XQuery for Streams (XQS): XQuery Streaming Extensions Design

Ivan Shcheklein

Moscow State University
shcheklein@mail.ru

Andrey Fomichev

Institute for System Programming of the Russian Academy of Sciences
fomichev@ispras.ru

Abstract

In this paper we introduce a new language for the XML streams querying – *XQuery for Streams* or *XQS* for short. Its primary purpose is to make possible to query potentially unbounded XML streams using the whole expressive power of the XQuery language which is the standard for the XML data manipulation today.

Proposed XQS language extends XQuery with *sliding windows* and special streaming operators. *Sliding windows* is an essential and powerful technique proven by the relational data streams management systems that can be applied to the XML streams querying as well. In this paper we describe syntax and semantics of the XQS language.

1 Introduction

Today a considerable part of research papers is focused on the XML streams and continuous queries over them [1, 2, 3]. In our opinion, there are a number of reasons of such intense interest. No doubt that the main cause is the growing force of XML positions as a universal data presentation and exchange format. By virtue of its comparative simplicity and ease of human and machine understanding, XML is used today everywhere; starting with the information exchange between different applications or services (using SOAP [4] for example) and up to the data formats for sport news [5] and stock prices providers [6].

Second important reason is the growing data streams querying requirement in general. Most modern applications need efficient facilities for storing, processing and analyzing data, which arrives from the network. For instance, such information can arrive from Web Services.

In contrast to persistent data, data streams may have behavior that interferes with an effective work with them using modern database management systems (DBMS). In general, we can consider *data stream* as potentially unbounded, continually arriving information [7]. Data streams considered in this work are supposed to be *homogeneous.* It means informally that there is only one data format (exactly XML) that is used to present streamed information. Also we assume that one or multiple parameters of the stream can vary with time (*e.g.*, incoming rate can differ).

In this paper we introduce a new language for the XML streams querying – XQuery for Streams (XQS). There are two *important goals*, which we kept in mind during the development of the XQS language:

- Queries must be well-understood and semantically clear.
- Language must provide powerful and effective facilities for unbounded XML streams processing.

1.1 Our Contribution

We believe that goals pointed previously have been successfully achieved during our research and suppose that the main contributions of this paper are the following:

- We distinguish and provide an overview of two approaches to the XML streams processing. Point out strengths and weaknesses of these methods.
- We formalize the XML stream notion as an unbounded sequence of items defined in the XQuery language data model.
- We propose XQS language for processing unbounded XML streams. Our language is based on XQuery and extends it with sliding windows operators. Similar approach was proposed heretofore and successfully implemented in CQL language [17], supported by the relational data stream management system STREAM [18].
- We believe that examples of XQS queries over XML streams within this paper show that many techniques proven by the relational data streams management systems (*e.g.*, sliding

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, Moscow, Russia, 2006

windows) can be successfully incorporated into the XML streams processing.

1.2 Paper Outline

The rest of the paper is organized as follows. Within the next section we consider existing approaches and key concepts in XML streams querying. In Section 3 we survey related work. Section 4 defines data model of our language. Primarily, the notion of the XML streams is discussed there. Section 5 proposes the XQS language and gives its detailed description. In this section we provide syntax and usage examples of the language's constructs. And finally, Section 6 concludes and points out possible directions of our future work.

2 XML Streams Querying Principles

Queries over continuous data streams have much in common with queries in traditional database management systems. However, there is one important distinction related to data streams, namely, queries over continuous streams are evaluated continuously as information continues to arrive. Such queries are called *continuous queries* [7].

Many areas of the data management can benefit from employing XML streams processing and querying. Among them are:

- efficient transformation of sequentially accessed XML documents (*e.g.*, XSLT processors that "on the fly" convert XML, arrived from the web, into XHTML [2]);
- data integration from slow and distributed sources [8];
- streams filtering according to complex queries before distribution to subscribers or, simply, personalized content delivery [3];

So far as we know, there are two approaches to querying XML streams – *pure automata* approach [1, 2, 3] and *combined automata and algebraic* approach [9]. Both approaches use essential XML streams data representation as streams of tokens and represent queries as state machines. But there is important distinction between them. Each query in the second approach initially is represented in algebraic form, so as in the most relational databases. It allows making more efficient query optimization.

