

Extending XQuery with Window Functions

Irina Carabus 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 [7]. XQuery 1.0 is a candidate recommendation of the W3C. So far, almost fifty XQuery implementations are advertised on the W3C web pages, including implementations from all major database vendors (e.g., IBM, Microsoft, and Oracle) 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 contin-

uous window query. Even if the communication data is not XML, XQuery is often the right choice because the XQuery data model was designed to process other data formats (e.g., CSV), too. 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., [26, 5, 10]. 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 XML use cases. The SQL extensions have been specifically tuned for streaming applications for which SQL is indeed applicable.

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 [15].

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* [16], is already a good match to 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 [5] 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 as a language for continuous queries.
- *Use Cases*: A series of examples that demonstrate the expressiveness of the proposed extensions.
- *Implementation Design*: Optimization techniques such as the indexing of windows and memory management techniques.
- *Linear Road Benchmark*: The results of running the Linear Road benchmark [6] 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 (i.e., windows) and iterates over these windows. The window boundaries are determined by the predicate on `author` in the `WHEN` clause of the `END` clause. The syntax and semantics of the FORSEQ clause are 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-

```
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

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 [3] can also be carried out on user sessions.
- *Social Networking*: An RFID reader 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 [22].
- *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 [24].
- *Financial Data Streams*: Window queries can be used in order to carry out fraud detection, chart pattern analysis, algorithmic trading and finding opportunities for arbitrage deals by computing call-put parities [17].

We compiled around sixty different use cases in these application areas in a separate document [17]. 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, most examples cannot be processed using SQL, even considering the latest extensions proposed in [26, 5, 10]. Most of these use cases involve other operators such as negation, 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 to address these applications.

¹In 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.

```

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

```

Figure 3: Grammar of Extended FLWOR Expression

3. FORSEQ CLAUSE

3.1 Basic Idea

Figure 2 gave 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*.

The FORSEQ clause can be seen as a generalization of the `FOR` and `LET` clauses. `FOR` binds the running variable to an item of the input sequence in each iteration; obviously, an item is a particular sub-sequence. The `LET` clause binds its variable to the whole input sequence; again, this is a special sub-sequence.

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.

The FORSEQ clause does not involve an extension or modification of the XQuery data model (XDM). 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 of 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. If the static type of the input sequence is \mathcal{T} , then the static type of the running variable of the FORSEQ clause is $\text{prime}(\mathcal{T})+$. Here, $\text{prime}(\mathcal{T})$ is the prime type as defined in the XQuery formal semantics for the `FOR` clause [14] and “+” denotes the *one or more* quantifier of the XQuery type system. A “+” quantifier is used because the running variable is never bound to the empty sequence. To give an 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: $(\text{string} \mid \text{integer})+$; i.e., a sequence of strings *or* integers because a sub-sequence can involve both strings and 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 [19]. Figure 4 shows an example of these three types of windows: These types differ in the way the 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 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, most SQL extensions propose to define windows based on size (i.e., number of items) or duration (time between 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

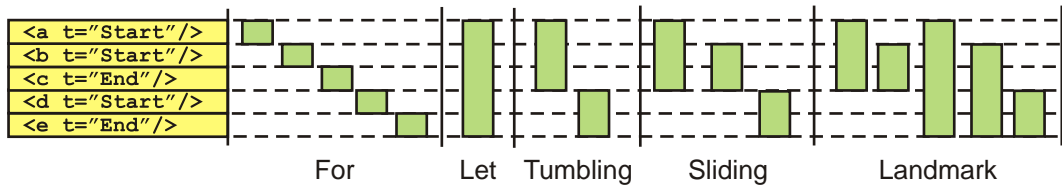


Figure 4: Window Types

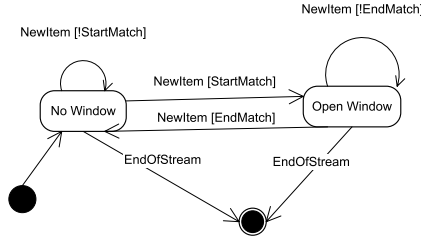


Figure 5: Window Automaton

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* [29]. 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 was given in Figure 2 in which each window was 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) [14] 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 `WHEN` condition of the `END` clause, that item is the last item of the window.

Any XQuery expression can be used in a `WHEN` clause (Figure 3). 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() ...
```

