees in each such department.

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

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

A *where* clause serves as a filter for the tuples of variable bindings generated by the *for* and *let* clauses. The expression is evaluated once for each of these tuples. If the resulting value is true,

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

**Fig. 2.** An Example Illustrating XQuery's FLWR Expressions

the tuple is retained, otherwise, it is discarded. A *return* clause is used to create an XML record after processing one iteration of the *for* loop. The details of the syntax are not important for our presentation.

To illustrate the use of *unordered*, a modification of the example in Figure 2 is presented in Figure 3. By enclosing the *for* loop inside the *unordered* expression, we are not enforcing any order on the execution of the iterations in the *for* loop, and in generation of the results. Without the use of *unordered*, the departments need to be processed in the order in which they occur in the document *depts.xml*. However, when *unordered* is used, the system is allowed to choose the order in which they are processed, or even process the query in parallel.

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

**Fig. 3.** An Example Using XQuery's Unordered Expression

## 3 High-level and Low-level Schemas and XQuery Representation

This section focuses on the interface for the system. We use two motivating examples, *satellite data processing* [7] and the *multi-grid virtual microscope* [1], for describing the notion of high-level and low-level schemas and XQuery representation of the processing.

### 3.1 Satellite Data Processing

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

**Fig. 4.** High-Level XML Schema for Satellite

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

The XML Schema shown in Figure 4 provides a high-level abstraction of the satellite data. The pixels captured by the satellite can be viewed as a sparse three dimensional array, where time, latitude, and longitude are the three dimensions. Pixels for several, but not all, time values are available for any given latitude and longitude. Each pixel has 5 short integers to specify the sensor data. Also, latitude, longitude, and time is stored within each pixel. With this high-level XML Schema, a programmer can easily define computations processing the satellite data using XQuery.

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

XQuery specification of such processing is shown in Figure 5. The code iterates over the two-dimensional space for which the output is desired. Since the order in which the points are processed is not important, we use the directive *unordered*. Within an iteration of the nested for loop, the *let* statement is used to create a sequence of all pixels that correspond to the those spatial coordinates. The desired result involves finding the pixel with the best NDVI value. In XQuery, such reduction can only be computed recursively.

### 3.2 Multi-Grid Virtual Microscope

The Virtual Microscope [8] is an application to support the need to interactively view and process digitized data arising from tissue specimens. The raw data for such a system is captured by digitally

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

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

**Fig. 5.** Satellite Data Processing Expressed in XQuery

scanning collections of full microscope slides at high power. In a typical dataset available when a virtual microscope is used in a distributed setting, the same portion of a slide may be available at different resolution levels, but the entire slide is not available at all resolution levels.

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

The digitized microscope slides can also be viewed as a three dimensional dataset. Each pixel has x and y coordinates and the resolution is the third dimension. The high-level XML Schema of virtual microscope is shown in Figure 6. For each pixel in a slide, three short integers are used to represent the RGB colors.

XQuery code for performing the computations is shown in Figure 7. We assume that the user is only interested in viewing the image at the highest possible resolution level, which means that averaging is never done to produce the output image. The structure of this code is quite similar to our previous example. Inside an unordered for loop, we use the *let* statement to compute a sequence, and then apply a recursive reduction.

```
< xs:element name="pixel" maxOccurs="unbounded" >
  < xs:complexType >
    < xs:sequence >
      < xs:element name="x" type="xs:integer"/ >
      < xs:element name="y" type="xs:integer"/ >
      < xs:element name="scale" type="xs:short"/ >
      < xs:element name="color1" type="xs:short"/ >
      < xs:element name="color2" type="xs:short"/ >
      < xs:element name="color2" type="xs:short"/ >
    < /xs:sequence >
  < /xs:complexType >
< /xs:element >
```

**Fig. 6.** High-Level XML Schema for Virtual Microscope

### 3.3 Low Level XML Schema and XQuery

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

In our system, XML files are mapped to flat files by a special mapping service. Pixels in each flat file are later partitioned and organized into chunks by data distribution and indexing services. A low-level XML Schema file is provided to the compiler after partitioning of the datasets to specify the actual data layout. Here, the pixels are divided into chunks. Each chunk is associated with a bounding box for all pixels it contains, which is specified by a lower bound and a higher bound. Within a chunk, the values of pixels are stored consecutively, with each pixel occupying three bytes for RGB colors.