Although, systems based on one of these methods solve the problem of the XML streams processing well, both approaches have some constraints. In the first approach developers face with hard balance between the expressive power of the query system can handle and the manageability of queries presentation constructs. Even if you write simple filtering queries expressed in limited XPath you can be confronted with huge number of states [26] or transitions [3] in the final query presentation. Moreover, it is hard to make effective optimization that allows multiple queries in the form of state machines to be simultaneously evaluated against the same stream, the crucial feature for the SAX-like XML streams processing [27, 11].

```
<bid>
    <itemno> … </itemno>
    <bidder> … </bidder>
    <bid-amount> … </bid-amount>
    <bid-date> … </bid-date>
</bid>
```

Figure 1: 'bid' node structure.

```
<item>
    <itemno> … </itemno>
    <seller> … </seller>
    <description> … </description>
    <reserve-price> … </reserve-price>
    <end-date> … </end-date>
</item>
```

Figure 2: 'item' node structure.

```
for $i in stream("items")//item,
    $b in stream("bids")//bid
where      $i/itemno = $b/itemno
return
    <description>
          {$i/description, $b/bidno}
    </description>
```

Query 1: Simple example of the two streams join.

In order to solve these problems developers of the Raindrop system [9] proposed second approach, which we called *combined* above. They use Rainbow [13] (the XQuery engine for persistent inputs) for initial algebraic plan construction and query optimization. Then this plan is rewritten in the state machines style to process tokenized streams.

Though, described problems seem to be easier with such combined framework, it has own restrictions. Just as the first approach it is not well suitable to process one or more complex continuous queries over multiple unbounded input streams.

One of the hardest problems in querying high-rate continuous streams with powerful XQuery-like language is that we are not able to store all incoming information – simply, we don't have unbounded secondary storage. Hence, we have problems with the blocking operators processing, *e.g.* aggregation or join. In addition, if streams are not being saved, we won't be able to query historical data, *i.e.* arrived earlier.

These challenges of limited memory and blocking operators processing over unbounded streams were discussed previously in the context of the relational streams [7, 14], but never for the XML streams so far as we know.

To illustrate this problem in context of the XML streams let us informally consider simple join example (shown in Query 1) over two hypothetical streams from online auction (nodes structure of the "items" and "bids" streams are shown in Figure 2 and in Figure 1 respectively). In this query `stream` function is used to get access to the streams and retrieve data that is being processed by XQuery FLWR operator.

Even so simple example gives us the idea of this challenge – we are not able to save all items and bids to

perform join afterwards. Moreover, if blocking version of join operation is used then we will never get the answer to this query.

2.1 Sliding Windows

In the case of the relational data streams management systems there are two solutions that help to solve these problems. Firstly, we can use non-blocking versions of operators, *e.g.* XJoin [16], to get results at the same time as new information continues to arrive. It is necessary to notice, that there are no problems with blocking operators in querying streams using state machines. Nevertheless, in most systems, exploiting this approach, we aren't able to run even such simple query as we considered above.

Secondly, to solve the main problem of the insufficient space, which we have pointed out above, special streaming operators, called *sliding windows* [7, 14] are used. Sliding windows allow producing approximate answer to a query by evaluating this query not over the entire stream, but rather only over one specially described part of this stream. For example, only data, arrived within the last day, can be considered in producing query answers.

Any query, using sliding windows to approximate streams, is evaluated by the system just in the same way as a simple query over persistent documents, when system has strictly defined bounded data to work with. Nevertheless, query is still considered as continuous, and system restarts it every time after the finite part of the arriving information (*i.e.*, streams, bounded by the windows and possible persistent data) has been processed. It is obvious, that each "restart" includes windows' updating. Some old data are possible to be dropped from the window and recently arrived to be attached to the window.

As a rule, systems which can handle streams, using sliding windows, propose extensions to query language that permit at least to specify type and size of the window [18, 14]. SQL is usually used as a base language in the case of the relational streams.

For example, let us consider simple CQL query (shown in Query 2), which is borrowed from [17] (CQL or *Continuous Query Language*, is supported by the STREAM system [18]):

```
SELECT DISTINCT vehicleId
FROM PositionSpeedStream
                    [RANGE 30 Seconds]
```

Query 2: CQL example with sliding windows.

