Using XQuery for Flat-File Based Scientific Datasets *

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

Abstract

XQuery is a recently developed query language for XML datasets. In this paper, we focus on the use of XQuery and other XML technologies for flat-file based scientific datasets. Traditionally, complex and domain-specific data layouts have complicated the processing of large datasets arising from scientific applications. The use of XML schemas and XQuery’s high-level structure can simplify the analysis on these datasets.

Though scientific data processing applications can be conveniently represented in XQuery, compiling them to achieve efficient execution involves a number of challenges. These are, 1) analysis of recursive functions to identify reduction computations involving only associative and commutative operations, 2) replacement of recursive functions with iterative constructs, 3) application of data-centric transformations on the structure of XQuery, and 4) translation of XQuery processing to an imperative language like C/C++, which is required for using a middleware that offers low-level data access functionality. This paper describes our solutions towards these problems and demonstrates significant benefits from the transformations we have developed.

1 Introduction

XQuery is a recently developed language for querying and processing XML datasets. We have initiated an effort for supporting the use of XQuery for applications that process scientific datasets. The motivation for our work is three-fold. First, distribution and use of data comprising scientific images and experimental or simulation results has traditionally been hindered by complex and specialized formats. XML is already being used for standardizing the distribution of data in many domains, and this trend can only be expected to continue. Thus, it is natural to use XQuery for specifying processing on data. Second, as we show in this paper, XQuery can significantly simplify the specification of processing on complex scientific datasets. Third, we believe that scientific data processing applications provide a new class of interesting benchmarks for XQuery. Most of the existing applications targeted for XQuery involve searching for records or documents with certain characteristics [20], and the existing implementation efforts have typically targeted these. In contrast, scientific data processing applications involve non-trivial computation.

We have particularly focused on applications that process multi-dimensional datasets, and involve generalized reductions. Analysis and processing of very large multi-dimensional scientific datasets (i.e., where data items are associated with points in a multidimensional attribute space) is an important component of science and engineering. Examples of these datasets include raw and processed sensor data from satellites [17], output from hydrodynamics and chemical transport simulations [15], and archives of medical images[1]. Our study of a variety of scientific data-intensive applications has shown that generalized reduction operations are very common in the processing structure. Processing for generalized reductions consist of three main steps. (1) Retrieving data items of interest, (2) Applying application-specific transformation operations on the retrieved input items, and, (3) mapping the input items to output items and aggregating, in some application specific way, all the input items that map to the same output data item. Most importantly, aggregation operations involve commutative and associative operations, i.e., the correctness of the output data values does not depend on the order input data items are aggregated.

While such scientific data processing applications
can be conveniently coded in XQuery, compiling them to achieve efficient execution involves a number of challenges. Because XQuery is a functional language, the only practical way for specifying reduction computations is using recursion. This, and some of the other related features of XQuery, lead to the following compiler analysis and restructuring challenges:

- Analysis of recursive functions to identify reduction computations involving only associative and commutative operations.
- Replacement of recursive functions with iterative constructs.
- Application of data-centric transformations on the structure of XQuery.
- Translation of XQuery processing to an imperative language like C/C++, which is required for using a middleware that offers low-level functionality. Particularly, the challenge is to deduce the types of data-structures to be used in the imperative language.

In this paper, we report our solutions towards the above problems. By implementing the techniques in a compiler and generating code for a runtime system called Active Data Repository (ADR) [5, 4], we are able to achieve efficient processing of disk-resident datasets.

2 XQuery Language

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. Two classes of expressions important for our work are as follows:

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

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

We illustrate the XQuery language and the for, let, where, and return expressions by an example, shown in Figure 1. 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, 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 1 is presented in Figure 2. 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.

![Figure 1: An Example Illustrating XQuery’s FLWR Expressions](image-url)
3 Scientific Data Processing Applications in XQuery

In this section, we describe a motivating scientific data processing application, satellite data processing [5], and show how it can be expressed using XQuery. Another example, multi-grid virtual microscope [1], is described in the appendix.