For each application whose XML data is transformed into ADR dataset by data distribution and indexing services, we provide several library functions written in XQuery to perform data retrieval. These library functions have a common name, *getData*, but the function parameters are different. Each *getData* function implements a unique selection operation based on its parameters. The *getData* functions are similar to physical operators of a SQL query engine. A physical operator of SQL engine takes as input one or more data streams and produces an output data stream. In our case, the default input data stream of a *getData* function is the entire dataset, while the output data stream is result of filtering the input stream by parameters of the *getData* function. For example, the *getData* function shown in Figure 8 (a) returns pixels whose $x$ and $y$ coordinates are equal to those specified by the parameters. The detailed implementation is based on the meta-data of the dataset, which is specified by the low-level XML Schemas. The *getData* function in Figure 8 (b) requires only one parameter, which retrieves pixels with specified $x$ coordinate. For space reason, the detailed implementation of only one *getData* function is shown here.

The XQuery code for virtual microscope that calls a *getData* function is shown in Figure 9. This query code is called *low-level XQuery* and is typically generated automatically by our compiler. The XQuery codes described in the above section operate on high-level data abstractions and are called *high-level XQuery*. The recursive functions used in both the low-level and high-level XQuery are the same.

```
unordered(
  for $i in ($x1 to $x2)
    for $j in ($y1 to $y2)
      let $p := document("vmscope.xml")data/pixel[(x=$i)
          and (y = $j) and (scale ≥ $z1) and (scale ≤ $z2) ]
      return
        <pixel>
          <latitute> {$i} </latitute>
          <longtitute> {$j} </longtitute>
          <summary> { accumulate($p)} </summary>
        </pixel>
)

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

**Fig. 7.** Multigrid Virtual Microscope Using XQuery

The low-level XML Schemas and *getData* functions are expected to be invisible to the programmer writing the processing code. The goal is to provide a simplified view of the dataset to the application programmers, thereby easing the development of correct data processing applications. The compiler translating XQuery codes obviously has the access to the source code of the getData functions, which enables it to generate efficient code. However, an experienced programmer can still have access to *getData* functions and low-level Schemas. They can modify the low-level XQuery generated by the compiler, or even write their own version of *getData* functions and low-level XQuery codes. This is the major reason why our compiler provides an intermediate low-level query format, instead of generating the final executable code directly from high-level codes.

# 4 Compiler Analysis

In this section, we describe the various analysis, transformations, and code generation issues that are handled by our compiler.

## 4.1 Overview of the Compilation Problem

Because the high-level codes shown in Figures 5 and 7 do not reflect any information of how the actual layout of data, the first task for our compiler is to generate corresponding low-level XQuery codes.

After such high-level to low-level query transformation, we can generate correct codes. However, there are still optimization issues that need to be considered. Consider the low-level XQuery

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

**(a)**

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

**(b)**

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

**(c)**

**Fig. 8.** getData functions for Multigrid Virtual Microscope

code for virtual microscope shown in Figures 9. Suppose, we translate this code to an imperative language like C/C++, ignoring the *unordered* directive, and preserving the order of the computation otherwise. It is easy to see that the resulting code will be very inefficient, particularly when the datasets are large. This is primarily because of two reasons. First, each execution of the *let* expression will involve a complete scan over the dataset, since we need to find all data-elements that will belong to the sequence. Second, if this sequence involves $n$ elements, then computing the result will require $n + 1$ recursive function calls, which again is very expensive.

We can significantly simplify the computation if we recognize that the computation in the recursive loop is a reduction operation involving associative and commutative operators only. This means that instead of creating a sequence and then applying the recursive function on it, we can initialize the output, process each element independently, and update the output using the identified associative and commutative operators. A direct benefit of it is that we can replace recursion by iteration, which reduces the overhead of function calls. However, a more significant advantage is that the iterations of the resulting loop can be executed in any order. Since such a loop is inside an *unordered* nested *for* loop, powerful restructuring transformations can be applied. Particularly, the code resulting after applying *data-centric* transformation [9, 10] will only require a single pass on the entire dataset.

Thus, the three key compiler analysis and transformation tasks are: 1) transforming high-level XQuery codes to efficient low-level query codes, 2) recognizing that the recursive function involves a reduction computation with associative and commutative operations, and transforming such a recursive function into a *foreach loop*, i.e., a loop whose iterations can be executed in any order, and 3) restructuring the nested unordered loops to require only a single pass on the dataset.

