Extending XQuery with Window Functions

Irina Botan Peter M. Fischer Daniela Florescu[◇]
Donald Kossmann Tim Kraska Rokas Tamosevicius

ETH Zurich Oracle[◇]
{firstname.lastname}@inf.ethz.ch dana.florescu@oracle.com

ABSTRACT

This paper presents two extensions for XQuery. The first extension allows the definition and processing of different kinds of windows over an input sequence; i.e., tumbling, sliding, and landmark windows. The second extension extends the XQuery data model (XDM) to support infinite sequences. This extension makes it possible to use XQuery as a language for continuous queries. Both extensions have been integrated into a Java-based open source XQuery engine. This paper gives details of this implementation and presents the results of running the Linear Road benchmark on the extended XQuery engine.

1. INTRODUCTION

XML has had a breakthrough success as a data format for three kinds of data: (a) communication data, (b) meta data, and (c) documents. Communication data has been the first big success story: XML is used as the format to exchange data in Web Services or to represent streams as RSS or Atom feeds. Examples for meta data represented in XML are configuration files, schemas (e.g., XML schemas, the Vista file system), design specifications (e.g., Eclipse's XMI), interface descriptions (e.g., WSDL), or logs (e.g., Web logs). In the document world, big vendors such as Sun (OpenOffice) and Microsoft (MS Office) have also moved towards representing their data in XML. XHTML and RSS blogs are further examples that show the success of XML in this domain.

With an increasing amount of XML data, there has been an increased demand to find the right paradigms to process this data. Arguably, XQuery is the most promising programming language for this purpose [6]. XQuery 1.0 is a recommendation of the W3C. So far, almost fifty XQuery implementations are advertised on the W3C web pages, including implementations from all major database vendors and several open source offerings.

Even though XQuery 1.0 is extremely powerful (it is Turing-complete), it lacks important functionality. In particular, XQuery 1.0 lacks support for window queries and continuous queries. This omission is somewhat disappointing because exactly this support is needed to process the main targets of XML: Communication data, meta data, and documents. Communication data is often represented as a stream of XML items and for many applications it is important to detect certain patterns in that stream: A credit card company, for instance, might be interested to learn if a credit card is used particularly often during one week as compared to other weeks. Implementing this audit involves a continuous window query. The analysis of a Web log, as another example, involves the identification and processing of user sessions, which again involves a window query. Document formatting requires operations such as pagination to enable users to browse through the documents a page at a time; again, pagination is best implemented using a window query.

Both window queries and continuous queries have been studied extensively in the SQL world; proposals for SQL extensions are described in, e.g., [22, 4, 7, 27]. That work, however, is not applicable in the XML world. The first and obvious reason is that SQL is not appropriate to process XML data. It can neither directly read XML data, nor can SQL generate XML data if the output is required to be XML (e.g., an RSS feed or a new paginated document). Another reason is that the SQL extensions are not expressive enough to solve all use cases mentioned above, even if all data were relational (e.g., CSV). The SQL extensions have been specifically tuned for particular streaming applications and support only simple window definitions based on time or window size.

The purpose of this work is to extend XQuery in order to support window queries and continuous queries. The goal is to have a powerful extension that is appropriate for all use cases, including the *classic* streaming applications for which the SQL extensions were designed and the more *progressive* use cases of the XML world (i.e., RSS feeds and document management). At the same time, the performance should be comparable to the performance of continuous SQL queries: There should not be a performance penalty for using XQuery. This work will also be submitted to the W3C for consideration of the emerging XQuery 1.1 recommendation (due in late 2007). Indeed, window queries have been identified as one of the requirements for XQuery 1.1 [11].

Obviously, our work is based on several ideas and the experience gained in the SQL world from processing data streams. Nevertheless, there are important differences to the SQL data stream approach. In fact, it is easier to extend XQuery because the XQuery data model (XDM), which is based on *sequences of items* [12], is already a good match to

VLDB '07, September 23-28, 2007, Vienna, Austria.
.

represent data streams and windows. As a result, the proposed extensions compose nicely with all existing XQuery constructs and all existing XQuery optimization techniques remain relevant. In contrast, extending SQL for window queries involves a more drastic extension of the relational data model and a great deal of effort in the SQL world has been spent on defining the right mappings for these extensions. As an example, CQL [4] defines specific operators that map between streams and relations.

In summary, this paper makes the following contributions:

- *Window Queries:* The syntax and semantics of a new `FORSEQ` clause in order to define and process complex windows using XQuery.
- *Continuous Queries:* A simple extension to the XQuery data model (XDM) in order to process infinite data streams and use XQuery for continuous queries.
- *Use Cases:* A series of examples that demonstrate the expressiveness of the proposed extensions.
- *Implementation Design:* Show that the extensions can be implemented and integrated with little effort into existing XQuery engines and that simple optimization techniques are applicable in order to get good performance.
- *Linear Road Benchmark:* The results of running the Linear Road benchmark [5] on top of an open source XQuery engine which was enhanced with the proposed extensions. The benchmark results confirm that the proposed XQuery extensions can be implemented efficiently.

The remainder of this paper is organized as follows: Section 2 gives a motivating example. Section 3 presents the proposed syntax and semantics of the new `FORSEQ` clause used to define windows in XQuery. Section 4 proposes an extension to the XQuery data model in order to define continuous XQuery expressions. Section 5 lists several examples that demonstrate the expressive power of the proposed extensions. Section 6 explains how to extend an existing XQuery engine and optimize XQuery window expressions. Section 7 presents the results of running the Linear Road benchmark on an extended XQuery engine. Section 8 gives an overview of related work. Section 9 contains conclusions and suggests avenues for future work.

2. USAGE SCENARIOS

2.1 Motivating Example

The following simple example illustrates the need for an XQuery extension. It involves a blog with RSS items of the following form:

```
<rss:item>
... <rss:author>...</rss:author> ...
</rss:item>
```

Given such a blog, the goal is to find all *annoying authors* who have posted three consecutive items in the RSS feed. Using XQuery 1.0, this query can be formulated as shown in Figure 1. This query involves a three-way self join which is not only tedious to specify but also difficult to optimize. In contrast, Figure 2 shows this query using the proposed `FORSEQ` clause. This clause partitions the blog into sequences of postings of the same author (i.e., windows) and iterates over these windows (Lines 1-4 of Figure 2). If a window contains three or more postings, then the author of this window of postings is annoying and the author is returned (Lines 5 and 6). The syntax and semantics of the `FORSEQ` clause are

```
for $first at $i in $rssfeed
let $second := $rssfeed[$i+1],
let $third := $rssfeed[$i+2]
where ($first/author eq $second/author) and
   ($first/author eq $third/author)
return $first/author
```

Figure 1: Annoying Authors: XQuery 1.0

```
forseq $w in $rssfeed tumbling window
   start curItem $first when fn:true()
   end  nextItem $lookAhead when
      $first/author ne $lookAhead/author
where count($w) ge 3
return $w[1]/author
```

Figure 2: Annoying Authors: Extended XQuery

defined in detail in Section 3 and need not be understood at this point. For the moment, it is only important to observe that this query is straightforward to implement and can be executed in linear time or better, if the right indexes are available. Furthermore, the definition of this query can easily be modified if the definition of *annoying author* is changed from, say, three to five consecutive postings. In comparison, additional self-joins must be implemented in XQuery 1.0 in order to implement this change. [1]

2.2 Other Applications

The management of RSS feeds is one application that drove the design of the proposed XQuery extensions. There are several other areas; the following is a non-exhaustive list of further application scenarios:

- *Web Log Auditing:* In this scenario, a window contains all the actions of a user in a session (from login to logout). The analysis of a Web log involves, for example, the computation of the average number of clicks until a certain popular function is found. Security audits and market-basket analyses can also be carried out on user sessions.
- *Financial Data Streams:* Window queries can be used in order to carry out fraud detection, algorithmic trading and finding opportunities for arbitrage deals by computing call-put parities [13].
- *Social Games / Gates:* An RFID reader at a gate keeps track of the people that enter and exit a building. People are informed if their friends are already in the building when they themselves access the building.
- *Sensor Networks:* Window queries are used in order to carry out data cleaning. For instance, the average of the last five measurements (possibly, disregarding the minimum and maximum) is reported, rather than reporting each individual measurement [18].
- *Document Management:* Different text elements (e.g., paragraphs, tables, figures) are grouped into pages. In the index, page sequences such as 1, 2, 3, 4, 7 are reformatted into 1-4, 7 [20].

