Efficient Evaluation of XQuery over Streaming Data

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 XML and emergence of streaming data model, processing queries over streaming XML has become an important topic. This paper presents a new framework and a set of techniques for processing XQuery over streaming data. As compared to the existing work on supporting XPath/XQuery over data streams, we make the following three contributions:

1. We propose a series of optimizations which transform XQuery queries so that they can be correctly executed with a single pass on the dataset.

2. We present a methodology for determining when an XQuery query, possibly after the transformations we introduce, can be correctly executed with only a single pass on the dataset.

3. We describe a code generation approach which can handle XQuery queries with user-defined aggregates, including recursive functions. We aggressively use static analysis and generate executable code, i.e., do not require a query plan to be interpreted at runtime.

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

1 Introduction

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. At the same time, a new model of data processing has also emerged in the database community. In this data model, data arrives in the form of continuous streams, usually from a data collection instruments or a long running computer simulation. The data needs to be analyzed in real-time, and using only a single pass on the data. Many important applications classes, like protecting network security, monitoring critical infrastructure, analyzing stock and business data, monitoring climate and environment involve analysis of streaming data [23, 6, 45].

With these two developments, processing and querying XML streams has become an important topic. We believe that there are two other important trends which also contribute to the need for processing XML streams. The first is related to distributed and grid-based processing. There have been rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Lambda Rail (NLR) effort. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster, and streaming model is gaining popularity. At the same time, XML has been widely adapted in web-based [19], distributed [9], and grid computing [21]. The second development is the popularity of virtual XML, where XML is used as a logical view to low-level data formats, such as flat-file Bioinformatics data [40] or network data1.

To query and process (virtual) XML data streams, XQuery designed by W3C [8] can be an ideal language, because of its declarative nature and powerful features.

---

1 Please see http://www.galaxquery.org/slides/xsym2004.pdf
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.

Currently, there is a limited work on query evaluation on XML streams, and most of this handles only XPath fragments [39, 38, 15]. Compared with XPath, XQuery is significantly more expressive, and therefore, more challenging to handle. Some techniques have been proposed for processing XQuery queries over streaming data [33, 29]. Transducer networks has been used in XSM [33] to handle a small subset of XQuery, in which only join and node creation operations are allowed. Flux [29], on the other hand, uses static analysis for optimize buffer size. There are two important limitations in both these efforts. First, neither of them can handle aggregation functions, which we believe can be critical in specifying the type of analysis that is often done on streaming data. Second, neither of them have presented query transformations techniques to reduce the number of traversals, which again can be important for enabling a larger number of queries to be executed correctly on streaming data.

This paper presents a new framework and a set of techniques for processing XQuery over streaming data. As compared to the existing work on supporting XPath/XQuery over data streams, we make the following contributions:

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

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

3) To the best of our knowledge, our system is the only XQuery engine on streaming data that support aggregates, including user defined recursive functions. Based on an intermediate representation called Generalized Nested Loops (GNL), we propose low-level transformation techniques, such as aggregation rewriting and recursion analysis, to optimize aggregations.

4) We propose a new technique to generate efficient streaming code, using our GNL representation.

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

The rest of the paper is organized as follows. A motivating application is described in Section 2. The overall problem is described in Section 3. Our high-level 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. Low level analysis and code generation are presented in Section 5. Experimental evaluation is presented in Section 6. We compare our work with related research efforts in Section 7 and conclude in Section 8.