```
unordered(
  for $i in ($x1 to $x2)
    for $j in ($y1 to $y2)
      let $p := getData ( $i, $j )
        where (scale ≥ $z1) and (scale ≥ $z2 ) ]
      return
        <pixel>
          <latitute> {$i} </latitute>
          <longtitute> {$j} </longtitute>
          <summary> { accumulate($p)} </summary>
        </pixel>
)
```

**Fig. 9.** Multigrid Virtual Microscope Using Low Level XQuery

An algorithm for the second task listed above was presented in our recent publication [11]. Therefore, we will only briefly review this issue, and focus on the first and the third tasks in the rest of this section.

### 4.2 High Level XQuery Transformation

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

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

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

We will use virtual microscope as an example to further describe the problem. As shown in Figure 7, in each iteration, the high-level XQuery code retrieves a desired set of elements from the dataset first, then, a recursive function is applied on this data stream to perform the reduction operation. There are three *getData* functions provided, each will retrieve an output data stream from the entire dataset. The issue is if and how the output stream from a *getData* functions can be used to construct the same data stream as used in the high-level query.

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

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

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

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

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

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

such that

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

and

$$O\,\frac{(x_1, x_2, \ldots, x_i)}{G} \supseteq O\,\frac{(I_1, I_2, \ldots, I_j)}{Q}$$

If a *getData* function $G$ is compatible with $Q$, it means that in any iteration of the query, we can call this *getData* function to retrieve a data stream from the dataset. Since this data stream is a superset of the desired data stream, we can perform another selection on it to get the correct data stream. Here, the second selection can be easily performed in memory and without referring to the low-level disk layout of the dataset. For the three functions shown in Figure 8, it is easy to see that the first two functions are compatible. Their selection criteria is either less or equally restrictive to what is used in the high-level query.

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

We use the following three step approach. First, we compute the relational algebra of the high-level XQuery and *getData* functions. A typical high-level XQuery program retrieves desired tuples from an XML file and performs computations on these tuples. We focus on the data retrieval part. The relational algebras of XQuery and the *getData* functions are shown in Figure 10 (a). Here, we use $\sigma_{(f)}E$ to represent *selection* from the entire dataset *E* by applying restriction *f*.

In the second step, we formalize these relational algebras into an equivalent *canonical form* that is easier to compare and evaluate. The canonical form we choose is similar to the disjunctive normal form (DNF), where the relations are expressed as unions of one or more intersections. Figure 10 (b) shows the equivalent canonical forms transformed. The actual canonical forms are internally represented by trees in our compiler.