We compiled around sixty different use cases in these application areas in a separate document [13]. All these examples have in common that they cannot be implemented well using the current Version 1.0 of XQuery without support for windows. Furthermore, many examples of [13] cannot be processed using SQL, even considering the latest extensions

[1] In fact, the two queries of Figures 1 and 2 are not equivalent. If an author posts four consecutive postings, this author is returned twice in the expression of Figure 1, whereas that author is returned only once in Figure 2.

$$
\begin{aligned}
FLWORExpr &::= (\textbf{ForseqClause}|ForClause|LetClause)+\ WhereClause?\ OrderByClause?\ \text{"return"}\ ExprSingle\\
ForseqClause &::= \text{"forseq"}\ \text{"\$"}VarName\ TypeDeclaration?\ \text{"in"}\ ExprSingle\ WindowType?\\
&\quad (\text{","}\text{"\$"}VarName\ TypeDeclaration?\ \text{"in"}\ ExprSingle\ WindowType?)*\\
WindowType &::= (\text{"tumbling window"}|\text{"sliding window"}|\text{"landmark window"})\ StartExpr\ EndExpr\\
StartExpr &::= \text{"start"}\ WindowVars?\ \text{"when"}\ ExprSingle\\
EndExpr &::= \text{"force"}?\ \text{"end"}\ WindowVars?\ \text{"when"}\ (\text{"newstart"}|ExprSingle)\\
WindowVars &::= (\text{"position"}|\text{"curItem"}|\text{"prevItem"}|\text{"nextItem"})\ \text{"\$"}VarName\ TypeDeclaration?\\
&\quad (\text{","}\ (\text{"position"}|\text{"curItem"}|\text{"prevItem"}|\text{"nextItem"})\ \text{"\$"}VarName\ TypeDeclaration?)*
\end{aligned}
$$

Figure 3: Grammar of Extended FLWOR Expression

proposed in [22, 4, 7, 27] because these examples require powerful constructs in order to define window boundaries. Most of these use cases involve other operators such as negation, existential and universal quantification, aggregation, correlation, joins, and transformation in addition to window functions. XQuery already supports all these operators which makes XQuery a natural candidate to extend, rather than inventing a new language from scratch in order to address these applications.

3. FORSEQ CLAUSE

3.1 Basic Idea

Figure 2 gives an example of the `FORSEQ` clause. The `FORSEQ` clause is an extension of the famous FLWOR expressions of XQuery. It is freely composable with other `FOR`, `LET`, and `FORSEQ` clauses. Furthermore, FLWOR expressions that involve a `FORSEQ` clause can have an optional `WHERE` and/or `ORDER BY` clause and must have a `RETURN` clause, just as any other FLWOR expression. A complete grammar of the extended FLWOR expression is given in Figure 3.

Like the `FOR` clause, the `FORSEQ` clause iterates over an input sequence and binds a variable with every iteration. The difference is that the `FORSEQ` clause binds the variable to a *sub-sequence* (aka window) of the input sequence in each iteration, whereas the `FOR` clause binds the variable to an *item* of the input sequence. To which sub-sequences the variable is bound is determined by additional clauses. The additional `TUMBLING WINDOW`, `START`, and `END` clauses of Figure 2, for instance, specify that `$w` is bound to each consecutive sub-sequence of postings by the same author. In that example, the window boundaries are defined by the change of author in postings in the `WHEN` clause of the `END` clause (details of the semantics are given in the next subsection).

The running variable of the `FORSEQ` clause (`$w` in the example) can be used in any expression of the `WHERE`, `ORDER BY`, `RETURN` clauses or in expressions of nested `FOR`, `LET`, and `FORSEQ` clauses. The only requirement is that those expressions must operate on sequences (rather than individual items or atomic values) as input. In Figure 2, for example, the `count` function is applied to `$w` in the `WHERE` clause in order to determine whether `$w` is bound to a series of postings of an *annoying author* (three or more postings).

As shown in Figure 3, FLWOR expressions with a `FORSEQ` clause can involve an `ORDER BY` clause, just like any other FLWOR expression. Such an `ORDER BY` clause specifies in which order the sub-sequences (aka windows) are bound to the running variable. By default, and in the absence of an `ORDER BY` clause, the windows are bound in ascending order of the position of the *last* item of a window. If two (overlapping) windows end in the same item, then their order is implementation-defined. For instance, *annoying authors* in the example of Figure 2 are returned in the order in which they made annoying postings. This policy naturally extends the order in which the `FOR` clause orders the bindings of its input variable in the absence of an `ORDER BY` clause.

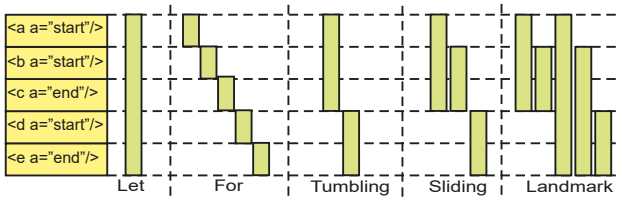

Figure 4: Window Types

The `FORSEQ` clause does not involve an extension or modification of the XQuery data model (XDM) [12]. Binding variables to sequences is naturally supported by XDM. As a result, the `FORSEQ` clause is fully composable with all other XQuery expressions and no other language adjustments need to be made. There is no catch here. In contrast, extending SQL with windows involves an extension to the relational data model and, as mentioned in the introduction, a great deal of effort has been invested into defining the exact semantics of such window operations in such an extended relational data model.

Furthermore, the XQuery type system does not need to be extended, and static typing for the `FORSEQ` clause is straightforward. To give a simple example, if the static type of the input is *string**, then the static type of the running variable is *string+*. The "+" quantifier is used because the running variable is never bound to the empty sequence. To give a more complex example, if the static type of the input sequence is *string*, integer** (i.e., a sequence of strings followed by a sequence of integers), then the static type of the running variable is: *(string+,integer* | string*,integer+)*; i.e., a sequence of strings, a sequence of integers, or a sequence of strings followed by integers. (Similarly, simple rules apply to the other kinds of variables that can be bound by the `FORSEQ` clause.)

3.2 Types of Windows

Previous work on extending SQL to support windows has identified different kinds of windows; i.e., tumbling windows, sliding windows, and landmark windows [15]. Figure 4 shows examples of these three types of windows; as a reference, Figure 4 also shows how the traditional `FOR` and `LET` clauses of XQuery work. The three types of windows differ in the way windows overlap: tumbling windows do not overlap; sliding windows overlap, but have disjoint first items;

and landmark windows can overlap in any way. Following the experiences made with SQL, we propose to support these three kinds of windows in XQuery, too. This subsection describes how the FORSEQ clause can be used to support these kinds of windows.