This query is composed from a sliding window operator, followed by an operator that performs projection and duplicate-elimination. The output relation of this query contains, at any time instant, the set of vehicles, transmitted a position-speed measurement within the last 30 seconds.

Incorporating sliding windows in the data streams processing is an essential method for data streams approximation that has several advantages. Firstly, it is well-defined and easily understood. The semantics of such approximation is clear, so that users of the system can be sure that they understand what result means. Most importantly, it accentuates recent information, which in the most cases of real-world applications is more important than old data. Moreover, for many such applications, sliding windows can be regarded not as an approximation technique imposed due to the impossibility of computing over all historical data, but rather as part of the desired query semantics explicitly expressed as part of the user's query.

3 Related Work

Although the problem of streams management was raised several years ago [20], there have been not so many researches in the area of the XML streams processing.

A number of XML processing systems have been developed to efficiently evaluate XPath [10] queries over streaming documents. We suppose, the XFilter system [21] was the first to define this problem, and to describe several evaluation techniques. It converts each query expressed in a limited XPath into a separate Finite State Machine (FSM); as a result it does not exploit commonality that exists among the path expressions. As an evolution of the XFilter system, YFilter [3] was proposed. It is also used to evaluate filtering queries expressed in a limited XPath, but it is able to detect common structure navigational parts of the queries. Approach that eliminates both common subexpressions in the structure navigation and in the predicates is presented in [11].

Olteanu et al. [22] describe a method which relaxes usage restrictions on the XPath reverse axes (*e.g. parent*, *preceding*) that usually exist in streams processing systems. It proposes two sets of rewriting rules that replace XPath expressions with equivalent ones without reverse axes. Unfortunately, the first set produces the same number of joins as there are reverse steps in the input path and the second set has an exponential complexity in the length of the input path.

XML streaming processor SPEX [1], based on transducer networks (*i.e.* state machines with stacks), uses some ideas of [22] to effectively evaluate complex XPath expressions.

All previous works discuss automata approach to XPath or limited XPath queries evaluation against XML streams. Also there are several papers that propose streaming execution of the XQuery language that is more flexible and powerful. Small non-recursive subset of XQuery is evaluated by state machines networks in [2]. Each primitive XQuery subexpression is translated into individual *XML State Machine* (XSM), optimized and then connected in one XSM network. This approach results in a huge number of states and transitions in final XSM and is not appropriate for querying potentially unbounded XML streams.

The issue of rewriting XQuery expressions (only supported subset) to have an ability to execute them correctly with a single pass through the dataset is

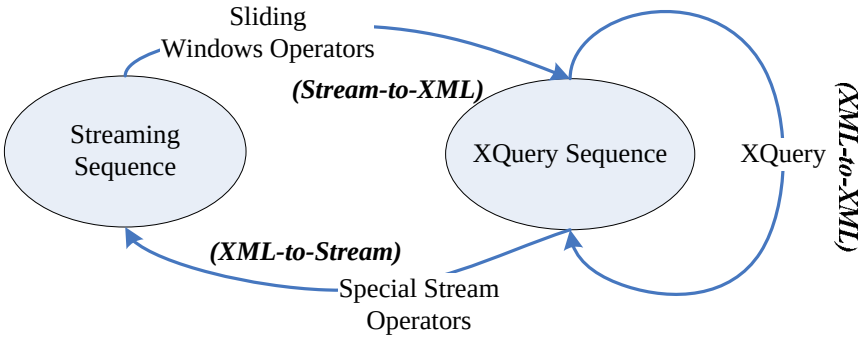

Figure 3: XQS Operator Classes.

discussed in [23]. Just as in [22] for XPath language, set of transformations for XQuery and method to determine if given XQuery query can be correctly rewritten are proposed in [23].

Instead of using state machines for each intermediate presentation form of the query, as in [23, 2] for example, *combined*, *i.e.* algebraic with automata, approach was presented in *Raindrop* system [9]. As we pointed above, the *Rainbow* persistent XQuery engine [13] is used to optimize query on first stages of rewriting. Moreover, in contrast to previous works, where final query is presented usually as one state machine, which can be used only in a "black box" style, modularity of the query is saved on all phases of its transformation and execution. Such method, we suppose, is more appropriate for streaming processing.

