Unifying the Processing of XML Streams and Relational Data Streams

Xin Zhou Hetal Thakkar Carlo Zaniolo
University of California, Los Angeles
Los Angeles, CA, 90095, USA
{xinzhou,hthakkar,zaniolo}@cs.ucla.edu

Abstract

Relational data streams and XML streams have previously provided two separate research foci, but their unified support by a single Data Stream Management System (DSMS) is very desirable from an application viewpoint. In this paper, we propose a simple approach to extend relational DSMSs to support both kinds of streams efficiently. In our Stream Mill system, XML streams expressed as SAX events, can be easily transformed into relational streams, and vice versa. This enables a close cooperation of their query languages, resulting in great power and flexibility. For instance, XQuery can call functions defined in our SQL-based Expressive Stream Language (ESL) using the logical/physical windows that have proved so useful on relational data streams. Many benefits are also gained at the system level, since relational DSMS techniques for load shedding, memory management, query scheduling, approximate query answering, and synopsis maintenance can now be applied to XML streams. Moreover, the many FSA-based optimization techniques developed for XPath and XQuery can be easily and efficiently incorporated in our system. Indeed, we show that YFilter, which is capable of efficiently processing multiple complex XML queries, can be easily integrated in Stream Mill via ESL user-defined and system-defined aggregates. This approach produces a powerful and flexible system where relational and XML streams are unified and processed efficiently.

1 Introduction

There is much current interest in processing streaming XML data, using queries expressed with languages such as XPath and XQuery [10, 14, 15, 28]. Meanwhile, a parallel line of research is focusing on the design of data stream management systems (DSMSs) that support continuous SQL queries over relational data streams [4, 13, 19, 20]. The integration of these two approaches is highly desirable because of the same considerations that now drive the seemingly unstoppable push to manage XML documents in traditional DBMSs. This strong drive for closer integration is created by the fact that many stream applications have to deal with both streaming relational and XML data, combine them, and publish results in any combination of the two formats. The technical benefits expected from this integration are also significant and include (i) consolidation of the two competing efforts now spent on system building and marketing and (ii) synergism between the two technologies by combining their respective areas of strength. For instance, research on streaming XML has produced advanced FSA-based techniques for supporting multiple complex queries on structured documents [14, 15, 28]. On the other hand, relational DSMSs are already providing solutions for many problems that remain unsolved in the framework of XML streams. For instance, DSMSs have shown how windows and other synopses are much needed in continuous queries to overcome the blocking behavior of traditional aggregates and to support efficient queries under limited memory [4, 19]. These constructs are now supported efficiently by all relational DSMSs and they will be very useful on XML streams as well.

In this paper, we present the design of a unified system that integrates the representation of the two kinds of streams and their processing. Our system supports continuous queries written in SQL, XQuery, and in a combination of the two languages—e.g., XQuery statements can use SQL:2003 OLAP functions and other user defined functions.

The architecture of our system was chosen after evaluating and discarding alternative designs. An obvious approach consists in building two subsystems—one for XML streams and the other for relational streams—and using pipes to communicate between them while converting between the two types of streams. This approach was discarded in favor of the approach presented in this paper that is preferable because it (i) avoids duplication of implementation efforts, and (ii) achieves tight integration of the data models and query languages. Our approach consists in extending a relational DSMS with efficient support for XML streams and XQuery. For this purpose, we use Stream Mill [12] and its Expressive Stream Language (ESL) that complies with SQL:2003 standards, but also supports efficiently user-defined aggregates because of their proven ability to express very complex queries [11, 29].

Our approach to support XML streams consists in encoding SAX events as relational tuples, and then using ESL to express the wide variety of tasks that integrate relational and XML streams. Queries expressed in XPath or XQuery are thus compiled into equivalent ESL programs that call on user-defined aggregates (UDAs) or system-defined aggregates (SDAs). In the rest of the paper, we describe this approach and the many benefits it offers from an application viewpoint. We also show that the various FSA-based optimization techniques proposed for XPath/XQuery [14, 15, 28] can be supported well in our approach, by either UDAs natively defined in ESL, or UDAs defined in procedural programming languages, or SDAs built into the system. In particular, we discuss ESL implementation of YFilter, which supports the parallel processing of multiple XQuery statements, without any extension to current relational query language.