Furthermore, previous work on windows for SQL proposed alternative ways to define the window boundaries (start and end of a window). Here, all published SQL extensions [22, 4, 7, 27] propose to define windows based on size (i.e., number of items) or duration (time span between the arrival of the first and last item). Our proposal for XQuery is more general and is based on using predicates in order to define window boundaries. Size and time constraints can easily be expressed in such a predicate-based approach (examples are given in the remainder of this paper). Furthermore, more complex conditions which involve any property of an item (e.g., the author of a posting in a blog) can be expressed in our proposal. One of the consequences of having predicate-based window boundaries is that the union of all windows does not necessarily cover the whole input sequence; that is, it is possible that an input item is not part of any window.

3.2.1 Tumbling Windows

The first kind of window supported by the FORSEQ clause is a so-called *tumbling window* [25]. Tumbling windows partition the input sequence into disjoint sub-sequences, as shown in Figure 4. An example of a query that involves a tumbling window is given in Figure 2 in which each window is a consecutive sequence of blog postings of a particular author. Tumbling windows are indicated by the TUMBLING WINDOW keyword as part of the WindowType declaration in the FORSEQ clause (Figure 3).

The boundaries of a tumbling window are defined by (mandatory) START and END clauses. These clauses involve a WHEN clause which specifies a predicate. Intuitively, the WHEN condition of a START clause specifies when a window should start. For each item in the sequence this clause is checked for a match. Technically, a match exists if the effective Boolean value (EBV) [10] of the WHEN condition evaluates to true. As long as no item matches, no window is started and the input items are ignored. Thus, it is possible that certain items are not part of any window. Once an item matches the WHEN condition of the START clause, a new window is *opened* and the matching item is the first item of that window. At this point, the WHEN condition of the END clause is evaluated for each item, including the first item. Again, technically speaking, the EBV is computed. If an item matches the END condition, that item is the last item of the window.

Any XQuery expression can be used in a WHEN clause (Figure 3), including expressions that involve existential quantification (on multiple sub-elements) or nested FLWOR expressions (possibly with FORSEQ). The semantics of the START and END clauses for tumbling windows can best be shown using the automaton depicted in Figure 5. The condition of the START clause is not checked for an open window. A window is only closed when its END condition is fulfilled or at the end of the input sequence.

To give two simple examples, the FOR clause of XQuery can be simulated with a FORSEQ clause as follows:

```
forseq $w in $seq tumbling window
   start when fn:true()
   end when fn:true()  ...
```

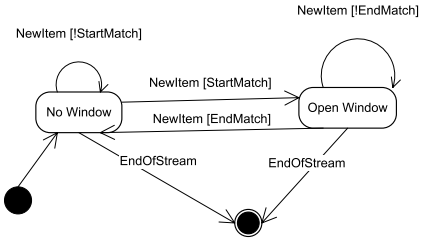

Figure 5: Window Automaton

That is, each item opens and immediately closes a new window (both START and END conditions are set to true) so that each item represents a separate window. The LET clause can be simulated with a FORSEQ clause as follows:

```
forseq $w in $seq tumbling window
   start when fn:true()
   end when fn:false() ...
```

The first item of the input sequence opens a new window (START condition is true) and this window is closed at the end of the input sequence. In other words, the whole input sequence is bound to the running variable as a single window.

In order to specify more complex predicates, both the START and the END clause allow the binding of new variables. The first kind of variable identifies the position of a potential first item (in the START clause) or last item (in the END clause), respectively. For instance, the following FORSEQ clause partitions the input sequence ($seq) into windows of size three; the last window might be smaller:

```
forseq $x in $seq tumbling window
   start position $i when fn:true()
   end position $j when $j-$i eq 2 ...
```

For each window, $i is bound to the position of the first item of the window in the input sequence; i.e., $i is 1 for the first window, 4 for the second window, and so on. Correspondingly, $j is bound to the position of the last item of a window as soon as that item has been identified; i.e., $j is 3 for the first window, 6 for the second window, and so on. In this example, $j might be bound to an integer that is not a multiple of three for the last window at the end of the input sequence.

Both $i and $j can be used in the WHEN expression of the END clause. Naturally, only variables bound by the START clause can be used in the WHEN condition of the START clause. Furthermore, in-scope variables (e.g., $seq in the examples above) can be used in the conditions of the START and END clauses. The scope of the variables bound by the START and END clauses is the whole remainder of the FLWOR expression. For instance, $i and $j could be used in the WHERE, RETURN and ORDER BY clauses or in any nested FOR, LET, or FORSEQ clauses in the previous example.

In addition to positional variables, variables that refer to the previous (*prevItem*), current (*currItem*), and next items (*nextItem*) of the input sequence can be bound in the START and END clause. In the expression of Figure 2, for instance, the END clause binds variable $lookAhead to the item that comes after the last item of the current window (i.e., the first item of the next window). These extensions are syntactic sugar because these three kinds of variables can be simulated

using positional variables; e.g., `end nextItem $lookAheand when $lookAheand ...` is equivalent to `end position $j when $seq[$j+1] ...` In both cases, an out-of-scope binding (at the end of the input sequence) is bound to the empty sequence.

3.2.2 Sliding and Landmark Windows

In the SQL world, several different kinds of windows were identified and found useful in practice. In addition to tumbling windows, so-called sliding and landmark windows are needed in many applications. In contrast to tumbling windows, both sliding and landmark windows can overlap. The difference between sliding and landmark windows is that two sliding windows never have the first item in common, whereas landmark windows do not have such a constraint (Figure 4). A more formal definition of sliding and landmark windows is given in [25].

Based on this experience, the `FORSEQ` clause also supports sliding and landmark windows. As shown in Figure 3, only the `TUMBLING WINDOW` keyword needs to be replaced in the syntax. Again, (mandatory) `START` and `END` clauses specify the window boundaries. The semantics are analogous to the semantics of the `START` and `END` clauses of a tumbling window (Figure 5). The important difference is that each item potentially opens one (for sliding windows) or several new windows (for landmark windows) so that conceptually, several automata need to be maintained at the same time. For space constraints, we cannot give the full formal semantics in this paper. Section 5 contains examples that illustrate these kinds of windows.

3.3 General Sub-sequences

In its most general form, the `FORSEQ` clause takes no additional clauses; i.e., no specification of the window type and no `START` and `END` clauses. In this case, the syntax is as follows (Figure 3):

```
forseq $w in $seq ...
```

This general version of the `FORSEQ` clause iterates over all possible sub-sequences of the input sequence. These subsequences are not necessarily consecutive. For example, if the input sequence contains the items (a, b, c), then the general `FORSEQ` carries out seven iterations ($2^n - 1$, with n the size of the input sequence), thereby binding the running variable to the following sub-sequences: (a), (a,b), (b), (a,b,c), (a,c), (b,c), and (c). Again, the sequences are ordered by the position of their last item (Section 3.1); i.e., the (a) sequence comes before the sequences that end with a "b" which in turn come before the sequences that end with a "c". Again, the running variable is never bound to the empty sequence.

This general `FORSEQ` clause is the most powerful variant. Landmark, sliding, and tumbling windows can be seen as special cases of this general `FORSEQ`. We propose to use special syntax for these three kinds of windows because use cases that need these three types of windows are frequent in practice [13]. Furthermore, the general `FORSEQ` clause is difficult to optimize. Use cases for which landmark, sliding, and tumbling windows are not sufficient, are given in [29] for RFID data management. In those use cases, regular expressions are needed in order to find patterns in the input stream. Such queries can be implemented using the general `FORSEQ` clause by specifying the relevant patterns (i.e., regular expressions) in the `WHERE` clause of the general `FORSEQ` expression.