It is important to notice that neither of the discussed works solves the general XML streams processing challenge, *i.e.*, when streams are unbounded and system supports powerful query language (XQuery, for example). Considered papers can be divided into two groups. Possibility of unbounded streams is explicitly or implicitly considered in the first group, but queries can be expressed only in restricted query languages [3, 21, 22]. On the other hand, papers from the second group [9] consider powerful languages, but "stream" notion there is equivalent to the sequentially accessed document, arrived from the network, but it is not unbounded indeed.

4 XQS Data Model

The data model that we want to define serves one purpose. It defines all permissible input and output values of operators and expressions in the XQS language. "Streaming variant" of the relational data model [7] usually is considered as extended traditional data model. Stream in this case is presented as potentially unbounded sequence of items, where each item is a conventional tuple. Furthermore, usually tuples within one stream have identical structure, *i.e.* they all have the same collection of attributes. Hence, we can informally point out that relational stream is considered as unbounded sequence of tuples or simply unbounded relation. Also, it is significant that this model does not exclude conventional finite relations. For instance, STREAM project [18] is positioned as a system that works with streams and persistent data simultaneously.

XQS data model is based on XPath/XQuery data model [24] that is called XDM for short. The key concept of the XDM is a *sequence* – ordered collection of zero or more items. Item can be either *atomic value* or one of the seven kinds of *nodes* - document, element, attribute, comment, namespace, text and processing instruction.

The same way as relational stream is unbounded relation, XML stream in XQS data model is considered as sequence with one important distinction – this sequence of items can be potentially unbounded. At the same time it is significant to note two features. First, XQS data model include XDM, thus we are able to work with XML streams and persistent XML storage simultaneously. And, second we must point out that each item of stream is identical to item in the XDM.

For instance, we can consider streams, which are used above on the Query 1. In the XQS data model *bids* stream can be presented as one potentially unbounded sequence of items, where each item is node 'bid', shown in Figure 1. In a similar manner *items* stream can be presented.

5 XQuery for Streams

As we remarked above, XQS language consists of three parts (classes of operators). Our language structure and mappings between the XDM and XQS data models are outlined in Figure 3:

- *Stream-to-XML* class - consists of three sliding windows operators, which perform mapping from unbounded streaming sequences to XQuery sequence;
- *XML-to-Stream* class performs reverse mapping and allows us to explicitly create new streams, which can be used as an answer or reused by Stream-to-XML operators;
- *XML-to-XML* class is equivalent to the XQuery language and is used to perform all required processing over information that can be produced by Stream-to-XML operators or retrieved from the persistent storage;

Although, most part of the computational and transformational expressions within the queries is

expressed using XML-to-XML class or simply XQuery (you can see this from the examples presented in the paper), we don't consider this class later. XQuery language is widely discussed and described and it is not the direct point of this paper.

At the remainder of this section syntax and descriptive semantics of Stream-to-XML and XML-to-Stream classes are considered.

5.1 Stream-to-XML Operators

Any Stream-to-XML sliding window operator takes a stream as input and produce finite sequence, which then can be passed to XQuery expressions. Currently there are three types of sliding windows in XQS language: *data*, *time* and *node* based. Syntax of these operators is as follows:

```
WindowExpr ::=   "{" (TimeWindow |
                 DataWindow |
                 NodeWindow)
                 ["," Predicates] "}".
TimeWindow ::=   "time" Time.
DataWindow ::=   "range"
                 IntegerLiteral.
NodeWindow ::=   "node"
                 IntegerLiteral
                 "," PathExpr[i].
```

Any windowed expression can be used only in the streaming context, which is provided by XML-to-Stream operators (we will consider them later) and by means of special `stream($uri as xs:string)` function. It is used to retrieve a streaming context by the name of the stream, just as `fn:doc` function retrieves a persistent XML document.

5.1.1 Data-Based Sliding Windows

The simplest sliding window operator is the ***data*-based window** (`DataWindow` in the grammar above). It takes one integer required parameter (its value must be one or greater) that defines size of the window over stream. To illustrate how it works, let us consider a simple example with the `range` operator – "*for last received 4 bids select average bid amount on the item with itemno equal 10*" (shown in Query 3).