2 A Motivating Application

```xml
unordered(
  for $i$ in ($minx$ to $maxx$)
    let $Sp$ := /stream/data/pixel
    where(( $Sp/x = $i) and ($Sp/y = $j ))
    return
      <pixel>
        <latitute> {$i} </latitute>
        <longtitute> {$j} </longtitute>
        <summary> {accumulate($Sp) } </summary>
      </pixel>
)

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

Figure 1: Satellite Data Processing Expressed in XQuery

In this section, we describe an application we refer to as satellite data processing [11]. 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
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, as described by Babu and Widom [7].

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

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

and the output stream is

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

A pipeline operator $f$ has the property:

\[ y_i = \text{g}(x_{h(i)}, b) \]

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

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

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

An example of a blocking operator is the sort operation.

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

\[ |\text{Output}(f)| \ll |\text{Input}(f)| \quad (1) \]

\[ \text{g}(x_1, x_2, \ldots, x_n) = \text{g}_i(\text{g}(x_1, x_2, \ldots, x_{n-1}), x_n) \quad (2) \]

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:

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

The first condition is straightforward. Let us consider the second condition. When the final output of a progressive blocking operator $f_1$ is referred by another operator $f_2$, which is either a pipeline or a progressive blocking operator, $f_2$ must wait until the computation of $f_1$ finishes. This blocks the pipeline or progressive blocking computation $f_2$ defines. Queries that satisfy this propriety are referred to as correlated aggregates [22], which in most cases can only be evaluated approximately with a single pass.

The dependence between blocking operators and pipeline or progressive blocking operators that prevents a query from being evaluated in a single pass can either be a control dependence or a data dependence. Control dependence between operators is involved in correlated sub-queries, where the result of a sub-query is used as a predicate to filter the tuple selection. Many research efforts have focused on de-correlating such queries using various unnesting techniques, in the context of both relational [28, 37] and object-oriented databases [32, 13, 16].

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

**Query 1:**

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

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

In the rest of this paper, we propose to use static analysis at the expression level to model the data flow and dependence information for XQuery. Static analysis will be used for guiding query transformation, as well as efficient code generation for query evaluation. Static analysis techniques have extensively been used in the programming language community for optimization of imperative languages [2] and quite recently, have been successfully used for analyzing and optimizing XQuery [29, 35].

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. Low-level analysis to facilitate code generation for XQuery with user-defined aggregates and recursive functions is discussed in the next Section. Initially, we give an overview of our overall framework.

4.1 Overview

Analysis based on relational algebra has recently been proposed to characterize the memory usage of SQL queries over continuous streams [5]. Since our focus is on XQuery, we do not use the algebra approach for the following two reasons. First, as we had stated earlier, developing an algebra to fully exploit the expressive power of XQuery is hard. For example, most existing XQuery engines handle this language one expression at a time, which does not allow aggressive optimizations. Second, unlike SQL, data dependence is frequently introduced in an XQuery code through the use of binary expressions for computations. Such dependence relationships are traditionally well represented by dependence graphs in optimizing compilers. Specifically, in the next subsection, we introduce a representation called the stream data flow graph.

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

Figure 2 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.

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

**Definition 2** 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 3. \( S1 \) 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 \( v_1 \). 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 [8]. Assume there is a cycle, then...

![Figure 3: Example of Stream Data Flow Graph](image)

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 3** 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 expression. In such a case, we denote \( v_1 \rightarrow v_2 \).

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

**Definition 4** Given two variables \( v_1, v_2 \), we say that \( v_2 \) is flow dependent on \( v_1 \) if: 1) \( v_2 \) is dependent on \( v_1 \), and 2) \( v_2 \) is not aggregate dependent on \( v_1 \). In such a case, we denote \( v_1 \rightarrow v_2 \).

Let us reconsider the Figure 3. 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.

4.3.1 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:**
```xml
let $b = count(stream/pixel[x>0])
return sum(stream/pixel/y) idvi $b
```

![Diagram](image)
Figure 4: Example of Horizontal Fusion

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 4. 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 5. 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:

```plaintext
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:

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

Figure 5: Horizontal and Vertical Fusion for Query 1

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.