3.4 Syntactic Sugar

There are use cases which benefit from additional syntactic sugar. The following paragraph presents such syntactic sugar.

3.4.1 End of Sequence

As mentioned in Section 3.2.1, by default the condition of the `END` clause is always met at the end of a sequence. That is, the last window will be considered even if its last item does not match the `END` condition. In order to specify that the last window should only be considered if its last item indeed matches the `END` condition, the `END` clause can be annotated with a special keyword `FORCE` (Figure 3). The `FORCE` keyword is syntactic sugar because the last window could also be filtered out by repeating the `END` condition in the `WHERE` clause.

3.4.2 NEWSTART

There are several use cases in which the `START` condition should implicitly define the end of a window. For example, the day of a person starts every morning when the person's alarm clock rings. Implicitly, this event ends the previous day, even though it is not possible to concretely identify a condition that ends the day. In order to implement such use cases, the `WHEN` condition of the `END` clause can be defined as `NEWSTART`. As a result, the `START` condition (rather than the `END` condition) is checked for each open window in order to determine when a window should be closed. Again, the `NEWSTART` option is syntactic sugar and avoids that the condition of the `START` clause is replicated in the `END` clause.

3.5 Summary

Figure 3 gives the complete grammar for the proposed extension of XQuery's FLWOR expression with an additional `FORSEQ` clause. Since there are many corner cases, the grammar looks complicated at first glance. However, the basic idea of the `FORSEQ` clause is simple. `FORSEQ` iterates over an input sequence, thereby binding a sub-sequence of the input sequence to its running variable in each iteration. Additional clauses specify the kind of windows. Furthermore, predicates in the `START` and `END` clauses specify the window boundaries. This mechanism is powerful and sufficient for a broad spectrum of use cases [13]. We are not aware of any use case that has been addressed in the literature on window queries that cannot be implemented in this way.

Obviously, there are many ways to extend XQuery in order to support window queries. In addition to its expressive power and generality, the proposed `FORSEQ` clause has two additional advantages. First, it composes well with other `FOR, LET`, and `FORSEQ` clauses as part of a FLWOR expression. Furthermore, any kind of XQuery expression can be used in the conditions of the `START` and `END` clauses, including nested `FORSEQ` clauses. Second, the `FORSEQ` clause requires no extension to the XQuery data model (XDM). As a result, the existing semantics of XQuery functions need not be modified. Furthermore, this feature enables full composability and optimizability of expressions with `FORSEQ`.

4. CONTINUOUS XQUERY

The second extension proposed in this paper makes XQuery a candidate language to specify continuous queries on potentially infinite data streams. In fact, this extension is orthogonal to the first extension, the `FORSEQ` clause: Both extensions are useful independently, although we believe that they will often be used together in practice.

The proposal is to extend the XQuery data model (XDM) [12] to support infinite sequences as legal instances of XDM. As a result, XQuery expressions can take an infinite sequence as input. Likewise, XQuery expressions can produce infinite sequences as output. A simple example illustrates this extension. A temperature sensor in an ice cream warehouse produces measurements of the following form every minute: `<temp>-8</temp>`. Whenever a temperature of ten degrees or higher is measured, an alarm should be raised. If the stream of temperature measurements is bound to variable `$s`, this alarm can be implemented using the following (continuous) XQuery expression:

```
declare variable $s as (temp)** external;
for $m in $s where $m ge 10
return <alarm> { $m } </alarm>
```

In this example, variable `$s` is declared to be an external variable that contains a potentially infinite sequence of temperature measurements (indicated by the two asterisks). Since `$s` is bound to a (potentially) infinite sequence, this expression is illegal in XQuery 1.0 because the input is not a legal instance of the XQuery 1.0 data model. Intuitively, however, it should be clear what this continuous query does: whenever a temperature above 10 is encountered, an alarm is raised. The input sequence of the query is infinite and so is the output sequence.

Extending the data model of a query language is a critical step because it involves refining the semantics of all constructs of the query language for the new kind of input data. Fortunately, this particular extension of XDM for infinite sequences is straightforward to implement in XQuery. The idea is to extend the semantics of non-blocking functions (e.g., for, forseq, let, distinct-values, all path expressions) for infinite input sequences and to specify that these non-blocking functions (potentially) produce infinite output. Other non-blocking functions such as retrieving the ith element (for some integer i) are also defined on infinite input sequences, but generate finite output sequences. Blocking functions (e.g., order by, last, count, some) are not defined on infinite sequences; if they are invoked on (potentially) infinite sequences, then an error is raised. Such violations can always be detected statically (i.e., at compile-time). For instance, the following XQuery expression would not compile because the `fn:max()` function is a blocking function that cannot be applied to an infinite sequence:

```
declare variable $s as (temp)** external;
fn:max($s)
```

Extending XDM does not involve an extension of the XQuery type system. *(temp)*** is the same type as *(temp)**. The two asterisks are just an annotation to indicate that the input is potentially infinite. These annotations (and corresponding annotations of functions in the XQuery function library) are used in the data flow analysis of the compiler in order to statically detect the appplication of a blocking function on an infinite sequence (Section 6).

A frequent example in which the `FORSEQ` clause and this extension for continuous query are combined, is the computation of moving averages. Moving averages are useful in, e.g., sensor networks as described in Section 2.2: Rather than reporting the current measurement, an average of the current and the last four measurements is reported for every new measurement. Moving averages can be expressed as follows:

```
declare variable $seq as (xs:int)** external;
forseq $w in $seq sliding window
   start position $s when fn:true()
   end position $e when $e - $s eq 4
return fn:avg($w)
```

5. EXAMPLES

This section contains four examples which demonstrate the expressive power of the `FORSEQ` clause and continuous XQuery processing. These examples are inspired by applications on Web log auditing, financial data management, building / gate control, and sensor networks (Section 2.2). A more comprehensive set of examples from these application areas and real customer use cases can be found in [13].

5.1 Web Log Analysis

The first example involves the analysis of a log of a (Web-based) application. The log is a sequence of entries. Among others, each entry contains a timestamp (`tstamp` element) and the operation (`op`) carried out by the user. In order to determine the user activity (number of *login* operations per hour), the following query can be used:

```
declare variable $weblog as (entry)* external;
forseq $w in $weblog tumbling window
   start curItem $s when fn:true()
   end nextItem $e when
      $e/tstamp - $s/tstamp gt 'PT1H'
return fn:count($w[op eq "login"])
```

This query involves a time-based tumbling window. The XQuery 1.0 recommendation supports the subtraction of timestamps, and 'PT1H' is the ISO (and XQuery 1.0) way to represent a time duration of one hour. This query also works on an infinite Web log for online monitoring of the log because the `FORSEQ` clause is non-blocking.

5.2 Stock Ticker

The second example shows how `FORSEQ` can be used in order to monitor an (infinite) stock ticker. For this example, it is assumed that the input is an infinite sequence of (stock) ticks; each stock tick contains the symbol of the stock (e.g., "YHOO"), the price of that stock, and a timestamp. The stock ticks are ordered by time. The query detects whenever a stock has gained more than ten percent in one hour.

```
declare variable $ticker as (tick)** external;
forseq $w in $ticker sliding window
   start curItem $f when fn:true()
   end   curItem $l when $l/price ge $f/price * 1.1
            and $l/symbol eq $f/symbol
where $f/tstamp - $l/tstamp le 'PT1H'
return $l
```

This query uses sliding windows in order to detect all possible sequences of ticks in which the price of the last tick is at

least ten percent higher than the price of the first tick, for the same stock symbol. The WHERE clause checks whether this increase in price was within one hour.