3.1 Satellite Data Processing

```xml
unordered(
  for $d in document("depts.xml")/dept
  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>
)
```

define function accumulate ($p) as double

```xml
{ 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))) } }
```

Figure 3: Satellite Data Processing Expressed in XQuery

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

The typical computation on this satellite data is as follows. A portion of Earth is specified through latitudes and longitudes of end points. A time range (typically 10 days to one year) is also specified. For any point on the Earth within the specified area, all available pixels within that time period are scanned and an application dependent output value is computed. To produce such a value, the application will perform computation on the input bands to produce one output value for each input value, and then 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 3. We currently assume a simplified data representation, where the input data is simply a set of pixels. Each pixel stores the latitude, longitude, time, and 16-bit measurements for the 5 bands. The code iterates over the two-dimensional space for which the output is desired. Since the order in which the points are processed is not important, we use the directive unordered. Within an iteration of the nested for loop, the let statement is used to create a sequence of all pixels that correspond to the those spatial coordinates. The desired result involves finding the pixel with the best NDVI value. In XQuery, such reduction can only be computed recursively.

4 Compiler Analysis

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

4.1 Overview of the Compilation Problem

For datasets that are stored as flat files, there are two ways in which XQuery is likely to be compiled. The first will be to translate it into an imperative language like C/C++, for which efficient compilers are available. The second will be to generate code for a middleware or runtime system, which offers data handling capabilities. Such systems also often offer interfaces based upon imperative languages. Thus, XQuery codes will likely need to be translated into codes in an imperative language.

Consider the code shown in Figure 3. Suppose, we translate it 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 [11, 14] will only require a single pass on the entire dataset.

Thus, the first three key compiler analysis and transformation tasks are: 1) recognizing that the recursive function involves a reduction computation with associative and commutative operations, 2) 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. Note that the essential analysis required for the first and the second tasks is the same.

Translating XQuery to imperative languages also leads to a new challenge, which is type inferencing. We need to deduce the types of the data-structures used in the target languages. The type system used in XQuery is significantly different from that in popular imperative languages, which makes the type inferencing problem a non-trivial one.

### 4.2 Analysis of Recursive Functions

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

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

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

(a)

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

(b)

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

(c)

---

Figure 4: Examples To Illustrate Reduction Analysis Algorithm

or should be functions of only the first element of the sequence \( $t$ \).

We use a series of examples to explain the algorithm we have developed. These examples are presented in Figure 4. Our algorithm is presented in Figure 5.

Our algorithm analyzes the abstract syntax tree (AST) of the function. Initially, it processes all leaf nodes of the tree. We focus on nodes of two types, the nodes that are a recursive call to the function \( F \) and the nodes that denote a variable defined by such a recursive call. In either of these cases, if the node is used as part of a return value, the node is inserted into the set \( S \). Consider the examples in Figure 4. For each of the cases (a) and (b), our algorithm will insert two nodes in the set \( S \). For the case (c), our algorithm initially find three nodes that are of interest, the recursive call node, the use of \( \text{val} \) inside the conditional, and the use of \( \text{val} \) as a return value. However, since only the last of these three is part of a returned value, it is the only one added to the set \( S \).

Note that a recursive reduction function may compute multiple simple types, each through a different reduction operation. For example, an averaging function may compute a sum field and a count field, applying reduction operations \( \text{add} \) and \( \text{add by one} \), respectively. Our algorithm separates the recursive calls used for
Analyze(\(AST \ T\))