4.3.2 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:**

```plaintext
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 7. `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
Figure 6: Example of Vertical Fusion (Query 3) 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 5(b). The data flow graph for the Query 3 after vertical fusion is shown in Figure 6(b).

**Figure 7: Algorithm for Vertical Fusion**

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 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 \( \theta(N) \). 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 8.

**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 \( V_1 \) 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 $n$ vertices, 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 an 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. There exists a sequence node $s \in V$, 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 operation to compute $a'$, since $s$ is the only sequence node in $V$, $a'$ must be aggregate dependent on $s$. Because $a'$ uses the value of $a$, there must be a path $a, v_1, \ldots, v_k, a', a \rightarrow v_1, \ldots, v_k \rightarrow a'$. Therefore, the third condition holds true.

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

Finally, it should be noted that as with all static 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:

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

This query has a redundant predicate [5]. 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.

## 5 Low-level Analysis and Code Generation

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

For achieving efficiency and handling a general class of XQuery codes, we generate executable for a query directly, instead of decomposing the query at the operator level and interpreting the query plan. This is similar in nature to the optimized codes that are generated by a compiler according to a specified underlying architecture. In comparison, interpreted codes generally suffer in efficiency, as has been shown for many languages, for example, Matlab [3]. Furthermore, operators cannot model some features of XQuery effectively, such as recursive functions.

### 5.1 GNL Representation

As described in the previous section, we generate a stream data flow graph from a given XQuery code. However, the stream data flow graph only represents a high-level view of the dependencies among variables and expressions, while details of the processing involved are not modeled. To facilitate generation of streaming code, we introduce a representation called Generalized Nested Loops (GNL). This representation helps exploit the imperative nature of XML parsers such as SAX2. Though our current implementation has been carried out on top of SAX, our code generation techniques are more general.

#### 5.1.1 Definition of GNL

**Definition 5** A GNL $N$ is a four tuple $(I, E_p, E_c, S)$ where,

1. $I$ is the index variable bound to $N$.
2. $E_p$ is the location path of the corresponding XPath expression.
3. $E_c$ is the predicate expression of $E_p$ (if any), and
4. $S$ is the loop body, which is an ordered sequence of operations performed on any tuple bound to the index variable $I$.

The semantic meaning of a GNL is similar to a foreach loop or iterator, which iterates over tuples specified by the XPath expression. For every element in the target XPath expression $E_p$ filtered with $E_c$, it is bound to the variable $I$, and each statement $s \in S$ will be executed according to their order in $S$.

By definition, the tuple stream that a GNL operates on is specified by its path expression $E_p$, and the aggregation operations are specified by the statement sequence in its body. With such a syntax structure, code generation for

[^2]: http://www.saxproject.org
5.1.3 Aggregation Rewrite for GNLs

A GNL with an empty loop body makes Java functions triggered by SAX events becomes easier. Since the sequence node represents a traversal on the data flow graph, an aggregate node with an XPath expression, this GNL is inserted as a statement in the loop body of the GNL for the node. Furthermore, as we will describe later, low-level optimizations techniques on rewriting recursive functions and user-defined aggregates are also facilitated by the GNL representation.

5.1.2 GNL Formation

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

If there is a super-node in the data flow graph, the GNL formation process is more complex. We create a GNL $G_0$ corresponding to the main query. We also create a GNL corresponding to each sequence node in the super-node that has an associated XPath expression. The GNL corresponding to the root sequence node in the super-node is inserted as a statement in the loop body of $G_0$. Consider any other node $n$ in the super-node. We find the closest ancestor of this node that has a GNL associated with it, and denote it as $CA$. If the node $n$ has a GNL associated with it (because it is a sequence node with an XPath expression), this GNL is inserted as a statement in the loop body of the GNL for $CA$. Otherwise, the statement corresponding to the node $n$ is inserted in the loop body of the GNL for $CA$. Finally, consider any atomic node that is not in the super-node. The statement corresponding to this node is inserted in the loop body of $G_0$. Whenever multiple statements are inserted in a loop body, their sequence must be consistent with their sequence in the original query.

GNL for the Query 2 is shown in Figure 9(a).

5.1.3 Aggregation Rewrite for GNLs

After the GNL representation is generated from the stream data flow graph, aggregation functions, including those defined by users, are typically placed outside the root GNL.

This is because the formation of GNL does not follow any edge that represents aggregate dependence. Therefore, we need to recognize such aggregation functions, rewrite them into operations that can be applied a tuple at a time, and move the aggregation into the corresponding GNL. Here, we assume that any aggregation operation on a tuple stream is performed through a function, which could be user-defined or internal.

An aggregate function can be easily recognized by finding all atomic nodes that are aggregate dependent on a sequence node or a super-node. For each such node $v$, we move the corresponding function call into the GNL by using function inlining. Internal functions, such as $sum$, $count$, and $average$, can be easily rewritten in an iterative fashion. For example, we can rewrite $\text{sum}(\cdot)$ as $tmp = tmp + v$, where $tmp$ is a temporary variable and $v$ is the tuple. For a user-defined function, including recursive functions, we apply a previously developed static analysis technique to extract an associative operation from the definition of the function [30]. The basic idea is to examine the syntax tree from leaves and apply tree pattern matching to retrieve the desired sub-tree. Our algorithm can only deal with linear recursive functions.

The GNL of Query 2 after aggregation rewriting is shown in Figure 9(b).

5.2 Code Generation

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

Similar to Peng and Chawathe [39], our processing assumes that an incoming data tuple is one of the following three types: 1) $\text{startElement}(n, \text{attr})$, which is the start event for the node $n$ with attribute list $\text{attr}$, 2) $\text{character}(n)$, which is the content of the node $n$, and 3) $\text{endElement}(n)$, which is the end event for the node $n$.

5.2.1 From GNL to SAX Event Handling

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

Specifically, we use the following strategy to evaluate a GNL. For a given GNL $E$ with $E_T = /x/y/z$, when the event $\text{endElement}(z)$ is triggered, the body of $E$ is executed once. For nested GNLs $E_1$ and $E_2$, with $E_2$ nested inside $E_1$, the processing for $E_2$ is always performed before the processing for $E_1$. The code generated for evaluation of Query 2 is shown in Figure 10.
Though our goal is to process queries on the fly, certain XML elements may need to be buffered. For example, a node may be issued for output after a condition involving its children nodes is evaluated. Clearly, because buffering requires memory, we want to buffer as few elements as possible. We use a filtering and projection technique similar to the one described in [29]. However, our technique is simpler because we do not process blocking operators like join and sort. Specifically, we buffer a node with its entire subtree if it is used as output. Also, we buffer any leaf node whose value is referred to in the query. In such a case, if \( v \) is not in the subtree of a node used as output, the buffer of \( v \) will be immediately dispatched when reference of \( v \) is finished. Using the GNL representation, we check the location path \( E_p \) and filter \( E_c \) of each GNL, and only mark a node for buffering if it is referred to in the body of that GNL.

The details of buffering for the Query 2 are shown in Figure 10. In this example, no node is used as output, and buffering of \( /x \) and \( /y \) is for using their values only. Therefore, these nodes are dispatched during the handling of `endElement` event.