5.3 Timeouts

A requirement in some monitoring applications is the definition of timeouts. For example, a doctor should be notified if the blood pressure of a patient does not drop significantly ten minutes after a certain medication has been given. As another example, a supervisor should react when a firefighter enters a burning building and stays longer than one hour. In order to implement the firefighter example, two data streams are needed. The first stream records events of the following form: <event person = ‘‘name’’ direction=’’in/out’’ tstamp=’’timestamp’’/>. The second stream is a heartbeat of the form: <tick tstamp=’’timestamp’’/>. This heartbeat could be generated by a system-defined function of the XQuery engine.

```
declare variable $evts as (event)** external;
declare variable $heartb as (tick)** external;
forseq $w in fn:union($evts, $heartb) sliding window
   start $curItem $in when $in/direction eq ‘‘in’’
   end $curItem $last
     when $last/tstamp - $in/tstamp ge ’PT1H’
      or ($last/direction = ‘‘out’’ and
       $last/person = $in/person)
where $last/direction neq ‘‘out’’
return <alarm> { $in } </alarm>
```

In this query, a new window is started whenever a firefighter enters the building. A window is closed either when a firefighter exits the building or after an hour has passed. An alarm is raised only in the latter case. This example assumes that the *fn:union()* function is implemented in a non-blocking way and consumes input continuously from all of its inputs.

5.4 Sensor State Aggregation

A frequent query pattern in sensor networks involves computing the current state of all sensors at every given point in time. If the input stream contains temperature measurements of the following form:

```
<temp id=’’1’’>10</temp>
<temp id=’’2’’>15</temp>
<temp id=’’1’’>15</temp>
```

then the output stream should contain a summary of the last measurement of each temperature sensor. That is, the output stream should look like this:

```
<values> <temp id=’’1’’>10</temp> </values>
<values> <temp id=’’1’’>10</temp>
         <temp id=’’2’’>15</temp> </values>
<values> <temp id=’’1’’>15</temp>
         <temp id=’’2’’>15</temp> </values>
```

This output stream can be generated using the following continuous query:

```
declare variable $sensors as (temp)** external;
forseq $w in $sensors landmark window
   start position $s when $s eq 1
   end when fn:true()
return <values> {
      for $id in fn:distinct-values($w/@id)
      return
           $w[@id eq $id][last()]
} </values>
```

Technically, within each window, the measurements are grouped by id of the sensor and the last measurement of each group is returned.

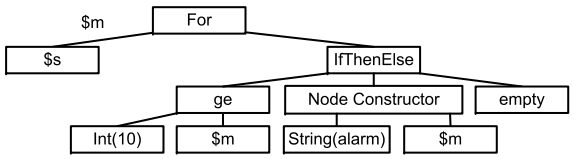

Figure 6: Example Plan (First Query of Section 4)

6. IMPLEMENTATION

This section describes how we extended the MXQuery engine[2], an existing Java-based open-source XQuery engine, in order to implement the FORSEQ clause and continuous XQuery processing. We used the extended MXQuery engine in order to validate all the use cases of [13] and run the Linear Road benchmark (Section 7).

6.1 MXQuery

The MXQuery engine was developed as part of a collaboration between Siemens and ETH Zurich. The main purpose of the MXQuery engine is to provide an XQuery implementation for small and embedded devices; in particular, in mobile phones and small gateway computers. Within Siemens, for instance, the MXQuery engine has been used as part of the Siemens Smart Home project in order to control lamps, blinds, and other devices according to personal preferences and weather information via Web services. Recently, MXQuery has also been used as a reference implementation for the XQuery Update language and XQueryP the XQuery scripting extensions.

Since MXQuery was designed for embedded systems, it has a simple and flexible design. The parser is a straightforward XQuery parser and creates an expression tree. In a functional programming language like XQuery, the expression tree plays the same role as the relational algebra expression (or operator tree) in SQL. The expression tree is normalized using the rules of the XQuery formal semantics [10]. After that, the expression tree is optimized using heuristics. (MXQuery does not have a cost-based query optimizer.) For optimization, MXQuery only implements a dozen of essential query rewrite rules such as the elimination of redundant sorts and duplicate elimination. The final step of compilation is code generation during which each expression of the expression tree is translated into an iterator that can be interpreted at run-time. As in SQL, iterators have an *open(), next(), close()* interface [16]; that is, each iterator processes its input an *item* at a time and only processes as much of its input as necessary. The iterator tree is often also called *plan*, thereby adopting the SQL query processing terminology. Figure 6 gives an example plan for the FOR query that raises an alarm when the temperature raises above ten degrees (first query of Section 4). Like Saxon [19], BEA's XQuery engine [14], and FluXQuery [21],

[2]The MXQuery engine can be downloaded via Sourceforge. MXQuery is short for MicroXQuery.

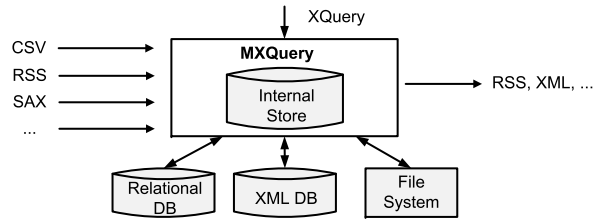

Figure 7: MXQuery Architecture

MXQuery was designed as a *streaming* XQuery engine. As shown in Figure 7, MXQuery can take input data from multiple sources; e.g., databases, the file system, or streaming data sources such as RSS streams or message queues. In order to take input from different sources, the data sources must implement the iterator API. That way, data from data sources can be processed as part of a query plan at runtime. MXQuery already has predefined iterator implementations in order to integrate SAX, DOM, StaX, and plain XML from the file system. Furthermore, MXQuery has predefined iterator implementations for CSV and relational databases. In order to access the data as part of an XQuery expression, an external XQuery variable is declared for each data source as shown in the examples of Section 5. A special Java API is used in order to bind the iterator that feeds the data from the data source to the external variable.

As shown in Figure 7, MXQuery also has an internal, built-in store in order to materialize intermediate results (e.g., for sorting of windows). In the current release of MXQuery, this store is implemented fully in main memory.

Although MXQuery was particularly designed for embedded systems, its architecture is representative. The following subsections describe how we extended the MXQuery engine in order to implement the `FORSEQ` clause and work on infinite data streams. We believe that the proposed extensions are applicable to a wide range of XQuery engines.

6.2 Plan of Attack

In order to implement the `FORSEQ` clause, the following adaptations were made to the MXQuery engine:

- The parser was extended according to the production rules of Figure 3. This extension was straightforward and needs no further explanation.
- The optimizer was extended using heuristics to rewrite `FORSEQ` expressions. These heuristics are described in Section 6.4.
- The runtime system was extended with four new iterators that implement the three different kinds of windows and the general `FORSEQ`. Furthermore, the MXQuery internal main-memory store (Figure 7) was extended in order to implement windows. These extensions are described in Section 6.3.

To support continuous queries and infinite streams, the following extensions were made:

- The parser was extended in order to deal with the new ** annotation, which declares infinite sequences.
- The data flow analyses of the compiler were extended in order to identify errors such as the application of a blocking operator (e.g., count) to a potentially infinite stream.
- The runtime system was extended in order to synchronize access to data streams and merge/split streams.

The first two extensions (parser and type system) are straightforward and can be implemented using standard techniques of compiler construction [3] and database query optimization [26]. The third extension is significantly more complex, but not specific to XQuery. For our prototype implementation, we followed as much as possible the approach taken in the Aurora project [1]. Describing the details is beyond the scope of this paper; the interested reader is referred to the literature on data stream management systems. Some features, such as persistent queues, recoverability, and security have not been implemented in MXQuery yet.