In this query the function call `stream("bids")` provides streaming context for the data sliding window `{range 4}`, which then returns sequence consisting of the last received 4 items from this stream (`bid1`, `bid2`, `bid3` and `bid4`). Afterwards this sequence is being processed by well-known XQuery operators. Data window working algorithm is illustrated in Figure 4.

It is obvious, that query over streaming data in most cases would have been useless if it was evaluated in the same way as queries over persistent data that is bounded in contrast to potentially unbounded streams. Conventional query over persistent storage is evaluated

```
let $b := stream("bids"){range 4}
          //bid[itemno = "10"]
return avg($b/bid-amount)
```

Query 3: Range operator simple example.

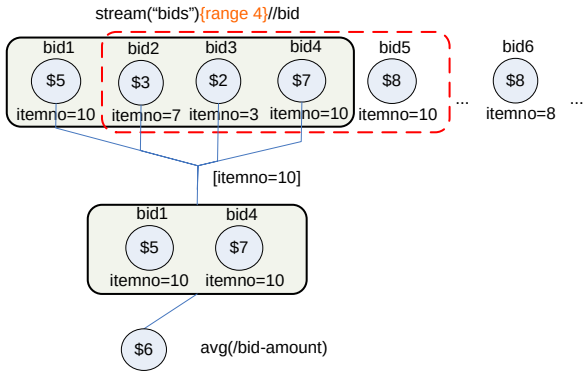

Figure 4: Range operator working mechanism.

once and deleted from the system afterwards. Therefore, all queries with sliding windows over streams are considered as *continuous queries*, which we have described above.

With such continuous semantics query (shown in Query 3) is being evaluated as before, until $6 is returned. After that system *slides* window over stream for one item ahead (`bid2`, `bid3`, `bid4` and `bid5` on the figure) and restarts query evaluation again – and so on.

XPath expression `//bid[itemno = "10"]` used in Query 3 filters sequence consisting of 4 items and returned by windowed operator `{range 4}`. Thus, in this case we have no any precise information about items quantity in variable `$b`. Such semantics is appropriate in some situations and almost useless in many others. For instance, if query computes some statistics it would be important condition to know and to define exact size of information processed on given iteration of the continuous query computation.

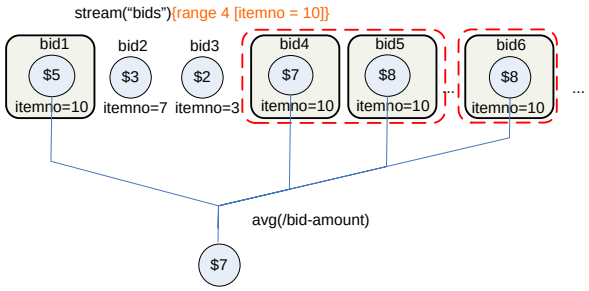

Figure 5: Range operator with predicates working mechanism.

```
let $b := stream("bids")
    {range 4, [itemno = 10]}//bid
return avg($b/bid-amount)
```

Query 4: Range operator with predicates simple example.

[i] *XPath*, *IntegerLiteral* and some other non-terminals in grammars, presented within this paper, are used in the same meaning as in the XQuery EBNF [12].

This problem can be solved by incorporating **`[Predicates] part`** in any windowed expression. Syntax of predicates is just the same as in the XPath language. With this clause every sliding window in XQS language is executed just as before, but for one distinction. Window in this case is made up only with items that meet the predicates conditions.

In Figure 5 and in Query 4 illustration of sliding window range operator with predicate and algorithm are given. In this case the same XPath predicate, `[itemno = "10"]`, has quite another meaning. It is used in this example to filter `bids` stream before window generation in such a way that every time window will consist of 4 items with `itemno` equals 10.

5.1.2 Time-Based Sliding Windows

Another way for streaming approximation in the XQS language is provided by the **time-based sliding windows** (`TimeWindow` non-terminal in the grammar). As you can see from the grammar, this type of windows can be defined in any query by the `{time T}` operator, in which parameter *T* specifies time distance. Just as execution of the next continuous query iteration begins, system fixes current starting time τ. Explicit parameter *T* and internal, dynamically (from iteration to iteration) changing parameter τ completely determine behavior of the time-based sliding window operator. It selects from the stream only items arrived into the system within the $(\tau - T, \tau]$ time interval.

