Code Transformations for One-Pass Analysis

Xiaogang Li Gagan Agrawal

Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{xgli,agrawal}@cse.ohio-state.edu

Abstract. With the growing popularity of streaming data model, processing queries over streaming data has become an important topic. Streaming data has received attention in a number of communities, including data mining, theoretical computer science, networking, and grid computing. We believe that streaming data processing involves challenges for compilers, which have not been addressed so far. Particularly, the following two questions are important:

– How do we transform queries so that they can be correctly executed with a single pass on streaming data ?
– How do we determine when a query, possibly after certain transformations, can be correctly executed with only a single pass on the dataset.

In this paper, we address these questions in the context of XML query language, XQuery. Because of XQuery's single assignment nature and special constructs for dealing with sequences, the above questions can be answered more easily than for a general imperative language. However, we believe our work also forms the basis for addressing these questions for more general languages.

1 Introduction

Increasingly, a number of applications across computer sciences and other science and engineering disciplines rely on, or can potentially benefit from, analysis and monitoring of *data streams*. In the stream model of processing, data arrives continuously and needs to be processed in *real-time*, i.e., the processing rate must match the arrival rate. There are two trends contributing to the emergence of this model. First, scientific simulations and increasing numbers of high precision data collection instruments (e.g. sensors attached to satellites and medical imaging modalities) are generating data continuously, and at a high rate. The second is the rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Light Rail (NLR) proposal and the interconnectivity between the TeraGrid and Extensible Terascale Facility (ETF) sites. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster.

Realizing the challenges posed by the applications that require real-time analysis of data streams, a number of computer science research communities have initiated efforts. In the theoretical computer science or data mining algorithms research area, work has been done on developing new data analysis or data mining algorithms that require only a single pass on the entire data [17]. At the same time, database systems community has been developing architectures and query processing systems targeting continuous data streams [4].

We believe that streaming data processing involves challenges for compilers, which have not been addressed so far. Particularly, the following two questions are important:

– How do we transform queries so that they can be correctly executed with a single pass on streaming data ?
– How do we determine when a query, possibly after certain transformations, can be correctly executed with only a single pass on the dataset.

In this paper, we address these questions in the context of XML query language, XQuery. XML is a flexible exchange format that has gained popularity for representing many classes of data, including structured documents, heterogeneous and semi-structured records, data from scientific experiments and simulations, digitized images, among others. As a result, querying XML documents has received much attention. To query and process XML data streams, XQuery designed by W3C [7] can be an ideal language, because of its declarative nature and powerful features. XQuery is a high-level language like SQL, but it also supports more advanced and complex features such as types and recursive functions. XQuery allows user-defined functions, which are often key for specifying the type of processing that is required for streaming data.

In this paper, we make the following contributions. In many cases, direct translation of a XQuery query requires multiple passes on the data, whereas the query can be transformed to correctly execute with only a single pass. We present techniques for enabling such transformations. We model the dependencies in the query using a representation we refer to as the *stream data flow graph*. We apply a series of high-level transformations, including *horizontal* and *vertical* fusion. These techniques enable a larger number of queries to be evaluated correctly on streaming data, and efficiently on any large dataset. Moreover, based on our stream data flow graph, we present a methodology to determine if a query can be evaluated correctly in a single pass. This enables us to avoid generating a query evaluation plan that is going to fail, and instead, a user can be given feedback sooner.

Our transformations are implemented as part of a XQuery compilation system. Our experiments with eight queries show that our techniques are able to transform them for single-pass execution, whereas naive execution is very expensive.

The rest of the paper is organized as follows. A motivating application is described in Section 2.3. The overall problem is described in Section 3. Our analysis, including the stream data flow graph, horizontal and vertical fusion techniques, and the analysis to determine if the query can be executed correctly on streaming data are presented in Section 4. Experimental evaluation is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2 Background: XML, XML Schemas, and XQuery

```
< 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

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. XML and related technologies form the core of the web-services model [13] and the Open Grid Services Architecture (OGSA) [15].

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.

Applications that operate on XML data often need guarantees on the structure and content of data. XML Schema proposals [5, 6] 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 recently 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*.
– 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.