In the third step, we compare the canonical forms of the high-level query and *getData* functions. For a given *getData* function, if its canonical form is an *isomorphic subtree* of the canonical form of the query, we can say that the *getData* function is compatible with the original query. This is because when replacing part of the relational algebra of the high-level query with a *getData* function, the query semantics are maintained. From Figure 10 (b) it is easy to see that the first two *getData* functions are compatible. $getData(\$x, \$y, \$z)$ is not compatible, because the its selection restriction on $\$z$ is $equal$, while the restriction of the query on $\$z$ is $\geq$ and $\leq$.

The next task is to choose the *getData* function which will result in the best performance. The algorithm we currently use is quite simple. Because applying restrictions early in a selection can reduce the number of tuples to be scanned in the next operation, a compatible *getData* function with the most parameters is preferred here. Formally, we select the function whose relational algebra in the canonical form is the *largest isomorphic subtree*. As shown in Figure 10 (c), the final function we choose is $getData(\$x, \$y)$. The resulting relational algebra for low-level XQuery is shown in Figure 10, part (c). Here, the pixels are retrieved by calling $getData(\$x, \$y)$ and then performing another selection on the output stream by applying the restriction on $scale$.

$$P_1(Hquery) : \sigma_{(x=*)\wedge(y=*)\wedge(scale\geq Z_1)\wedge(scale\leq Z_2)}E$$

$$P_1(getData(\$x)) : \sigma_{(x=*)}E$$

$$P_1(getData(\$x,\$y)) : \sigma_{(x=*)\wedge(y=*)}E$$

$$P_1(getData(\$x,\$y,\$z)) : \sigma_{(x=*)\wedge(y=*)\wedge(scale=*)}E$$

**(a)** Relational algebras for high-level query and getData function

$$P_2(Hquery): (\sigma_{(x=*)}E) \cap (\sigma_{(y=*)}E) \cap (\sigma_{(scale\geq Z_1)}E) \cap (\sigma_{(scale\leq Z_2)}E)$$

$$P_2(getData(\$x)) : \sigma_{(x=*)}E$$

$$P_2(getData(\$x,\$y)) : \sigma_{(x=*)}E \cap \sigma_{(y=*)}E$$

$$P_2(getData(\$x,\$y,\$z)) : \sigma_{(x=*)}E \cap \sigma_{(y=*)}E \cap \sigma_{(scale=*)}E$$

**(b)** Relational algebras in canonical form

$$P_3(Query) : \sigma_{(scale\geq Z_1)\wedge(scale\leq Z_2)}(\sigma_{(x=*)\wedge(y=*)}E)$$

**(c)** Relational algebra for low level query

**Fig. 10.** Relational Algebra Based Approach for High-level to Low-level Transformation

### 4.3 Reduction Analysis and Transformation

Now, we have a low-level XQuery code, either generated by our compiler or specified directly by an experienced programmer. Our next task is analyzing the reduction operation defined in low-level query. The goals of this analysis is to generate efficient code that will execute on disk-resident datasets and on parallel machines.

The reductions on tuples that satisfy user-defined conditions are specified through recursive functions. Our analysis requires the recursive function to be linear recursive, so that it can be transformed it into an iterative version. Our algorithm examines the syntax tree of a recursive function to extract desired nodes. These nodes represent associative and commutative operations. The details of the algorithm are described in a related paper [11]. After extracting the reduction operation, the recursive function can be transformed into a *foreach* loop. An example of this is shown in Figure 11. This *foreach* loop can be executed in parallel by initializing the *output* element on each processor. The reduction operation extracted by our algorithm can then be used for combining the values of *output* created on each processor.

### 4.4 Data Centric Transformation

Replacing the recursive computation by a *foreach* loop is only an enabling transformation for our next step. The key transformation that provides a significant difference in the performance is the *data-centric transformation*, which is described in this section.

In Figure 11, we show the outline of the virtual microscope code after replacing recursion by iteration. Within the nested *for* loops, the *let* statement and the recursive function are replaced by