We haven't said anything about the type of parameter *T* yet, because it doesn't matter and can be defined in any implementation-dependent way. For instance, in our approach we use integer number so far, which presents time interval in milliseconds. But more complicated variants (*e.g.* `xs:duration`, defined in the XMLSchema data types specification [25]) can be used in the same manner.

It's obvious, that time-based windows are almost irreplaceable when there is need to work with the time intervals in the query. Assume, for example, we have to compute statistics for the bids, arrived within the last day – *"for the bids, arrived within the last day, return* `<statistics>` *element, contained all distinct item numbers* (`itemno`) *and quantity of the bids for each item"*. Such query is very simple to express using XQS language constructs – as it is shown in Query 5, for instance.

Time-based sliding windows are able to include predicates, as before data windows were. In addition, semantics in this case also remains just the same – XPath predicates block filters stream before the window construction.

But careful reader may observe that such filters inside time-based window yield the same result as identical predicates outside do. *E.g.* queries, shown in Query 4 and in Query 3, will be the same if `range` operator is replaced by `time` one. It happens, because in the first place system approximates stream by strict time interval, as we have described before. Hence, result

```
let $b := stream("bids"){time 1 day}
return
   <statistics> {
      for $i in distinct($b/itemno)
      return
         <item>
         {$i, count($b/itemno[. = $i])}
         </item> }
   </statistics>
```

Query 5: Example of the time-based sliding window usage.

doesn't depend on position of predicates indeed. Outside or inside it is – we will have the same sequence in both cases.

Nevertheless ability to define inside predicates for time-based windows is not useless at all. It is enough to realize that every window must be completely saved into internal buffers to be recalculated during next iteration beginning – some items of the previous window can be dropped, some can be added. However, inside predicates in the time-based windows case may significantly reduce quantity of items to be saved and can be viewed as explicit query optimization or system ability to optimize some queries, by pushing down predicates.

To understand this statement you can consider two similar cases - `{time T}` window and `{time T, P}` window, where `P` is some given XPath predicate. In the first case we have sequence of items arrived within the given time interval and all these items will be saved. Possibly, they will be filtered later by outside predicate `P`. In the second case, on the other hand, system will save only predicate proven items from the ones, arrived within the same time interval.

```
<bids>
   <bid>
      <itemno> … </itemno>
      <bidder> … </bidder>
      <bid-amount> … </bid-amount>
      <bid-date> … </bid-date>
   </bid>
   …
   <bid>
      <itemno> … </itemno>
      <bidder> … </bidder>
      <bid-amount> … </bid-amount>
      <bid-date> … </bid-date>
   </bid>
   …
<bids>
```

Figure 6: Possible individual item of the new 'bids' stream.

```
let $b := stream("bids")
   {node 4, bid}[itemno = "10"]
return avg($b/bid-amount)
```

Query 6: Node operator simple example.