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() ...
```

That is, 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` 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 $lookAhead when $lookAhead ...` 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 [29].

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 operationally, 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. 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. Furthermore, the general `FORSEQ` clause is difficult to optimize. Use cases for which landmark, sliding, and tumbling windows are not sufficient are given in [33] 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. Besides the examples of [33], we could not find additional use cases in the literature for which such a general `FORSEQ` clause is needed.

3.4 Fine Points

There are several additional corner cases which have not been addressed in the previous paragraphs. These are addressed in the following paragraphs. Again, the full grammar of the proposed extensions is given in Figure 3.

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).

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. The `newstart` option is syntactic sugar and avoids that the condition of the `START` clause is replicated in the `END` clause.

3.4.3 Time-based Windows

Many use cases for window queries involve the notion of time. For example, an application might be interested in looking at windows of items an hour at a time. Many applications provide a time stamp as part of the input sequence; for instance, every item of an RSS feed or a Web log comes with a time stamp, and most sensors annotate their measurements with time stamps. Such time stamps can naturally be used in the predicates of the `WHEN` conditions of the `START` and `END` clauses in order to specify time-based windows.

If the application requires a time-based window and the input stream does not involve time stamps for each item, then we recommend to transform the input stream by calling an external user-defined function that annotates each item with a time stamp. In the following example, which defines tumbling windows of one hour duration, the `ext:addTStamp` functions adds a `tstamp` attribute to each item of the input sequence.

```
forseq $w in ext:addTStamp($seq) tumbling window
  start curItem $first when fn:true()
  end curItem $last when
    ($last/@tstamp - $first/@tstamp) eq 'P1H'
```

‘P1H’ is the ISO (and XQuery 1.0) way to represent a time duration of one hour. This way to define time-based windows is in contrast to most proposals to extend SQL for win-

down queries (e.g., [5]): those SQL proposals use a system-defined time, rather than an application-defined time. We chose to use application-defined time stamps because it is the more general approach. Obviously, in our approach, system-defined time stamps can be implemented, too, e.g., by defining the `ext:addTStamp` function accordingly. Furthermore, for many applications that already define their own notion(s) of time, using a system-defined time is not appropriate. A prominent example is RSS, which defines its own time stamp (a `date` element) and for which the time of publishing a posting might be totally different to the time(s) that the posting is actually consumed at various sites.

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 quite 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 [17]. 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.

Despite this expressive power, the proposed extensions can be implemented efficiently. All existing XQuery optimization techniques continue to be applicable and there are specific techniques that help to speed up the execution of FORSEQ expressions (Section 6). Conceptually, there is no reason why an XQuery window query should be less efficient than an equivalent SQL window query (if it exists), and the benchmark results with the Linear Road benchmark seem to confirm this observation (Section 7).

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) [16] 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 10 (Fahrenheit) 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 i th 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 application 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: Rather than reporting the current measurement, an average of the current and the last two 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 - $s eq 2
return fn:avg($w)
```

5. EXAMPLES

This section contains three more sophisticated examples that demonstrate the expressive power of the FORSEQ clause and continuous XQuery processing.

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 log entry contains the name of the user and the operation carried out by the user. In order to identify the length of each user session, the following query can be used (a session starts with a *login* operation and ends with a *logout*):

```
declare variable $weblog as (entry)* external
for $u in fn:distinct-values($weblog/user)
forseq $w in $weblog[user eq $u] tumbling window
  start curItem $in when $in/op eq 'login'
  end curItem $out when $out/op eq 'logout'
return count($w)
```

Based on this template, more complex analyses can be carried out. For example, the overall average of the length of user sessions can be computed or a moving average as defined in Section 4, if the log is monitored continuously (i.e., online). This query works on an infinite Web log because both FORSEQ and `distinct-values()` are non-blocking functions.

5.2 Sorting Infinite Sequences

FORSEQ can also be combined effectively with other XQuery extensions such as the recently proposed XQueryP [9]. This way, sophisticated programs can be written on data streams; programs which can otherwise only be written using an imperative programming language such as Java.

The following example shows how an infinite stream can be sorted. The idea is to assume that the input is *quasi-sorted*, but that some items in the input are out-of-order. If an item is out-of-order by no more than two positions, it is put back into the right position. If the item is out-of-order by more than two positions, it is removed and not part of the output stream. For example, the input 1, 4, 3, 5, 2, ... would be sorted to 1, 3, 4, 5, ...: The 3 is put into the right position and the 2 is removed because it is out-of-order by more than two positions.