\[ S = \emptyset \]
for each leaf node \(n\) in \(T\)
  if \(n\) is a recursive function call node
    if \(n\) is used as part of a returned value
      \(S = S \cup \{n\}\)
    else if \(n\) is a variable defined by a recursive call
      if \(n\) is used as part of a returned value
        \(S = S \cup \{n\}\)
Partition nodes in \(S\) based upon the field computed
let \(S_1, \ldots, S_k\) denote the \(k\) partitions
for \(i = 1, \ldots, k\)
  \(R = \emptyset\)
  for each node \(n\) in \(S_i\)
    \(R = R \cup \{\text{Findnode}(n)\}\)
  if \(S_i\) and \(R\) are singleton sets
    let \(R = \{t\}\)
    if \(t\) is an associative and commutative operation
      mark the sub-tree with \(t\) as the root of the reduction operation
    else mark the function as non-transformable
  else
    if each node in \(S_i\) is in different control path
      and each node in \(R\) is the same associative and commutative operation
      let \(t'\) be the least common ancestor of all nodes in \(R\)
      mark the sub-tree with \(t'\) as the root of the reduction operation
    else mark the function as non-transformable
\}

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

Figure 5: Algorithm for Analysis of Recursive Functions

computing each distinct field. Thus, the set \(S\) is partitioned into \(k\) subsets, \(S_1, \ldots, S_k\). For each of three examples in Figure 4, the value of \(k\) is 1.

Our algorithm subsequently processes each of these sets independently. For each node \(n\) in a set \(S_i\), we apply the function \(\text{Findnode}(n)\). This function finds the closest ancestor of \(n\) that has more than one child. The node thus obtained combines the value at \(n\) with another value. In Figure 4, case (a), the application of the function \(\text{Findnode}\) to the two nodes in the set \(S_1\) finds the two occurrences of the \textit{max} function. In case (b), the results are the \textit{max} function and the \textit{else} if clause. In case (c), the result is the \textit{if} statement.

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

In the example (c) in Figure 4, the node \(t\) such determined is an \textit{if} statement, and cannot automatically be determined by the compiler to be an associative and commutative operation. However, if we have additional information that the recursive function uses associative and commutative operations, we could still use the sub-tree rooted at \(t\) as the reduction operation. Several languages use annotations for marking reduction computations, including HPF-2 [12] and a data parallel dialect of Java we have used in our earlier work [10].

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

```
unordered(
    for $i$ in ($\text{minx to max}$)
    for $j$ in ($\text{miny to max}$)
        foreach element $e$ in document("satellite.xml")/data/pixel
            if ($e/x = i$) and ($e/y = j$)
                insert $e$ to the sequence $S$
                Initialize the output
        foreach element $e$ in $S$
            Apply the reduction function and update output
    return output
)
```

(a)

```
unordered(
    for $i$ in ($\text{minx to max}$)
    for $j$ in ($\text{miny to max}$)
        Initialize the output
        foreach element $e$ in document("satellite.xml")/data/pixel
            if ($e/x = i$) and ($e/y = j$)
                Apply the reduction function and update output
    return output
)
```

(b)

```
for $i$ in ($\text{minx to max}$)
    for $j$ in ($\text{miny to max}$)
        foreach element $e$ in document("satellite.xml")/data/pixel
            $i = e/x$
            $j = e/y$
            if ($i > \text{minx}$) and ($i \leq \text{max}$) and
               ($j > \text{miny}$) and ($j \leq \text{max}$)
                Apply the reduction function and update output[i,j]
```

(c)

Figure 6: Transformations on the Satellite Data Processing Code: Removing Recursion (a), Applying Loop Fusion (b), and Data-Centric Transformation (c)

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 6, part (a), we show the outline of the satellite data processing code after replacing recursion by iteration. Within the nested `for` loops, the `let` statement and the recursive function are replaced by 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. A minor improvement to this code is shown in Figure 6, part (b). A loop fusion is performed to require only a single `foreach` loop.

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 [11, 14]. The overall idea here is 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 and performing loop fusion. Because of the language constructs and the nature of the applications we focus on, the details of the algorithm we use are different from the previous work reporting such transformations.

We use the following approach. We create an abstract iteration space, corresponding to the nested `for` loops within the `unordered` construct. Then, for each element in any document that is accessed in the program, we determine a necessary and sufficient condition for the element to be read in an iteration $i$ of the abstract iteration space. If a necessary and sufficient condition cannot be found, our compiler cannot perform the transformation. Otherwise, using this condition, we synthesize a mapping from an element to the set of iterations in which it is accessed. In our example, this is a singleton set.