1.1 Organization of the Paper

The rest of the paper is organized as follows. After a short review of related work in Section 2, Section 3 discusses how we unify the two kinds of data streams with ESL. Section 4 explores continuous queries for which a close integration of relational and XML query languages is desirable—e.g., to allow XQuery to support analytics on data streams. Section 5 explains ESL UDAs in detail. In Section 6, we describe the FSA-based processing of multiple XQuery/XPath statements using UDAs and SDAs. The overall system implementation is studied in Section 7. Section 8 concludes our discussion.

2 Related Work

There is much ongoing research work on data streams and continuous queries [13, 19, 20, 26]. CQL [4] introduces several stream-oriented concepts and SQL-based constructs. Windows represent a very important construct in CQL, insofar as they are used in the computation of aggregates and joins on streams, and to map from streams to relations. Windows have also found many applications in traditional databases and are now part of the new OLAP Function standards of SQL:2003 [1, 6], which are supported in most commercial DBMSs.

In the meanwhile, the need for processing streaming XML data is growing fast, propelled by a strong demand from applications [14, 28], in which data aggregation, time-windows, and path expressions frequently need to be supported. XQuery (which embeds XPath) is expected to be the language of choice for these applications. Many navigation-based techniques to answer queries on streaming XML input have been proposed, with different application foci. Techniques explored include single-query processing [30, 15], processing of multiple XPath expressions [28], processing of XPath for twig-queries [9], predicate evaluation [7], backward-axes handling [8], and streaming queries with disk-resident index assistant [21]. In comparison, until now little has been proposed on the problem of integrating relational and XML streams. In the approach proposed in [22], the stream of XML tokens is viewed as multiple punctuated relational streams, on which path and twig matching can be supported via specialized stream joins.

There is also ongoing research on the features of XQuery language itself. The basic XQuery has shown to be effective at expressing simple streaming applications, such as those discussed in [14]. User-defined functions enable native extensibility in XQuery, which makes the language Turing-complete [25] and thus capable, at least in principle, of expressing every possible application on stored data. Unfortunately, it lacks support for basic grouping functionalities which are normal in SQL, not to mention complex OLAP queries, and non-blocking queries on streaming data. There has been much research work in this area [5, 18]; for instance, Natix [27] provides a tuple-based algebra that includes grouping operators for construction of XML elements. The addition of a "group by" clause to XQuery to better support OLAP applications and analytics was proposed in [17, 24]. However, these extensions focus on querying stored data, as opposed to data streams on which only non-blocking aggregates are allowed.

3 Unified View for Relational and XML Data Streams

In this section, we discuss how to create, query, and unify relational and XML data streams. All of these objectives are realized by Expressive Stream Language (ESL), which is the application language of the Stream Mill system that supports:

- Continuous queries (CQ) on data streams,
- Ad hoc queries on database tables, and
- Spanning applications that combine and compare streaming data with stored data.

To facilitate the learning of the language and its use on spanning applications, ESL is based on SQL and minimizes extensions from its SQL:2003 standards [12, 11].

3.1 Relational Data Streams

In the Stream Mill system, each data stream is imported from an external wrapper via the (mandatory) **SOURCE** clause in its **`CREATE STREAM`** declaration.

For example, consider a hypothetical online web store for book auctions, where both new and used books are traded. Bidders can place bid on books with a certain BookID. Let us assume that bids for used books arrive as the relational data stream declared as follows:

Example 1. *Relational data streams definition.*

```
CREATE STREAM UsedBookBidStream (BookID int,
    BidderID char(10), BidPrice real, BidTime timestamp)
SOURCE 'port4445'
```

Example 1 above uses a wrapper (SOURCE 'port4445') that is created automatically by the system for each port used in the program. Rather than using these defaults, users can easily create their own wrappers as described in [12].