```
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. 2. An Example Using XQuery's FLWR and Unordered Expressions

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, 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.

The last key-word we explain is *unordered*. 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.

2.3 A Motivating Application

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

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

Fig. 3. Satellite Data Processing Expressed in XQuery

We now describe an application we refer to as *satellite data processing* [9]. We show how it can be expressed in XQuery, and the issues involved in transforming and executing it correctly on streaming data.

This application involves processing the data collected continuously from satellites and creating composite images. A satellite orbiting the Earth collects data as a sequence of pixels. Each pixel is characterized by the spatial coordinate (the latitude and longitude) and a time coordinate. The satellite contains sensors for five different bands. Thus, each pixel captured by the satellite stores the latitude, longitude, time, and 16-bit measurements for each of the 5 bands.

The typical computation on this satellite data is as follows. A portion of Earth is specified through latitudes and longitudes of end points. For any point on the Earth within the specified area, all available pixels (corresponding to different time values) 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 3. 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.

The computations performed to obtain the output value of a given spatial coordinate are often associative and commutative. In such cases, these computations can be performed correctly on streaming data. When a pixel is received, we can find the spatial coordinate it corresponds to, and update the output value for that spatial coordinate.

However, direct translation of the XQuery specification, as we had shown in Figure 3, will require multiple scans on the entire dataset. It is clearly desirable that the streaming XQuery processor can transform the query to execute it correctly with only a single pass on the entire dataset. Thus, we have the following challenges:

1. How can we systematically and correctly transform a given XQuery query so that it can be executed on streaming data, when possible ?
2. How can we determine if a given XQuery query, possibly after our transformations, can be executed correctly with only a single pass on the entire dataset ?

We address the above two challenges in the rest of this paper.

3 Preliminaries

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

3.1 Evaluation Model

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

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

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

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

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

and the output stream is

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

A pipeline operator f has the property:

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

where, h is monotonically increasing and b is a bounded size buffered synopsis of $x_1, x_2, \ldots, x_{h(i)-1}$. An example of a pipeline operator is the selection operation.

Blocking Operator: A blocking operator must receive all its input before generating the output. Using the above notation for input and output, for a blocking operator we have

$$[y_1, y_2, ...y_k] = g(x_1, x_2, ..., x_n)$$

An example of a blocking operator is the sort operation.

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

$$|Output(f)| \ll |Input(f)| \tag{1}$$

$$g(x_1, x_2, ..., x_n) = g_1(g(x_1, x2, ..., x_{n-1}), x_n) \tag{2}$$

In such cases, the operator can be evaluated as follows. At each step, we only need to buffer the temporary results and can discard the input. This is because the Equation 2 ensures that the input is no longer necessary for the later computations. Equation 1 ensures that temporary results can actually be buffered in our evaluation model. An example of such an operator is the count operation.

3.2 Problem Overview

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

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

The first condition is straight-forward. Let us consider the second condition. When the final output of a progressive blocking operator f_1 is referred by another operator f_2, which is either a pipeline or a progressive blocking operator, f_2 must wait until the computation of f_1 finishes. This blocks the pipeline or progressive blocking computation f_2 defines. Queries that satisfy this propriety are referred to as *correlated aggregates* [16], which in most cases can only be evaluated approximately with a single pass.

The dependence between blocking operators and pipeline or progressive blocking operators that prevents a query from being evaluated in a single pass can either be a control dependence or a data dependence. The following query, referred to as the `Query 1`, is an example where data dependence between operators is involved. Here, `pixel` contains two elements, `x` and `y`.

```
Query 1:
let $b = count(stream/pixel[x>0])
  for $i in stream/pixel
     return $i/x idvi $b
```

4 High-level Analysis

This section describes the high-level analysis done in our system. Our goal is to correctly transform the query so that it can be processed in a single pass, when it is possible, and also to recognize when single pass analysis is not possible. Initially, we give an overview of our overall framework.