An array of output elements corresponding to the iteration space is allocated and initialized in the beginning. Subsequently, each element from the document is accessed and mapped to zero or more iterations in which it can be accessed. Then, the computations associated with these iterations are performed, and the corresponding elements of the output array are modified. The outline of the resulting code for the satellite data processing is shown in Figure 6, part (c).

4.4 Type Analysis and Conversion

As stated previously, our compiler generates code for Active Data Repository (ADR) [3, 4], which uses C++ virtual functions to specialize the processing for a particular application. To generate C++ code for an XQuery program, one challenging task is the correct and efficient conversion of various types in XQuery to corresponding C++ types. This problem needs to be addressed by any compiler for XQuery whose target code is in an imperative or object-oriented language.

XQuery is a statically typed functional language. The type annotations of each operand, argument, and function are always either explicitly declared or can be generated while validating against a Schema. This static feature of the type system provides various possibilities for compiler optimizations and analysis, such as the well studied method for static type checking [3], and in our case, static type analysis for translation to
an imperative language. Although type systems of both C++ and XQuery are static, the mapping between these two is not straightforward. There are several issues that make the translation problem challenging.

```
typedef struct {
  int type;
  union {
    double d;
    int i;
  }
} DType;

typeswitch (d) {
  case DType d_double: return d.d;
  case DType d_int: return d.i;
  default: return 0;
}
```

Figure 7: XQuery expression with different types

First, unlike variables in an imperative language, an expression in XQuery may be associated with values of several different types. An example of such an expression is shown in Figure 7. The expression here may return either a double or an integer, depending upon the type of the variable `pixel`. To perform the translation, our compiler needs to collect the static types of all possible branches and compute a union of these types. For union of simple types, we will generate a corresponding union type in the target code. For union of complex types, we use the polymorphism of C++ language.

The second issue is related to the parametric polymorphism of XQuery. An actual argument of a function is only required to be of a subtype of the declared type of the corresponding formal function parameter. For example, the `max` function in Figure 7 can be invoked with an argument whose type may be either integer or double. Moreover, some parameters may be declared as `AnyType`, implying that their type can only be known by validating against an XML Schema. Therefore, to infer the type of a formal parameter, we need to gather information about the actual arguments from call sites, and if necessary, from Schema definitions. In cases where such parametric polymorphism is used, we use `function cloning`, i.e., we generate a copy of the function for each distinct type we infer. We believe this gives better performance than using polymorphism of C++, considering the limited occurrences of such functions in the actual case studies. Overall, our static type analysis algorithm is based on constraint-based type inference algorithm [13, 2], and the static type checking algorithm listed as part of the specification of XQuery formal semantics [8]. The goal of this algorithm is to infer the collection of all possible static types for a given expression, which will be used to guide the code generation in our compiler. The algorithm is top-down and recursive, since the static type of an expression depends only on the types of its sub-expressions.

The first step in our algorithm is to initialize the type of each expression, if the type is explicitly defined in the XQuery program or in an XML Schema. After initialization of type variables, we will analyze possible types for an expression by propagating type variables from subexpressions according to possible runtime data flow. Specially, if the target expression has only one possible value, the outcome is just propagated from its operands. For operands with different but compatible types, a `least upper bound` for these compatible types is returned as the result. If the target expression is a function call expression, information about the types of the actual arguments needs to be gathered from the call sites. This is done by applying the algorithm for each actual argument. The resulted types are assigned to each formal parameter, and the function body of the corresponding function definition is processed by the algorithm. For any other expression whose value is defined by subexpressions in more than one branch, we simply apply the algorithm for all sub-expressions in each branch and compute a union for all possible results. The detailed algorithm is listed in Figure 8.

```
TypeAnalysis (Expression E) {
  Set S = φ
  if t is explicitly assigned to E
    return t;
  else if E is a function call expression {
    E = body of function definition expression of E
    for each actual argument e of E
      assign type of actual arguments to formal parameters
      return TypeAnalysis (E)
  }
  else if E has single control flow path {
    for each subexpression E of E {
      S=S∪ (TypeAnalysis (E))
    }
    return least upper bound of S
  }
  else for each control path P of E {
    S=S∪ (TypeAnalysis (P))
    return S
  }
}
```