This kind of sorting is beneficial if the incoming stream is generated by several sources and the items arrive in the wrong order due to race conditions in the network. A typical usage scenario is a sensor network with many sensors. Assuming that the input is a sequence with positive integers (greater 0) and contains no duplicates, this window-based sorting can be implemented using the following XQueryP program with a FORSEQ clause:

```
declare execution sequential;
declare variable $seq as (xs:int)** external;
declare variable $runMax as xs:int := 0 ;
forseq $w in $seq sliding window
  start position $s when fn:true()
  end position $e when $e - $s eq 2
return
  for $x in $w
  where $x gt $runMax and $x le $w[1]
  order by $x
  return { set $runMax := $x; $x}
```

5.3 Sensor State Aggregation

Another frequent use case 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>
```

6. IMPLEMENTATION

This section describes how we extended an existing Java-based open-source XQuery engine in order to implement the FORSEQ clause and continuous XQuery processing. We used that extended XQuery engine in order to validate all the use cases of [17] and run the Linear Road benchmark (Section 7).

Implementing the extensions for continuous XQuery, we adopted implementation concepts from the SQL world and data stream management systems such as Aurora, Telegraph, and STREAM. Almost all the work to turn an XQuery engine into a *continuous* XQuery engine is independent of SQL and XQuery; this work involves concepts for main memory management with garbage collection, synchronization of accesses to data streams, splitting and merging of streams, and the recoverability of data streams in the presence of system failures (i.e., avoid the double processing of items after a failure). Implementing the extensions for the FORSEQ clause and window queries, the architecture and optimization techniques of existing *streaming* XQuery engines could be leveraged. The remainder of this section gives an overview of the extensions we made and provides more details of some of the implementation and optimization aspects that are specific to the proposed extensions.

6.1 Streaming XQuery Engines

There are two different kinds of XQuery engines. The first kind is integrated into an XML and/or extended relational database system. Examples are the XQuery implementations of the big database vendors; e.g., IBM, Microsoft, and Oracle [31]. An example for this type of XQuery engine developed in academia is MonetDB/XQuery [8]. These engines

compile XQuery expressions (and SQL queries) into a tree of operators (e.g., extended relational algebra) and this tree of operators is executed in the database at runtime. The execution is typically carried out using an *open-next-close* iterator model [20], as in any traditional database system.

The second type of XQuery engine consists of so-called *streaming XQuery engines*. These query engines work independently from a database and can take input data from multiple sources; e.g., databases, the file system, or message queues. In order to take input from different sources, these engines define a low-level data API such as SAX or StaX. Non-XML sources (e.g., CSV, EXCEL, relational databases) can be integrated by implementing the data API on top of these non-XML sources, which is typically straightforward. Using a streaming data API such as SAX or StaX, it is straightforward to feed an infinite stream of items as input into a streaming XQuery engine. Like the database XQuery engines, streaming XQuery engines compile an XQuery expression into a tree of operators and execute these operators using an iterator model. Examples of streaming XQuery engines are Saxon [23], BEA's XQuery engine [18], and FluXQuery [25].

We chose to extend the open-source streaming XQuery engine MXQuery². Naturally, this kind of XQuery engine is more appropriate to process continuous XQuery expressions because the data API makes it possible to feed infinite streams into the engine. In theory, the extensions could be implemented in a database XQuery engine, too. In particular, the FORSEQ extensions can be implemented in any kind of XQuery engine.

To implement the FORSEQ clause, the following adaptations were made:

- 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 6.3.
- The runtime system was extended with four new iterators that implement the three different kinds of windows and the general FORSEQ. Furthermore, the main memory management was extended. These extensions are described in Section 6.2.

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. To this end, all operators must be annotated in order to specify whether and on which of its inputs an operator is blocking. Many XQuery engines already carry out this kind of data flow analysis and the annotation of blocking operators for optimization purposes.
- 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 [4] and database query optimiza-

tion [30]. 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 [2]. 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 our prototype yet.

6.2 FORSEQ Iterators

As mentioned in the previous section, the only extension of the runtime system in order to support the FORSEQ clause was the implementation of four new iterators that implement the three different kinds of windows and the general FORSEQ. The implementation of these four iterators is similar to the implementation of an iterator that implements a FOR clause: All these iterators implement second-order functions which bind variables and then execute a function on those variable bindings. For FORSEQ, as for FOR, the function to execute on the variable bindings is implemented as an iterator tree (just like any other expression) and encodes nested FOR, LET, and FORSEQ clauses (if any) as well as WHERE, ORDER BY, and RETURN clauses. The XQuery engine that we chose to extend has similar mechanics as those described in [18], but all XQuery engines have some mechanics to implement second-order functions such as FOR and these mechanics can be leveraged for the FORSEQ iterators.

The difference between a FOR iterator and the four different FORSEQ iterators 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.

The most interesting aspect of the implementation of the FORSEQ iterators for tumbling, sliding, and landmark windows is main memory management and garbage collection. The items of the input sequence are materialized in main memory. Figure 6 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. 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 an ORDER BY clause; in this case, FORSEQ is not applicable.

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. In Figure 6, consumed items are indicated as white squares; active items are indicated as colored squares. In Figure 6, Window 1 is closed whereas

²www.mxquery.org

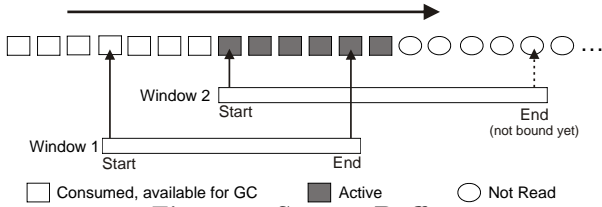


Figure 6: Stream Buffer

Window 2 is still open; as a result only the items of Window 2 are marked as active in the stream buffer.

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* (not shown in Figure 6). 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 de-chaining 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 main memory can be 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.

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 [32], including the bitmap representation to encode sub-sequences. This algorithm produces the sub-sequences 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.

6.3 Optimizations

This section lists several 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).

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., [11]). For example, rewriting reverse axes [28] can be applied and is just as useful for FORSEQ queries as for any other query.

6.3.1 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 less 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
```