4.1 Overview

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

Figure 4 shows the key phases in our system. First, we construct the stream data flow graph representing the data dependence information for the query. Then, we apply a series of high-level transformations to prune and merge the stream data flow graph. Such techniques not only simplify the later analyses, but most importantly, they can rewrite some queries to enable single pass processing. After pruning the graph, a *single pass analysis* algorithm will be applied to the resulting data flow graph to check if single pass evaluation is possible. If the answer is no, further processing will not be performed. Otherwise, we apply low-level transformations and our code generation algorithm, and efficient single pass execution code is generated.

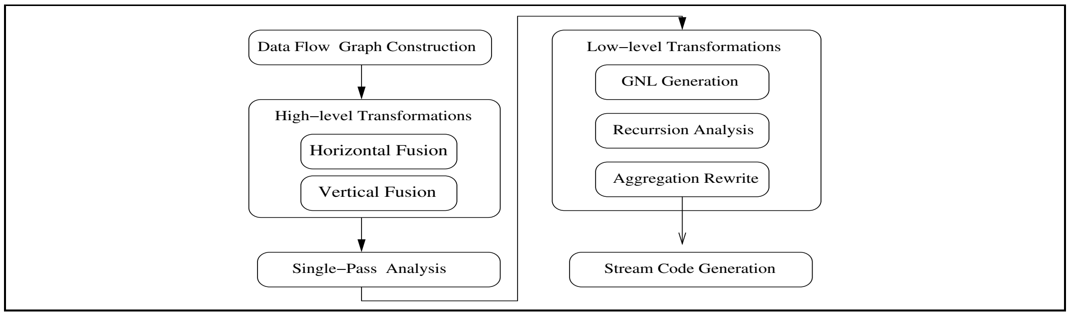

Fig. 4. Overview of the Framework

4.2 Stream Data Flow Graph

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

Definition 1 *A stream data flow graph is a directed graph in which each node represents a variable in the original query and the directed edges $e = (v_1, v_2)$ implies that v_2 is dependent on v_1.*

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

The stream data flow graph for the `Query 1` described in the previous section is shown in Figure 5. $S1$ is the implicit variable that represents the XPath expression `stream/pixel[x>0]`. Similarly, $S2$ is used to represent `stream/pixel`. The output of the aggregate function `count()` is represented by $v1$. Here i in the for clause is treated as an atom variable to represent each item in the binding sequence.

Lemma 1. *The stream data flow graph for a valid XQuery query is acyclic.*

Proof: The proof directly follows from the single assignment feature of XQuery [7]. Assume there is a cycle, then one of the following conditions must hold true: 1) a variable v is defined more than once, or 2) a variable v is referred to without definition.

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

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

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

Aggregate dependence typically exists between a progressive blocking operator and its output.

Definition 3 *Given two variables v_1,v_2, we say that v_2 is flow dependent on v_1 if: 1) v_2 is dependent on v_1, and 2) v_2 is not aggregate dependent on v_1. In such a case, we denote $v_1 \rightarrow v_2$.*

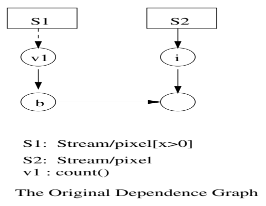

Fig. 5. Example of Stream Data Flow Graph

Let us reconsider the Figure 5. We have used dashed arrows to represent aggregate dependence, and solid arrows for flow dependence.

4.3 High-level Transformations

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

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

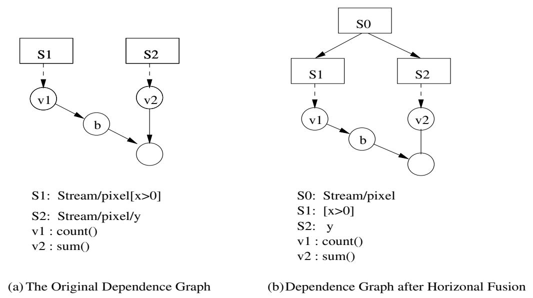

Fig. 6. Example of Horizontal Fusion

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

As an example, we consider the following query:

```
Query 2:
let $b = count(stream/pixel[x>0])
     return sum(stream/pixel/y) idvi $b
```

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

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

The stream data flow graph for the `Query 2` after horizontal fusion is shown in Figure 6. In this example, a new sequence node $S0$ is generated corresponding to the common prefix `/stream/pixel`. The label of the two original sequence node are changed to the remaining XPath expressions, which are $[x > 0]$ and $/y$, respectively. Each new node is linked to $S0$.