6.3 Runtime System

6.3.1 Window Iterators

The implementation of the `FORSEQ` iterators for tumbling, sliding, and landmark windows is similar to the implementation of a `FOR` iterator: All these iterators implement second-order functions which bind variables and then execute a function on those variable bindings. All XQuery engines have some sort of mechanics to implement such second-order functions and these mechanics can be leveraged for the implementation of the `FORSEQ` iterators. The MXQuery engine has similar mechanics as those described in [14] to implement second-order functions: In each iteration, a `FORSEQ` iterator binds a variable to a sequence (i.e., window) and then it executes a function on that variable binding. The function to execute is implemented as an iterator tree as shown in Figure 6 for a `FOR` iterator. This iterator tree encodes `FOR`, `LET`, and `FORSEQ` clauses (if any) as well as `WHERE` (using an `IfThenElse` iterator), `ORDER BY`, and `RETURN` clauses. In general, the implementation of second-order functions in XQuery is comparable to the implementation of nested queries in SQL.

The only difference between a `FOR` iterator and a `FORSEQ` iterator is the logic that computes the variable bindings. Obviously, the `FOR` iterator is extremely simple in this respect because it binds its running variable to every item of an input sequence individually. The `FORSEQ` iterator for tumbling windows is fairly simple, too. It scans its input sequence from the beginning to the end (or infinitely for a continuous query), thereby detecting windows as shown in Figure 5. Specifically, the effective Boolean value of the conditions of the `START` or `END` clauses are computed for every new item in order to implement the state transitions of the automaton of Figure 5. These conditions are also implemented by iterator trees. The automata for sliding and landmark windows are more complicated, but the basic mechanism is the same and straightforward to implement.

6.3.2 Window Managment

The most interesting aspect of the implementation of the `FORSEQ` iterators is main memory management and garbage collection. The items of the input sequence are materialized in main memory. Figure 8 shows a (potentially infinite) input stream. Items of the input stream that have been read and materialized in main memory are represented as squares; items of the input stream which have not been read yet are represented as ovals. The materialization of items from the input stream is carried out lazily, using the iterator model. Items are processed as they come in, thereby identifying new windows, closing existing windows, and processing the windows (i.e., evaluating the `WHERE` and `RETURN` clauses). This way, infinite streams can be processed. Full materialization is only needed if the query involves blocking operations such as `ORDER BY`, but such queries are illegal on

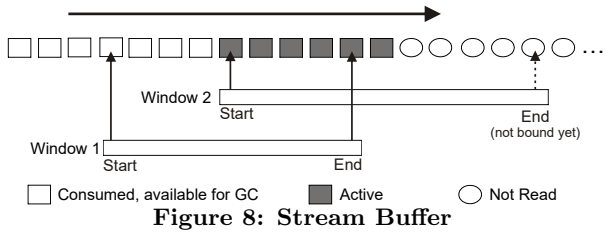

Figure 8: Stream Buffer

infinite streams (Section 4).

According to the semantics of the different types of windows, an item can be marked as *active* or *consumed* in the stream buffer. An active item is an item that is involved in at least one open window. Correspondingly, consumed items are items that are not part of any active window. An item is immediately marked as consumed if no window is open and it does not match the `START` condition of the `FORSEQ` clause (Figure 5). Otherwise, an item is marked as consumed if all windows that involve that item have been fully processed; this condition can be checked easily by keeping a *position pointer* that keeps track of the minimum first postion of all open windows. In Figure 8, consumed items are indicated as white squares; active items are indicated as colored squares. In Figure 8, Window 1 is closed whereas Window 2 is still open; as a result only the items of Window 2 are marked as active in the stream buffer. (The postion pointer is not shown in Figure 8; it marks the start of Window 2.)

Consumed items can be garbage collected. To implement memory allocation and garbage collection efficiently in Java, the stream buffer is organized as a linked list of *chunks*. (Chunking and chaining are not shown in Figure 8 for readibility.) That is, memory is allocated and de-allocated in chunks which can store several items. If all the items of a chunk are marked *consumed*, that chunk is released by dechaining it from the linked list of chunks. Furthermore, all references to closed windows are removed. At this point, there are no more live references that refer to that chunk, and the space is reclaimed by the Java garbage collector.

Windows are represented by a pair of pointers that refer to the first and last item of the window in the stream buffer. Open windows only have a pointer to the first item; the last pointer is set to `NULL` (i.e., unknown). Obviously, there are no open windows that refer to chunks in which all items have been marked as *consumed*. As a result of this organization, items need to be materialized only once, even though they can be involved in many windows. Furthermore, other expressions that potentially require materialization can re-use the stream buffer, thereby avoiding copying the data.

6.3.3 *General* `FORSEQ`

The implementation of the general `FORSEQ` varies significantly from that of the three kinds of windows. In particular, representing a sub-sequence by its first and last item is not sufficient because the general `FORSEQ` involves the processing of non-contiguous sub-sequences. In order to enumerate all sub-sequences, our implementation uses the algorithm of Vance/Maier [28], including the bitmap representation to encode sub-sequences. This algorithm produces the subsequences in the right order so that no sorting of the windows is needed in the absence of an `ORDER BY` clause. Furthermore, this algorithm is applicable to infinite input streams. Additional optimizations are needed in order to avoid memory overflow for a general `FORSEQ` on infinite streams; e.g., the hopeless window detection, described in the next section.

6.4 Optimizations

This section lists several simple optimizations that we found useful in our implementation. In particular, these optimizations were important in order to meet the requirements of the Linear Road benchmark (Section 7). Each of these optimizations serves one or a combination of the following three purposes: a.) reducing the memory footprint (e.g., avoid materialization); b.) reducing the CPU utilization (e.g., indexing); c.) improving streaming (e.g., producing results early). Although we are not aware of any streaming SQL engine which implements all these optimizations, we believe that most optimizations are also applicable for streaming SQL. A condition for most optimizations is that a predicate-based approach to define window boundaries is adopted. So far, no such streaming SQL proposals have been published.

The proposed list of optimizations is not exhaustive and doing a comprehensive study of the effectiveness of alternative optimization techniques is beyond the scope of this paper. The purpose of this list is to give an impression of the kinds of optimizations that are possible. All these optimizations are applied in addition to the regular XQuery optimizations on *standard* XQuery expressions (e.g., [8]). For example, rewriting reverse axes can be applied and is just as useful for `FORSEQ` queries as for any other query.

6.4.1 *Predicate Movearound*

The first optimization is applied at compile-time and moves a predicate from the `WHERE` clause into the `START` and/or `END` clauses of a `FORSEQ` query. This optimization can be illustrated by the following example:

```
forseq $w in $seq landmark window
   start when fn:true()
   end when fn:true()
where $w[1] eq ‘‘S’’ and $w[last] eq ‘‘E’’ return $w
```

This query can be rewritten into the following equivalent query, which computes significantly fewer windows and can therefore be executed much faster and with lower memory footprint:

```
forseq $w in $seq landmark window
   start curItem $s when $s eq ‘‘S’’
   force end curItem $e when $e eq ‘‘E’’
return $w
```

6.4.2 *Cheaper Window*

In some situations, it is possible to rewrite a landmark window query into a sliding window query or a sliding window query into a tumbling window query. This rewrite is useful because tumbling windows are cheaper to compute than sliding windows, and sliding windows are cheaper than landmark windows. This rewrite is frequently applicable if schema information is available. If it is known (given the schema), for instance, that the input sequence has the following structure “a, b, c, a, b, c, ...”, then the following expression

```
forseq $w in $seq sliding window
   start curItem $s when $s eq ‘‘a’’
   end curItem $e when $e eq ‘‘c’’
return $w
```

can be rewritten into the following equivalent expression:

```
forseq $w in $seq tumbling window
   start curItem $s when $s eq ''a''
   end curItem $e when $e eq ''c''
return $w
```