6.3.2 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
```

6.3.3 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. One optimization approach is to apply memoization; that is, caching the result of evaluating the condition of the END clause. General-purpose memoization for XQuery which is also applicable in this context, has been studied in [13]. Another option is to build an index on the predicate used in the WHEN condition of the END clause. This is the option we chose to implement. 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.

6.3.4 Improved Pipelining

In some situations, it is not necessary to store items in the stream buffer (Figure 6). 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.

6.3.5 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.

6.3.6 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.

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 [6] using the extended streaming XQuery engine. The Linear Road benchmark is the only existing benchmark for data stream management systems (DSMS). This benchmark is very challenging. So far, the results of only three compliant implementations have been published: Aurora [6], an (unknown) relational database system [6], and IBM Stream Core [21]. Both the Aurora and IBM Stream Core implementations are low-level, based on a native (C) implementations 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 [5]; 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 also 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 that produces a stream of events composed of car positions and queries. The data format is CSV, which could already be processed by our streaming XQuery engine. 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 , corresponding 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, our extended XQuery engine 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 7 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 [7] and demonstrated in several examples in this paper. This approach is in line with the approaches taken in [6, 21], the only other published and compliant benchmark implementations. Aurora, however, uses 60 boxes (!).

³In our experiments, we encountered the same bugs as reported in [21] with the validation tool and data generator. Otherwise, all our results validated correctly.

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 7) 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 [6] and [21]. A Sun JVM in Version 1.5.0.09 was used, the maximum heap size was set to 2 GB (=RAM available in [6],[21]). The results can be summarized as follows for the different scale factors L :

- $L=1.0$: Our implementation is fully compliant.
- $L=1.5$: The results are compliant until minute 45 (out of 180), after which our implementation begins to exceed the response times at the change of minutes. This can be explained by periodic operations that occur at these minute changes. It should be noted, however, that the response times did not degrade catastrophically (as observed in [6]), but slowly kept increasing until the end of the benchmark. The highest response time observed was 26 seconds.

Due to time constraints, we were not able to implement all optimization ideas in order to deal with those response time peaks. By the end of the year, we expect to have compliant results for an L of 1.5. All optimization ideas involve improved scheduling of the data streams and increased parallelism; the XQuery processing itself, the main goal of this work, showed more than sufficient performance already.

The best published results so far are compliant with an L of 2.5 [6, 21]. This is still out of reach for our implementation. However, the differences are surprisingly small given that our focus was to extend a general-purpose XQuery engine whereas those implementations directly target the Linear Road benchmark. As a result, those implementations have highly-tuned representations, temporary storages and synchronization primitives for streams. Furthermore, our implementation is in XQuery, whereas those benchmark implementations are hand-optimized and extremely complex. A third explanation for the differences in performance is that our XQuery engine is written in Java, rather than C.

[6] mentions that there is a SQL implementation of the benchmark, which is compliant at an L of 0.5 (contrasting the L of 1.0 in our implementation). The the maximum response times at L 1.0 and 1.5 were several orders of magnitude worse (2031 and 16346 seconds, respectively). Details of that SQL implementation of the benchmark are not given; it seems that that implementation is quite complex. 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 that beats our XQuery implementation.