Sometimes horizontal fusion in a query may lead to incorrect results, because of inter-dependence among the traversal of sequences. As an example, consider the `Query 1`. The data flow graph after horizontal fusion is shown in Figure 7. When we combine the traversal to compute count and the final output together, in each iteration, the output will be computed using partial result of $b, which is not correct. In our method, we just apply horizontal fusion irrespective of such inter-dependence. Later, during single pass analysis, such dependence will be detected and the query will be eliminated from further processing.

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

```
unordered(
 for $i in (1 to 2)
   let $b: =//stream/pixel[x=$i]
      return count($b))
```

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

```
unordered(
   let $b1: =//stream/pixel[x=1]
   let $b2: =//stream/pixel[x=2]
      return count($b1), count($b2)
```

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

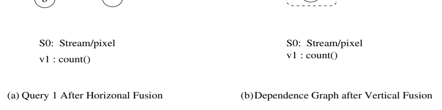
S0
v1
i
b
S0: Stream/pixel
v1 : count()
(a) Query 1 After Horizonal Fusion
S0
v1
i
b
S0: Stream/pixel
v1 : count()
(b) Dependence Graph after Vertical Fusion

Fig. 7. Horizontal and Vertical Fusion for Query 1

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

Consider the following example.

```
Query 3:
let $b: = for $i in  stream/pixel[x>0]
   return $i
for $j in $b/y
   return $j
   where $j = count($b)
```

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

As described in 3.2, we only need to check dependence between a progressive blocking operator and a pipeline operator, while dependence among pipeline operators can be ignored. In vertical fusion, we try to merge multiple pipeline operations on each traversal path into a single cluster in the stream data flow graph. The cluster obtained after fusion is referred to as a *super-node*. A super-node is represented in the data flow graph with a dashed box enclosing all the merged nodes. By doing so, the pipeline operation and the progressive blocking operations can be separated, and the number of isolated nodes in the data flow graph is reduced. This significantly simplifies later analysis on their dependence relationships.

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

The details of the algorithm are shown in Figure 9. R is the set of the nodes in the graph that do not have an incoming edge. N denotes the set of nodes that have been inserted in any super-node. $\bar{N}$ denotes the compliment of N, i.e., the nodes in the graph that are not in the set N. The algorithm picks a sequence node s_i. It follows the flow dependence edges (denoted as $\rightarrow$) to find nodes that can be fused into a super-node with s_i. These nodes are put in the set M. Any node that has already been fused into a super-node, (i.e., is not in $\bar{N}$) is not inserted in M.

The data flow graph for the `Query 1` after vertical fusion is shown in Figure 7(b). The data flow graph for the `Query 3` after vertical fusion is shown in Figure 8 (b).

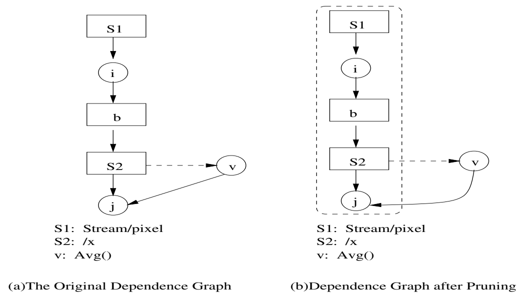

Fig. 8. Example of Vertical Fusion (Query 3)

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

4.4 Single Pass Analysis

After horizontal and vertical fusion, analyzing whether a query can be evaluated in a single pass becomes simpler. *For our discussion here, we treat all nodes in a super-node after vertical fusion as a single sequence*

```
Vertical_Fusion
Input: 1) data flow graph G =(V ,E)
       2) root set R

$N = \oslash$
foreach node $s_i \in R$ {
    if $s_i$ is a sequence node
        $M = \{s_i\}$
    do {
        $N = N \cup M$
        Let $T = \{v | \exists x, (x \in M) \wedge (x \to v)\}$
        $M = M \cup (T \cap \bar{N})$
    } until $(T \cap \bar{N} == \oslash)$
    fuse M into super-node
}
end
```

Fig. 9. Algorithm for Vertical Fusion

node. With this, any stream data flow graph that contains more than one sequence node cannot be evaluated in a single pass. This is because each such node represents one traversal of a sequence of length $\theta(\mathcal{N})$. If two sequence nodes are not fused with vertical fusion to apply pipelined execution, two traversals must be used. Thus, we have the following theorem.