```java
foreach startElement (e1) {
  switch(e1, node)
  x: buffer.add(x)
  y: buffer.add(y)
}

foreach endElement (e1) {
  switch(e1, node)
  x: if (buffer.dispatch(x) > 0)
    \( v_1 = v_1 + 1 \)
  y: \( v_2 = v_2 + buffer.dispatch(y) \)
  root: { b = \( v_1 \); return b / \( v_2 \) }
}
```

Figure 10: Evaluation Code for Query 2

## 6 Experimental Results

To evaluate our implementation of the framework and the techniques presented in this paper, we conducted a series of experiments. We compared our implementation with other well known XQuery processors which are publically available. Specifically, we use Galax (Version 0.3.1) [17], Saxon (Version 8.0) [27] and Qizx/Open (Version 0.4/\( \rho \)1) [1]. All these query processors are implemented using a SAX Parser, which we believe makes the comparison reasonable. Note that our transformations can enable many queries to be executed with a single pass, whereas other systems may require multiple passes for them. Thus, an advantage of our framework is that we may allow execution with a limited memory, or on unbounded streams, while the other systems may simply fail to execute the same query in such scenarios. For our comparison here, we only focus on execution times, and not the memory requirements or the ability of a system to process query on streaming data.

We used two sets of queries for our experiments. The first set comprised the queries 1, 5, 6, 7, and 20 from the XMark benchmark set [41]. 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 [18]. 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 [26]. Each of these three applications uses recursive functions to perform aggregations. After applying our techniques and optimizations, including analysis of recursive functions, aggregate rewriting, and horizontal and vertical fusion, each of these could be processed correctly using only a single pass on the entire data stream. We generated synthetic datasets of varying sizes to evaluate performance on these applications.

The results of our experiments are shown in Figure 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.
### 7 Related Work

Currently, much of the research efforts on XQuery focuses on evaluation on disk-resident XML documents. Techniques that query XML views over relational database engines include [43, 42, 14, 44, 34] and approaches based on native XML datasets include [12, 24, 25]. Although various evaluation and optimization strategies have been proposed, they are not specially designed to handle continuous XML streams.

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 [4, 15, 10]. To deal with predicates and other features, such as closures where buffering of certain elements is necessary, transducers have been used in XSQ [39] and SPEX [38].

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 [33]. Without query transformation and rewriting, their techniques will not work on streaming data when the queries are not strictly written to execute on streaming data. In Flux [29], 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 systematically 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/XQL processor [20] 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 [5]. 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.

Optimization of nested queries has been investigated in the context of SQL [28, 37] and Object-Oriented database programming languages [32, 13, 16]. However, XQuery
is a more powerful and expressive language, and therefore, it involves additional challenges for efficient evaluation and translation. The merge of path navigations is reported in the Lrell query language for XML [36], but is limited to paths bound to the same variable. In comparison, our loop fusion techniques are based on data dependence information and are much more powerful. Horizontal loop fusion technique is used in [31] to separate lists of XML-QL, but is quite different from the fusion optimizations we have developed.

8 Conclusions

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

References