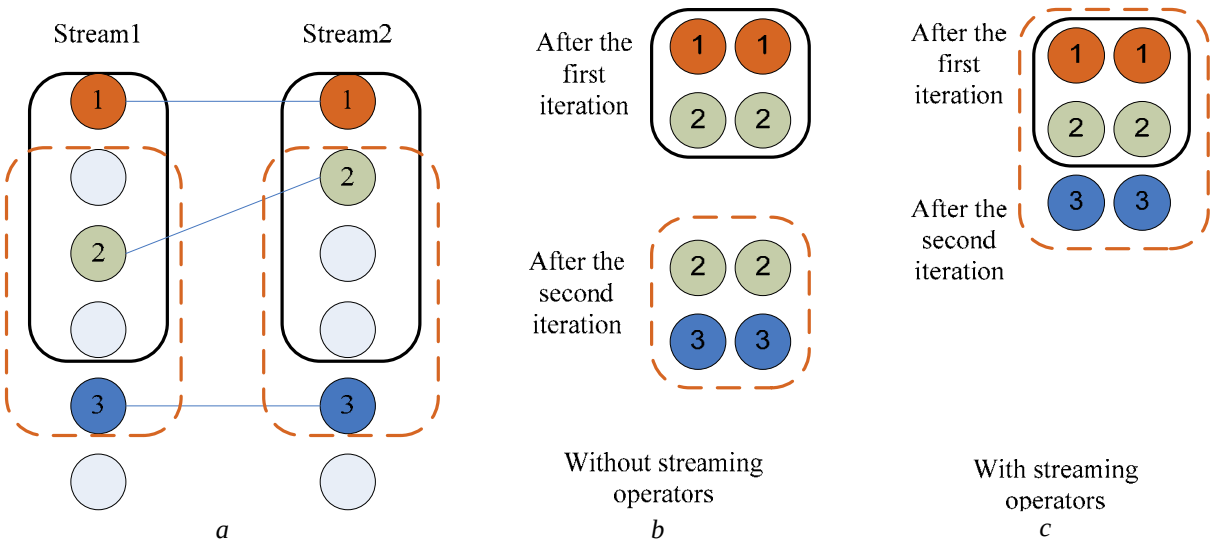

Figure 7: Visualization of join operation over two streams.

5.1.3 Node-Based Sliding Windows

The last sliding window operator, supported in XQS language is **node-based sliding window** (according to the `NodeWindow` non-terminal in the grammar). This window permits to take into consideration irregular nature of XML data.

Let's consider more complicated version of the `bids` stream. Single item of this new stream is shown on the Figure 6. The only distinction from the first version lies in the ability of every stream item to include several bids. But we are still interesting in working with precise number of the `bid` nodes, which is impossible or, at least, very hard to do with sliding windows previously described.

The node-based operator is able to solve this problem. It depends on two required parameters – positive integer number and XPath expression define size and required content of the window respectively. For instance, node-based window, shown in Query 6, selects exactly 4 bids (as before) from the new version of the bids stream.

Mechanism of the window construction in this case is similar to the one of the data window with predicates. It is composed of three steps. Firstly, given XPath expression is evaluated on every received item. Second step filters results of the first step over optional predicates. And the last step selects required amount of items. So, we are able to have a window with exact size and content.

5.2 XML-to-Stream Operators

Since every previously described XQS example over streams is considered and evaluated as continuous query, than its result (in the iterations aggregation) can be considered as a stream too. The straightforward approach is to consider result of the continuous query as a stream obtained by concatenation of every iteration result. Nevertheless, such method has some obvious shortcomings that can be reduced only by possibility of the explicit output stream definition with required characteristics.

In order to illustrate the problem let's consider example that shows that straightforward concatenation can be useless in some cases. Example which performs join over two streams is shown in Query 7 (*a*). It includes two simple data-based windows that 'slide' (*i.e.* oldest item is dropped from the window's buffer and next received is inserted into it) simultaneously before the iteration's beginning.

Such evaluation and results' composing strategies lead to the possible existence of duplicate items in the output stream. Example of duplicates is shown in Figure 7 (*a*, *b*), which visualizes join of the two integer streams.

To sum up, XML-to-Stream operators were added

Query 7: Query example of the two streams join.

a – with potential duplicates

```
for $i in stream("items"){range 4}
for $b in stream("bids"){range 4}
where $i/itemno = $b/itemno
return <bid-with-info>
    <description>{ $i/description }
    </description>
    <bid-amount>{ $b/bid-amount }
    </bid-amount>
    </bid-with-info>
```

b – with duplicates elimination

```
istream(
  for $i in stream("items"){range 4}
  for $b in stream("bids"){range 4}
  where $i/itemno = $b/itemno
  return <bid-with-info>
      <description>{ $i/description }
      </description>
      <bid-amount>{ $b/bid-amount }
      </bid-amount>
      </bid-with-info> )
```

into XQS for a number of reasons that we have pointed briefly above:

- explicit output parameters definition (such as the duplicates absence, for example);
- explicit streaming and non-streaming context and operators separation;
- streaming and sliding windows operators' reusability inside queries.

There are two streaming operators, supported by XQS language – `istream`, which allows performing duplicates elimination, and `fstream` operator, which presents straightforward concatenation semantics described at the begin of this section. First is shown in Query 7 (*b*). It presents the same query, but in this case we can be fully confident in duplicate absence within the output stream.

6 Conclusion and Future Work

In this paper we have proposed a new language for the XML streams processing. Sliding windows and special streaming operators combined with well-known and powerful XQuery make XQS a powerful and very convenient language for querying unbounded XML streams with unpredictable behaviour. We have presented syntax, semantics and usage examples of new operations within this work.