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

However, for queries whose stream data flow graph contains only one sequence node, a single pass evaluation may still not be possible. Two types of dependence relationship may prevent the query from being executed in a single pass. Examples of these two cases are shown in Figure 10.

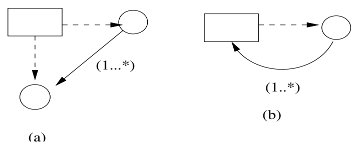

Fig. 10. Stream Data Flow Graphs that Require Multiple Traversals

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

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

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

Lemma 2. *If a stream data flow graph G contains a cycle, it is formed after horizontal or vertical fusion.*

Proof: From lemma 1 there is no cycle in the original stream data flow graph. Therefore, the cycle must be formed by either horizontal fusion or vertical fusion.

Theorem 3 *If there is a cycle in a stream data flow graph G, the corresponding query may not be evaluated correctly using a single pass.*

Proof: From the lemma above, the cycle is formed after horizontal or vertical fusion. If the cycle is formed right after horizontal fusion of s_1 and s_2, there must be a path between s_1 and s_2, which implies dependence of s_2 on s_1. In this case, horizontal fusion will generate incorrect results, and single pass evaluation is impossible.

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

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

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

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

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

1. *There are multiple sequence nodes.*
2. *There is a cycle involved.*
3. *$\exists$ sequence node $s \in V$, $\exists$ atomic nodes $a_1 \in V$,$a_2 \in V$, a_1 and a_2 are aggregate dependent on s, and there is a path from a_1 to a_2.*

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

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

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

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

```
let $p: =  stream/pixel/x
 for $i in $p
   where $i <= max($p)
   return $i
```

This query has a *redundant predicate* [3]. Though the predicate always returns true and does not impact the results from the query, it introduces a cycle in our graph, and disallows processing with a single pass. Our analysis can be extended to recognize and remove such redundant predicates, but we do not expect them to arise frequently in real situations.

XMark Query 1

Size	Ours	Qizx	Saxon	Galax
1.16M	0.76	1.03	2.46	4.65
5.75M	2.26	3.2	5.57	24.59
30M	9.98	11.23	MO	173.85
120M	13.97	MO	MO	*
240M	27.59	MO	MO	*

XMark Query 5

Size	Ours	Qizx	Saxon	Galax
1.16M	0.74	1.09	2.46	4.93
5.75M	2.30	3.35	5.55	25.26
30M	10.02	13.9	MO	174.08
120M	13.95	MO	MO	*
240M	27.87	MO	MO	*

XMark Query 6

Size	Ours	Qizx	Saxon	Galax
1.16M	0.73	1.07	2.42	4.75
5.75M	2.26	3.21	5.39	24.96
30M	9.94	13.68	MO	215.64
120M	13.87	MO	MO	*
240M	27.81	MO	MO	*

XMark Query 7

Size	Ours	Qizx	Saxon	Galax
1.16M	0.74	1.13	2.44	6.6
5.75M	2.28	3.45	5.53	47.79
30M	9.95	13.96	MO	MO
120M	13.70	MO	MO	MO
240M	27.44	MO	MO	MO

XMark Query 20

Size	Ours	Qizx	Saxon	Galax
1.16M	0.78	1.32	2.57	5.15
5.75M	2.31	3.59	5.93	26.38
30M	10.00	15.77	MO	190.22
120M	14.16	MO	MO	*
240M	27.81	MO	MO	*

Satellite Processing

Size	Ours	Qizx	Saxon	Galax
0.05M	0.28	5.88	3.08	72.03
0.10M	0.33	20.48	4.45	136.7
0.66M	0.48	945.5	18.76	944.4
10.6M	3.47	*	MO	MO
100M	28.31	MO	MO	MO