6.4.3 Indexing Windows

Using sliding and landmark windows, it is possible that several thousand windows are open at the same time. In the Linear Road benchmark, for example, this situation is the norm. As a result, the END condition must be checked several thousand times (for each window separately) with every new item (e.g., car position reading). Obviously implementing such a check naïvely is a disaster. Therefore, it is advisable to use an index on the predicate of the END clause. Again, this indexing is best illustrated with the help of an example:

```
forseq $w in $seq landmark window
  start curItem $s when fn:true()
  end curItem $e when $s/@id eq $e/@id
return $w
```

In this example, windows consist of all sequences in which the first and last items have the same *id*. (This query pattern is frequent in the Linear Road benchmark which tracks cars identified by their id on a highway.) The indexing idea is straightforward. An "@id" index (e.g., a hash table) is built on all windows. When a new item (e.g., a car position measurement with the *id* of a car) is processed, then that index is probed in order to find all matching windows that must be closed. In other words, the set of open windows can be indexed just like any other collection.

6.4.4 Improved Pipelining

In some situations, it is not necessary to store items in the stream buffer (Figure 8). Instead, the items can directly be processed by the WHERE clause, RETURN clause, and/or nested FOR, LET, and FORSEQ clauses. That is, results can be produced even though a window has not been closed. This optimization can always be applied if there is no ORDER BY and no FORCE in the END clause. It is straightforward to implement for tumbling windows. For sliding and landmark windows additional attention is required in order to coordinate the processing of several windows concurrently. The query of Section 5.1 is a good example for the usefulness of this optimization.

6.4.5 Hopeless Windows

Sometimes it is possible to detect at runtime that the END clause or the predicate of the WHERE clause of an open window cannot be fulfilled. We call such windows *hopeless windows*. Such windows can be closed immediately, thereby saving CPU cost and main memory. The query of Section 5.2 is a good example for which this optimization is applicable: After an hour, an open window can be closed due to the WHERE condition even though the END condition of the window has not yet been met.

6.4.6 Aggressive Garbage Collection

In some cases, only one or a few items of a window are needed in order to process the window (e.g., the first or the last item). Such cases can be detected at compile-time by analyzing the nested expressions of the FLWOR expression (e.g., the predicates of the WHERE clause). In such situations, items in the stream buffer can be marked as *consumed* even though they are part of an open window, resulting in a more aggressive chunk-based garbage collection. A good example for this optimization is the query of Section 5.3.

7. EXPERIMENTS AND RESULTS

7.1 Linear Road Benchmark

To validate our implementation of FORSEQ and continuous XQuery processing, we implemented the Linear Road benchmark [5] using the extended MXQuery engine. The Linear Road benchmark is the only existing benchmark for data stream management systems (DSMS). This benchmark is challenging. So far, the results of only three compliant implementations have been published: Aurora [5], an (unknown) relational database system [5], and IBM Stream Core [17]. Both the Aurora and IBM Stream Core implementations are low-level, based on a native (C) implementation of operators or processing elements, respectively. The implementation of the benchmark on an RDBMS uses standard SQL and stored procedures, but no details of the implementation have been published. There is also an implementation of the benchmark using CQL [4]; however, no results of running the benchmark with that implementation have been published. To the best of our knowledge, our implementation is the first compliant XQuery implementation of the benchmark.

The benchmark exercises various aspects of a DSMS, requiring window-based aggregations, stream correlations and joins, efficient storage and access to intermediate results and querying a large (millions of records) database of historical data. Furthermore, the benchmark poses real-time requirements: all events must be processed within five seconds.

The benchmark describes a traffic management scenario in which the toll for a road system is computed based on the utilization of those roads and the presence of accidents. Both toll and accident information are reported to cars; an accident is only reported to cars which are potentially affected by the accident. Furthermore, the benchmark involves a stream of historic queries on account balances and total expenditures per day. As a result, the benchmark specifies four output streams: Toll notification, accident notification, account balances, and daily expenditures. (The benchmark also specifies a fifth output stream as part of a travel time planning query. No published implementation has included this query, however. Neither have we.)

The benchmark specification contains a data generation program which produces a stream of events composed of car positions and queries. The data format is CSV which is natively supported by MXQuery. Three hours worth of data are generated. An implementation of the benchmark is compliant if it produces the correct results and fulfills the five seconds real-time requirement. The correctness of the results are validated using a validation tool so that load shedding or other load reduction techniques are not allowed.[3] Fulfilling the real-time requirements becomes more and more challenging over time: With a scale factor of 1.0, the data generator produces 1,200 events per minute at the beginning and 100,000 events per minute at the end.

The benchmark specifies different scale factors L, corre-

[3]In our experiments, we encountered the same bugs as reported in [17] with the validation tool and data generator. Otherwise, all our results validated correctly.

sponding to the number of expressways in the road network. The smallest L is 0.5. The load increases linearly with the scale factor.

7.2 Benchmark Implementation

As mentioned in the previous section, our benchmark implementation is fully in XQuery, extended with `FORSEQ` and continuous queries. Our first attempt was to implement the whole benchmark in a single XQuery expression; indeed, this is possible! However, MXQuery was not able to optimize this huge expression in order to achieve acceptable (i.e., compliant) performance. As a consequence, we decided to (manually) partition the implementation into eight continuous XQuery expressions and five (temporary) stores; i.e., a total of 13 *boxes*. Figure 9 shows the corresponding workflow: The input stream produced by the Linear Road data generator is fed into three continuous XQuery expressions which in turn generate streams which are fed into other XQuery expressions and intermediate stores. Binding an input stream to an XQuery expression is done by *external* variable declarations as specified in the XQuery recommendation [6] and demonstrated in the examples in Section 5. This approach is in line with the approaches taken in [5, 17], the only other published and compliant benchmark implementations. Aurora, however, uses 60 boxes (!).

Seven threads were used in order to run the continuous XQuery expressions and move data into and out of data stores. Tightly coupled XQuery expressions (with a direct link in Figure 9) ran in the same thread. The data stores were all main-memory based (not persistent and not recoverable) using a synchronized version of the stream buffer described in Section 6.

7.3 Results

The implementation of the benchmark was evaluated on a Linux machine with a 2.2 GHz AMD Opteron processor and 4GB of main memory. Our hardware is comparable to the machines used in [5] and [17]. A Sun JVM in Version 1.5.0_09 was used, the maximum heap size was set to 2 GB which corresponds to the available RAM used in the experiments reported in [5],[17]. The results can be summarized as follows for the different scale factors **L**:

- *L=0.5, 1.5, 1.5:* MXQuery is fully compliant.
- *L=2.0:* MXQuery is not compliant. The maximum response time was 23 seconds; five seconds are allowed.

The best published results so far are compliant with an L of 2.5 [5, 17]. These implementations are low-level C implementations that do not use a declarative language (such as SQL or XQuery). An L of 2.5 is still out of reach for our implementation. However, the differences are surprisingly small (less than a factor of 2) given that our focus was to extend a general-purpose XQuery engine whereas those implementations directly target the Linear Road benchmark. Also, MXQuery is written in Java which comes with a performance penalty.