Extensions to XQuery, incorporated by the XQS language, were successfully prototyped in the Sedna XML database system [15]. This implementation is based now on tuple-based algebraic approach, used in the Sedna system. We suppose it can be suitable almost for all streaming processing scenarios, especially for document-centric XML data. In future work we are planning to analyse capability of the XQS implementation with combined approach, described in this paper.

References

[1] D. Olteanu, T. Furche, F. Bry. Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity. Technical Report, Computer Science Institute, Munich, German, 2003

[2] B. Ludascher, P. Mukhopadhyay, Y. Papakonstantinou. A Transducer Based XML Query Processor. Proceedings of the 28th VLDB Conference, China, 2002

[3] Y. Diao, M. Altinel, M. Franklin, H. Zhang, P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. ACM Transactions on Database Systems, 28(4):467—516, December 2003

[4] SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, 24 June 2003, www.w3.org/TR/2003/REC-soap12-part1-20030624

[5] SportsML – Sports Markup Language. The International Press Telecommunications Council (IPTC), www.iptc.org

[6] Delayed Stock Quote. Web-service, www.xmethods.com

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and Issues in Data Stream Systems. Invited paper in Proceedings of the 2002 ACM Symposium on PODS, June 2002

[8] A. Levy, Z. Ives, D. Weld, Efficient Evaluation of Regular Path Expressions on Streaming XML Data, Technical report, University of Washington, 2000.

[9] H. Su, E. A. Rundensteiner, M. Mani. Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. In Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[10] XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999, http://www.w3.org/TR/xpath.html

[11] A. Gupta, D. Suciu. Stream Processing of XPath Queries with Predicates. SIGMOD, San Diego, June 2003.

[12] XQuery 1.0: An XML Query Language. W3C Candidate Recommendation 3 November 2005, http://www.w3.org/TR/2005/CR-xquery-20051103

[13] X. Zhang, K. Dimitrova, L. Wang, M. El-Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, E. A. Rundensteiner. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In SIGMOD Demo, page 671, 2003.

[14] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein and others. TelegraphCQ - Continuous Dataflow Processing for an Uncertain World. Proceedings of the CIDR Conference, Asilomar, CA, January 2003.

[15] M. Grinev, A. Fomichev, S. Kuznetsov, K. Antipin, A. Boldakov, D. Lizorkin, L. Novak, M. Rekouts, P. Pleshachkov."Sedna: A Native XML DBMS", Submitted to International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P), 2004.

[16] T. Urhan, M. Franklin. XJoin - Getting Fast Answers from Slow and Bursty Networks. University of Maryland, Technical Report CS-TR-3994, February 1999

[17] A. Arasu, S. Babu , J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Stanford University, Technical Report, October 2003

[18] R. Motwani, J. Widom, A. Arasu and others. Query Processing, Resource Management, and Approximation in a Data Stream Management

System. Proceedings of the CIDR Conference, Asilomar, CA, January 2003

[19] M. Sullivan. Tribeca: A Stream Database Manager For Network Traffic Analysis. In Proceedings of the 1996 VLDB, page 594, Sept. 1996.

[20] D. Terry, D. Goldberg, D. Nichols, B. Oki. Continuous Queries over Append-Only Databases. In Proceedings of the 1992 ACM SIGMOD, pages 321–330, June 1992.

[21] M. Altinel, M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination. In Proceedings of the 2000 VLDB, 2000.

[22] D. Olteanu, H. Meuss, T. Furche, F. Bry. XPath: Looking forward. In Proceedings of EDBT Workshop XMLDM, pages 109-127, 2002.

[23] X. Li, G. Agrawal. Efficient Evaluation of XQuery over Streaming Data. In Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

[24] XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Candidate Recommendation, 3 November 2005, http://www.w3.org/TR/2005/CR-xpath-datamodel-20051103/

[25] XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[26] Todd J. Green, Gerome Miklau, Makoto Onizuka, Dan Suciu. Processing XML Streams with Deterministic Automata. Technical report, University of Washington, 2001

[27] Tim Furche. Optimizing Multiple Queries against XML Streams. Diploma thesis, University of Munich, July 2003