Virtual Microscope

Size	Ours	Qizx	Saxon	Galax
0.05M	0.28	47.51	2.01	18.97
0.10M	0.32	*	2.47	38.98
0.66M	0.44	*	7.66	300.18
2.70M	1.54	*	24.56	MO
10.6M	3.29	*	MO	MO
100M	27.88	*	MO	MO

Karp Frequent Item

Size	Ours	Qizx	Saxon	Galax
0.05M	0.26	*	4.71	25.09
0.10M	0.32	*	10.66	122.63
0.66M	0.61	*	554.07	MO
2.70M	1.80	*	8302.7	MO
10.6M	5.61	*	MO	MO
100M	29.41	*	MO	MO

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

Fig. 11. Experiments Results for XMark Queries and Real Streaming Applications (All Execution Times in Seconds)

5 Experimental Results

Our transformations have been implemented as part of a XQuery compilation system that is based on the open source SAX parser[1]. In this section, we demonstrate that many XQuery queries can be transformed to achieve single-pass execution, whereas their naive execution results in much more expensive processing. For this purpose, we took 8 XQuery queries, and compared our implementation with other well known XQuery processors which are publically available. Specifically, we use Galax (Version 0.3.1) [11], Saxon (Version 8.0) [19] and Qizx/Open (Version 0.4/_p1) [1]. All these query processors are implemented using a SAX Parser, which we believe makes the comparison reasonable.

We used two sets of queries for our experiments. The first set comprised the queries 1, 5, 6, 7, and 20 from the XMark benchmark set [25]. These five queries were chosen because each of them could be processed in a single pass either directly, or after our transformations. We use datasets of different sizes, which were generated by the XMark data generator using factors 0.01, 0.05, 0.25, 1, and 2, respectively. The second set comprised three real applications which involve streaming data. Satellite data processing was described earlier in Section 2. Virtual microscope is an application to support interactive viewing and processing of digitized data arising from tissue specimens [12]. Frequent element counting is a well known data mining problem, here we use the one-pass algorithm by Karp *et al.* to find a superset of frequent items in a data stream [18]. Each of these three applications uses recursive functions to perform aggregations. After applying our techniques and optimizations, including analysis of recursive functions, aggregate rewriting, and horizontal and vertical fusion, each of these could be processed correctly using only a single pass on the entire data stream. We generated synthetic datasets of varying sizes to evaluate performance on these applications.

[1] http://www.saxproject.org

The results of our experiments are shown in Figure 11. Our experiments were conducted on a 933 MHz Pentium III workstation, with 256 MB of RAM, and running Linux version 7.1, with JDK V1.4.0. Each of the systems we compared was executed on this same environment. In the tables in Figure 11, `Ours` denotes our basic framework. Because we use compiled Java byte code, the running time shown in the tables excludes the compilation time for other XQuery systems. All available options for fast execution and optimization are turned on for each system. Specifically, for Galax, we disable sorting and duplicate removal on Path expressions, and set the option of projection to be on.

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

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

For the three real streaming applications, our implementation outperforms other systems by at least one order of magnitude, and often, much more. None of the other systems was able to execute these applications with only a single pass on the data, whereas, our techniques and transformations enabled such execution.

6 Related Work

Our work is related to the large body of research in the area of loop transformations. Many loop transformations, such as loop fusion [27] or data-centric transformations [21] can often help in executing a code with a single-pass or fewer passes on the input data. However, we are not aware of any previous work on systematically transforming and analyzing code for single pass analysis on streaming data.

Language and compiler support for streaming data has been considered by the StreamIt effort at MIT [26]. There work also does not consider analysis and transformation to enable single-pass analysis.

There have been many research efforts on efficient evaluation of XPath expressions over streaming data. Because of the regularity of XPath expressions, automaton based approaches are most popular when predicates are not present [2, 10, 8]. To deal with predicates and other features, such as closures where buffering of certain elements is necessary, transducers have been used in XSQ [24] and SPEX [23].