The only compliant SQL implementation of the benchmark [5] is at an L of 0.5 (contrasting an L of 1.5 of our XQuery implementation). The maximum response times of the SQL implementation at L 1.0 and 1.5 were several orders of magnitude worse than the benchmark allows (2031 and 16346 seconds, respectively). Details of that SQL implementation of the benchmark are not given; however, it seems that the overhead of materializing all incoming events in a

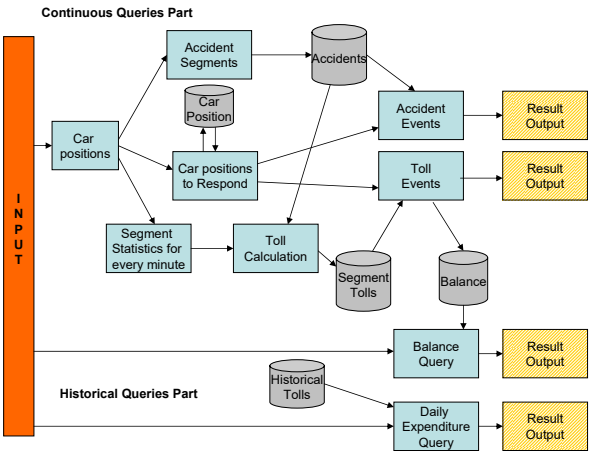

Figure 9: Data Flow of LR Implementation

relational database is prohibitive. As part of the STREAM project, a series of CQL queries were published in order to implement the benchmark. However, no performance numbers were ever published using the CQL implementation. In summary, there does not seem to be a SQL implementation of the benchmark that beats our XQuery implementation. Fundamentally, there is no reason why either SQL or XQuery implementations would perform better on this benchmark because essentially the same optimizations are applicable to both languages. Due to the impedance mismatch between streams and relations, however, it might be more difficult to optimize streaming SQL because certain optimizations must be implemented twice (once for operators on streams and once for operators on tables).

8. RELATED WORK

As mentioned in the introduction, window queries and data-stream management have been studied extensively in the past; a survey is given in [15]. Furthermore, there have been numerous proposals to extend SQL; the most prominent examples are AQuery [22], CQL [4], and StreaQuel [7]. StreamSQL [27] is a recent activity (started in November 2006) that tries to standardize streaming extensions for SQL. As part of all that work, different kinds of windows were proposed. In our design, we were careful that all queries that can be expressed in these SQL extensions can also be expressed in a straightforward way using the proposed XQuery extensions. In addition, if desired, special kinds of streams such as the i-streams and d-streams devised in [4] can be implemented using the proposed XQuery extensions. Furthermore, we adopted several important concepts of those SQL extensions such as the window types. Nevertheless, the work on extending SQL to support windows is not directly applicable to XQuery because XQuery has a different data model and supports different usage scenarios. Our use cases, for instance, involved certain patterns, e.g., the definition of window boundaries using general constraints (e.g., on authors of RSS postings) that cannot be expressed in any of the existing SQL extensions. All SQL extensions published so far are only able to specify windows based on size or time constraints; those SQL extensions are thus not expressive enough to handle these use cases, even if the data is relational. Apparently, StreamSQL will adopt

the predicate-based approach, but nothing has been published so far. (The StreamSQL documentation in [27] still uses size and time constraints only.)

Recently, there have also been proposals for new query languages in order to process specific kinds of queries on data streams. One example is SASE [29] which was proposed to detect patterns in RFID streams; these patterns can be expressed using regular expressions. Another proposal is Wavescope [23], a (Turing-complete) functional programming language in order to process signals in a highly scalable way. While such languages and systems are useful for particular applications, the goal of this work is to provide general-purpose extensions to an existing main-stream programming language. Again, we made sure in our design that all the SASE and Wavescope use cases can be expressed using the proposed XQuery extensions; however, our implementation does not scale as well for those particular use cases as the SASE and Wavescope implementations.

There have been several prototype implementations of stream data management systems; e.g., Aurora [1], Borealis [2], Cayuga [9], STREAM [4], and Telegraph [7]. All that work is orthogonal to the main contribution of this paper. In fact, our implementation of the linear road benchmark makes extensive use of the techniques proposed in those projects.

The closest related work is the work on positional grouping in XQuery described in [20]. This work proposes extensions to XQuery in order to layout XML documents, one of the usage scenarios that also drove our design. The work in [20] was inspired by functionality provided by XSLT in order to carry out certain XML transformations. However, many of our use cases on data streams cannot be expressed using the proposed extensions in [20]; our proposal is strictly more expressive. Furthermore, the work of [20] does not discuss any implementation issues. Another piece of related XML work discusses the semantics of infinite XML (and other) streams [24]. That work is orthogonal to our work.

9. CONCLUSION

This paper presented two extensions for XQuery: Windows and continuous queries. Due to their importance, similar extensions have been proposed recently for SQL, and several ideas of those SQL extensions (in particular, the types of windows) have been adopted in our design. Since SQL was designed for different usage scenarios, it is important that both SQL and XQuery are extended with this functionality: Window queries are important for SQL; but they are even more important for XQuery! We implemented the proposed extensions in an open source XQuery engine and ran the Linear Road benchmark. The benchmark results seem to indicate that XQuery stream processing can be implemented as efficiently as SQL stream processing and that there is no performance penalty for using XQuery.

The most important avenue for future work is to propose the devised XQuery extensions to the W3C for possible standardization in the emerging XQuery 1.1 recommendation.

On a more technical side, extending XML Schema to describe stream properties is an important complimentary work. Such extensions should include rate and rate variance descriptions to support resource allocation and scheduling hints. Furthermore, declaring certain values as monotonously increasing (e.g., those representing time or counters) enable optimizations in query processing and memory allocation.

Acknowledgments. This work was supported (in part) by NCCR-MICS, a center funded by the Swiss National Science Foundation.

10. REFERENCES

[1] D. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[2] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2):121–142, 2006.

[5] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

[6] S. Boag et al. XQuery 1.0: An XML Query Language, 2007.

[7] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[8] D. Che, K. Aberer, and T. Özsu. Query Optimization in XML Structured-Document Databases. *VLDB Journal*, 15(3):263–289, 2006.

[9] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. In *EDBT*, 2006.

[10] D. Draper et al. XQuery 1.0 and XPath 2.0 Formal Semantics, 2006.

[11] D. Engatarov. XQuery 1.1 Requirements. W3C Internal.

[12] M. Fernandez et al. XQuery 1.0 and XPath 2.0 Data Model (XDM), 2006.

[13] P. M. Fischer, D. Kossmann, T. Kraska, and R. Tamosevicius. FORSEQ Use Cases. http://www.dbis.ethz.ch/research/publications. Technical Report, ETH Zurich, November, 2006.

[14] D. Florescu et al. The BEA streaming XQuery processor. *VLDB Journal*, 13(3):294–315, 2004.

[15] L. Golab and T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5 –14, 2003.

[16] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[17] N. Jain et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD*, 2006.

[18] S. Jeffery et al. Declarative Support for Sensor Data Cleaning. In *Pervasive*, 2006.

[19] M. Kay. Saxon: The XSLT and XQuery processor. http://saxon.sourceforge.net/.

[20] M. Kay. Positional Grouping in XQuery. In *XIME-P*, 2006.

[21] C. Koch et al. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB*, 2004.

[22] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *VLDB*, 2003.

[23] S. Madden. Wavescope: A data management system for signals. Stanford InfoSeminar, Jan. 2007.

[24] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*, 2005.

[25] K. Patroumpas and T. Sellis. Window Specification over Data Streams. In *International Conference on Semantics of a Networked World (ICSNW)*, 2006.

[26] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.

[27] StreamSQL.org. StreamSQL documentation. http://streamsql.org/pages/documentation.html.

[28] B. Vance and D. Maier. Rapid Bushy Join-Order Optimization with Cartesian Products. In *SIGMOD*, 1996.

[29] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *SIGMOD*, 2006.