Figure 8: Algorithm for Static Type Analysis

After finishing type analysis for each expression in a XQuery program, we can generate C++ code that correctly implements the type system of XQuery. The C++ code generated after type analysis and code generation for the code in Figure 7 is shown in Figure 9. Because the `typeswitch` expression may return either a double or an integer type, a `union` type is declared to keep the result of this expression. The variable `$pixel` may be bound to two complex types when validated against a Schema. Therefore, we declare a superclass for `$pixel`, from which two subclasses can be derived. Also in this example, because the actual arguments of the function `max` can be either double or integer, a clone of the function is generated for each type.
5 Experimental Results

To evaluate our techniques and current prototype compiler implementation, we have evaluated the impact of our restructuring transformations, i.e., removing recursion and applying data-centric execution, on sequential execution.

We compared sequential performance between two versions, opt and naive. The naive version corresponds to a direct translation of XQuery codes to C++. The opt version includes replacing recursion with iteration and application of data-centric transformation. The resulting code is similar to the one shown in Figure 6, part (c).

We used four applications for performing the comparison. sum is a very simple sum reduction, where values associated with pixels having the same x and y coordinates are added together. irreg is a simple irregular reduction, where the values associated with edges are used to increment the values associated with corresponding nodes. satellite and mg-vscope are two real applications we described in Section 3.

Our experiments were conducted on a 933 MHz Pentium III workstation, with 256 MB of RAM, and running Linux version 7.1. We used four synthetic datasets for each of four applications. Their approximate sizes were 150 MB, 300 MB, 600 MB, and 1.2 GB, respectively. Thus, we studied the benefits of transformations as the size of the dataset is increased.

The results from sum are presented in Figure 10. The output produced by this kernel is a 2 × 2 array. Thus, the naive version requires 4 passes over the entire data, whereas, the opt version requires only a single pass. The difference between the opt and naive versions is consistently a factor of 4. Thus, it appears that the execution time is dominated by memory and disk access times.

The results from irreg kernel are presented in Figure 11. The output of this application is a 12×12 array. The difference between the two versions is a factor of 4 for the 150 MB dataset, and a factor of 8 for each of the other three datasets. Note that the first dataset can be cached on memory, therefore, subsequent passes over the data only require memory accesses. In comparison, the other three dataset cannot be cached and multiple passes over the dataset result in more disk accesses. This accounts for higher difference in performance for the larger datasets.
aggressive transformations can be applied for optimizing them.

7 Conclusions

With the wide acceptance of XML as the format for exchanging information, we envision that XML query language XQuery will be a popular language for specifying processing over datasets. This paper has described our initial efforts in compiling XQuery. Particularly, our focus has been on data-intensive reductions over scientific datasets.

Our work has demonstrated that XQuery simplifies the specification of processing over datasets. At the same time, however, naive translation of XQuery results in very slow execution. We have shown that aggressive restructuring transformations are the key to improving performance, and their application results in several-folds improvement in performance.

References


Appendix: The Multi-Grid Virtual Microscope Application

The Virtual Microscope [9] is an application to support the need to interactively view and process digitized data arising from tissue specimens. The raw data for such a system is captured by digitally scanning collections of full microscope slides at high power. In a typical dataset available when a virtual microscope is used in a distributed setting, the same portion of a slide may be available at different resolution levels, but the entire slide is not available at all resolution levels.

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

XQuery code for performing such computations is shown in Figure 14. We have made two simplifications. First, we assume that each pixel stores the x and y coordinates, as well as the resolution level. Second, we assume that the user is only interested in viewing the image at the highest possible resolution level, which means that averaging is never done to produce the output image. The structure of this code is quite similar to our previous example. Inside an unordered for loop, we use the let statement to compute a sequence, and then apply a recursive reduction.