Compared with XPath, XQuery is more expressive, and therefore, involves additional challenges. Currently, there is limited work on processing XQuery queries over streaming data. Transducer networks have also been used to handle a subset of XQuery, in which only join and node creation operations are investigated [22]. Without query transformations and rewriting, their techniques will not work on streaming data when the queries are not strictly written to execute on streaming data. In Flux [20], an intermediate representation (IR) extends XQuery with new constructs for event-based processing. XQuery is translated into this event-based IR and the buffer size is optimized by analyzing the DTD as well as the query syntax. Fusion of *for* expressions has been discussed in Flux, but algorithms to systemically perform such optimizations are not provided. In comparison, we present systematic and powerful techniques for optimizing and transforming queries that are not specifically written for single-pass processing. For code generation based on SAX events, we use a similar approach to enable efficient buffering. As we stated earlier, our additional contribution in code generation is handling user-defined aggregations with the use of GNLs. The BEA/XQRL processor [14] supports pipelined processing of streams by implementing the iterator model at the expression level. However, query optimizations specially designed for XML streams are limited in this system, and large documents cannot be processed.

Algebraic approach for deciding whether a SQL-like query can be evaluated with a single pass on continuous streams has been proposed recently by Babu and Widom [3]. Their approach cannot handle user-defined aggregates and computations described with binary expressions, which are both frequently used in XQuery. Unlike SQL, developing an algebra to handle complete XQuery is hard. As an example, user defined functions allowed as part of XQuery can be very hard to model through such an algebra, and we are not aware of any existing effort which is able to do this.

7 Conclusions

Our work has been driven by growing popularity of the streaming data model. We have considered the following two questions. First, how do we transform queries so that they can be correctly executed with a single pass on streaming data. Second, how do we determine when a query, possibly after certain transformations, can be correctly executed with only a single pass on the dataset.

We have addressed these questions in the context of XML query language, XQuery. However, we believe our work also forms the basis for addressing these questions for more general languages.

References

1. Qizx/open: An open source implementation of xml query in java. http://www.xfra.net/qizxopen/.
2. Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, 2000.
3. Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. *ACM Transactions on Database Systems*, 29(1):162–194, 2004.
4. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 2002 ACM Symposium on Principles of Database Systems (PODS 2002) (Invited Paper)*. ACM Press, June 2002.
5. 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.
6. 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.
7. 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.
8. C. Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML documents with XPath Expressions. *VLDB Journal: Very Large Data Bases*, 11(4):354–379, December 2002.
9. 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.
10. Y. Diao, P. Fischer, and M. J. Franklin. Y. Filter: Efficient and Scalable filtering of XML Documents. In *Proceedings of the 18th International Conference of Data Engineering*, 2002.
11. Mary F. Fernandez, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing Xquery 1.0: The Galax experience. In *VLDB 2003: Proceedings of 29th International Conference on Very Large Data Bases, September 9–12, 2003, Berlin, Germany*, pages 1077–1080, 2003.
12. 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.
13. Chris Ferris and Joel Farrell. What are Web Services. *Communications of the ACM (CACM)*, pages 31–35, June 2003.
14. Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan, and Geetika Agrawal. The BEA/XQRL Streaming XQuery Processor. In *VLDB 2003: Proceedings of 29th International Conference on Very Large Data Bases, September 9–12, 2003, Berlin, Germany*, pages 997–1008, 2003.
15. Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure Working Group, Global Grid Forum*, June 2002.
16. Johannes Gehrke, Flip Korn, and Divesh Srivastava. On Computing Correlated Aggregates over Continual Data Streams. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 13–24, 2001.
17. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams. In *Proceedings of 2000 Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 359–366. ACM Press, 2000.
18. Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
19. Michael H. Kay. Saxon: The xslt and xquery processor. http://saxon.sourceforge.net/.
20. C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proceedings of the 30th International Conference on Very Large Data Bases*, 2004.
21. 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.
22. B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.
23. D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In *Proceedings of ICDE 2003, Psoter Session*, 2003.
24. Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 431–442, 2003.
25. A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R.Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 974–985, 2002.
26. William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of Conference on Compiler Construction (CC)*, April 2002.
27. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.