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 [19]. Furthermore, there have

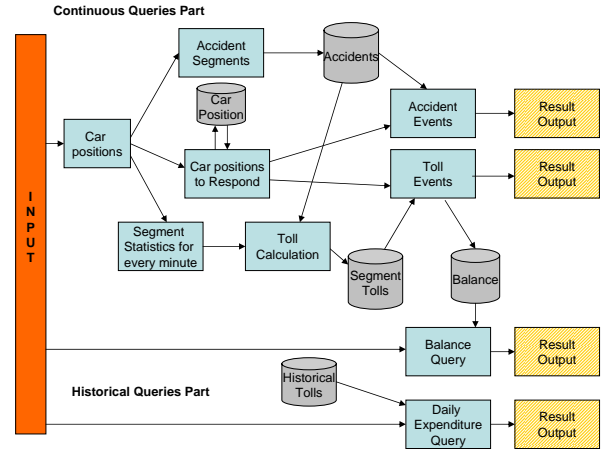


Figure 7: Data flow model of LR implementation

been numerous proposals to extend SQL; the most prominent examples are AQuery [26], CQL [5], and StreaQuel [10]. As part of 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 [5] 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. The SQL extensions typically focus on specifying windows based on size or time constraints.

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 [33] which was proposed to detect patterns in RFID streams; these patterns can be expressed using regular expressions. While such languages and systems might be 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 use cases can be expressed using the proposed XQuery extensions; however, our implementation does not scale as the SASE implementation does.

There have been several prototype implementations of stream data management systems; e.g., Aurora [2], Borealis [1], Cayuga [12], STREAM [5], and Telegraph [10]. 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 [24]. This work proposes extensions to XQuery in order to process XML documents, one of the usage scenarios that also drove our design. The work in

[24] 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 [24]; our proposal is strictly more expressive. Furthermore, the work of [24] does not discuss any implementation issues. Another piece of related XML work discusses the semantics of infinite XML (and other) streams [27]. 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 work on rigorous and comprehensive optimization techniques for continuous and windowed XQuery expressions. In this work, only initial optimization ideas were sketched. Furthermore, we plan to propose the devised XQuery extensions to the W3C for possible standardization in the emerging XQuery 1.1 recommendation.

10. REFERENCES

- [1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.
- [2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.
- [3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *SIGMOD*, 1993.
- [4] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [5] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2):121–142, 2006.
- [6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.
- [7] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language, 2006.
- [8] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery — The Relational Way. In *VLDB*, 2005.
- [9] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, and J. Robie. XQueryP: Programming with XQuery. In *XIME-P*, 2006.
- [10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.
- [11] D. Che, K. Aberer, and T. Özsu. Query Optimization in XML Structured Document Databases. *VLDB Journal*, 15(3):263–289, 2006.
- [12] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. In *EDBT*, 2006.
- [13] Y. Diao, D. Florescu, D. Kossmann, M. Carey, and M. Franklin. Implementing Memoization in a Streaming XQuery Processor. In *XSym*, 2004.
- [14] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics, 2006.
- [15] D. Engatarov. XQuery 1.1 Requirements. W3C Internal.
- [16] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM), 2006.
- [17] P. M. Fischer, D. Kossmann, T. Kraska, and R. Tamosevicius. Windows for XQuery Use Cases.doc. Technical Report, ETH Zurich, Nov. 2006.
- [18] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *VLDB Journal*, 13(3):294–315, 2004.
- [19] L. Golab and T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.
- [20] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.
- [21] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD*, 2006.
- [22] S. Jeffery, G. Alonso, M. Franklin, W. Hong, and J. Widom. Declarative Support for Sensor Data Cleaning. In *Pervasive*, 2006.
- [23] M. Kay. Saxon: The XSLT and XQuery processor. <http://saxon.sourceforge.net/>.
- [24] M. Kay. Positional Grouping in XQuery. In *XIME-P*, 2006.
- [25] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB*, 2004.
- [26] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *VLDB*, 2003.
- [27] D. Maier, J. Li, P. Tucker, K. Tuftte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*, 2005.
- [28] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshops*, 2002.
- [29] K. Patroumpas and T. Sellis. Window Specification over Data Streams. In *International Conference on Semantics of a Networked World (ICSNW)*, 2006.
- [30] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.
- [31] M. Rys, D. Chamberlin, and D. Florescu. XML and Relational Database Management Systems: The Inside Story. In *SIGMOD*, 2005.
- [32] B. Vance and D. Maier. Rapid Bushy Join-Order Optimization with Cartesian Products. In *SIGMOD*, 1996.
- [33] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *SIGMOD*, 2006.