```
unordered(
  for $i in ($x1 to $x2)
    for $j in ($y1 to $y2)
      foreach element $e in getData($i, $j )
        if (( $e/scale ≥ $z1 ) and ($e/scale ≤ $z2 ))
          Insert $e to the sequence $p
      Initialize the output
      foreach element $e in $p
        Apply the reduction function and update output
      return output
)
```

**Fig. 11.** Recursion Transformations for Virtual Microscope

two *foreach* loops. The first of these loops iterates over all elements in the document and creates a sequence. The second *foreach* loop performs the reduction by iterating over this sequence.

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

Since the input dataset is never modified, it is clearly possible to execute such code to require only a single pass over the dataset. However, the challenge is to perform such transformation automatically. We apply the *data-centric* transformation that has previously been used for optimizing locality in scientific codes [9, 10]. The overall idea here to iterate over the available data elements, and then find and execute the iterations of the nested loop in which they are executed. As part of our compiler, we apply this transformation to the intermediate code we obtain after removing recursion. The results of performing data-centric transformation on the virtual microscope are shown in Figures 12. This code requires only one scan of the entire dataset.

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

**Fig. 12.** Data-Centric Transformations on Virtual Microscope Code

## 5 Experimental Results

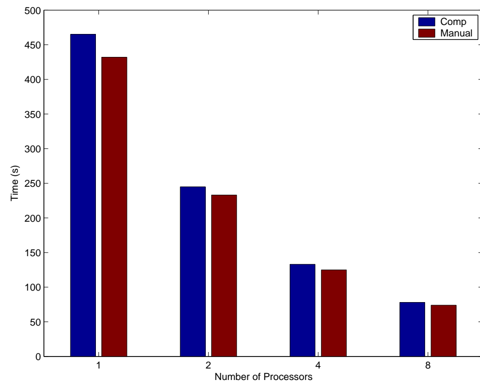


**Fig. 13.** Parallel Performance of `satellite`

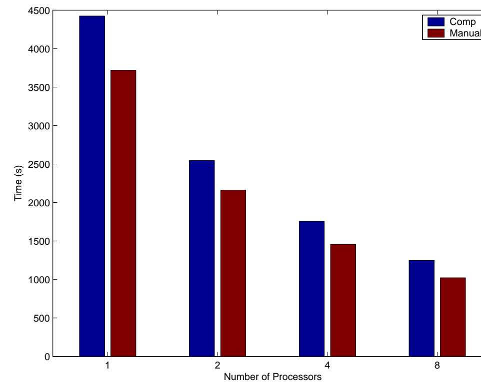


**Fig. 14.** Parallel Performance for `mg-vscope`

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

The goal of our experiments was to demonstrate that even with high-level abstractions and a high-level language like XQuery, our compiler is able to generate reasonably efficient code. The compiler generated codes for our two applications were compared against versions whose performance was reported in earlier work [9]. These versions were generated by a compiler starting from a data parallel dialect of Java, and were further manually optimized. For our discussion, the versions generated by our current compiler are referred to as `comp` and the baseline version is referred to as `manual`.

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

The results from `satellite` are presented in Figure 13. The results from `mg-vscope` are presented in Figure 14. For both the applications and on 1, 2, 4, and 8 nodes, the `comp` versions are slower. However, the difference in performance is only between 5% and 8% for `satellite` and between 18% and 22% for `mg-vscope`. The speedups on 8 nodes is around 6 for both versions of `satellite` and around 4 for both versions of `mg-vscope`. The reason for limited speedups is the high communication volume.

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

## 6 Conclusions

In this paper, we have described a system that offers an XML based front-end for storing, retrieving, and processing flat-file based scientific datasets. With the use of aggressive compiler transformations, we support high-level abstractions for a dataset, and hide the complexities of the low-level layout from the application developers. Processing on datasets can be expressed using XQuery, the recently developed XML Query language. Our preliminary experimental results from two applications have shown that despite using high-level abstractions and a high-level language like XQuery, the compiler can generate efficient code.

## References


1. Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.
2. D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. XML Schema part 1: Structures, W3C working draft. Available at http://www.w3.org/TR/1999/xmlschema-1, May 1999.
3. P. Biron and A. Malhotra. XML Schema part 2: Datatypes, W3C working draft. Available at http://www.w3.org/TR/1999/xmlschema-2, May 1999.
4. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C Working Draft, available from http://www.w3.org/TR/xquery/, November 2002.
5. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Available at http://www.w3.org/TR/REC-xml, February 1998.
6. Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.
7. Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.
8. R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., October 1997. Also available as University of Maryland Technical Report CS-TR-3777 and UMIACS-TR-97-35.
9. Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiler supported high-level abstractions for sparse disk-resident datasets. In *Proceedings of the International Conference on Supercomputing (ICS)*, June 2002.
10. Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.
11. Xiaogang Li, Renato Ferreira, and Gagan Agrawal. Compiler Support for Efficient Processing of XML Datasets. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 67–77. ACM Press, June 2003.
12. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with databases: alternative and implications. In *In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM Press, June 1998.