New streams can be defined and transduced from existing streams, in a fashion similar to that used to define virtual views in SQL. For instance, to derive a stream consisting of the bids where the bidding price is above 200, we can write:

Example 2. *Performing Selection operation on streams.*

```
CREATE STREAM expensiveItems AS
    SELECT BookID, BidderID, BidPrice, BidTime
    FROM UsedBookBidStream WHERE BidPrice > 200
```

3.2 XML SAX Event Data Streams

Say that, besides the relational stream of bids on used books, there is another stream of XML messages, which records the bidding information on new books. A sample bid may look like that of Figure 1, below:

```
<Bid  BidTime = "2005/02/25T13:24:34">
    <BookID>100001</BookID>
    <BidderID>TC0027</BidderID>
    <BidPrice>65.00</BidPrice>
</Bid>
```

Figure 1. A sample XML message for a bid

These streaming XML bids are parsed using the Simple API for XML (SAX); a standard interface to parse streaming XML [23], which provides a sequential view of an XML document through a stream of events. In Stream Mill system, we then represent the SAX stream using a triplet-based format, (*event, name, value*), that we call SAX-3. Thus, the following SAX-3 relational stream is generated, by a SAX based XML parser, from the XML stream of Figure 1:

```
('start', 'Bid', _),
('attr', 'BidTime', '2005/02/25T13:24:34'),
('start', 'BookID', _)
('text', _, '100001'),
('end','BookID', _),
    ...
('end', 'Bid', _)
```

Here, 'start' denotes the start-of-element event, and 'end' denotes the end-of-element event. In these two triplets, the *name* column contains the element name, and the *value* column is null. The second triplet shows that each attribute of an element is represented by an 'attr' event with the remaining two columns storing the attribute's name and value, respectively. Another special event is 'text', for which the second column is null and the third column contains the actual text.

The following statement defines a SAX-3 event stream in ESL:

Example 3. *XML SAX-3 event stream definition.*

```
CREATE STREAM SAX-3-Events ( event varchar(10),
    name varchar(50), value varchar(50) )
SAXSOURCE 'port4448'
```

The **`SAXSOURCE 'port4448'`** clause in this stream definition specifies the port where we have a special wrapper to "wrap" XML SAX events. The wrapper basically takes in the SAX events from the SAX based XML parser and transforms the events into (*event, name, value*) triplet structures.

In addition to this 'vanilla' wrapper, Stream Mill can support more specialized wrappers for more efficient and compressed representation of SAX events. For example, the length of data type **`varchar`** can be adjusted if the DTD or XML Schema is available. Furthermore, for element like **`<BookID>`**, which is a leaf element containing only plain text, we can combine the two consecutive SAX-3 event tuples; for instance ('start', 'BookID', _) and ('text', _, '100001') can be merged into tuple: ('start', 'BookID', '100001').

In Stream Mill, we can use arbitrary XQuery FLOWR statements to write continuous queries on XML streams. Such queries take SAX-3 events as input and return SAX-3 events as output. For instance, to derive a stream consisting of the bid XML messages where the bidding price is above 200, we can write:

Example 4. *Simple XQuery on a SAX-3 event stream.*

```
CREATE STREAM NewSAX-3-Events AS (
    FOR $b IN Stream(SAX-3-Events)//Bid
    WHERE xs:real($b/BidPrice) > 200
    RETURN ( $b ) )
```

Queries written in XQuery return SAX-3 event streams, and output wrappers can then be used to produce streaming XML documents from these.

The support for XPath/XQuery statements in ESL will be discussed in Section 6, where we also discuss the role of UDAs in the FSA-based parallel processing of multiple XML queries.

3.3 From SAX Streams to Relational Streams

SAX-3 event streams are normal relational streams of triplets and thus can be processed using ESL. This allows us to support applications where XML streams and relational streams must be combined. For instance, suppose we need to merge the bid streams of old books and new books. We need to transform the XML-structured new-book bids of Figure 1 into the 4-column flat relational format of Example 1. However, the bids on new books are first transformed into a stream of SAX-3 events, thus we transform these SAX-3 events to the relational tuples, such as (100001, 'TC0027', 65.00, '2005/02/25T13:24:34'). This transformation in SQL would require four self-joins, which is an expensive operation, of the SAX-3 stream. However, this operation can be easily expressed and efficiently implemented via user-defined aggregates (UDAs) supported in ESL, as shown in Example 5. (The actual definition of `Flatten()` is given in Section 5.2.)

Example 5. *Creation of relational data streams out of XML SAX-3 event streams.*

```
CREATE STREAM NewBookBidStream AS (
    SELECT Flatten(event, name, value)
    FROM SAX-3-Events)
```

An immediate benefit of this flattening is that the streams for used books, `UsedBookBidStream`, and new books, `NewBookBidStream`, can now be merged using the union operator to support continuous queries on bids for both new and used books.

Union is a very important operation on data streams, and can often be used to avoid join operations that might require unbounded memory in extreme cases, or return approximate results by finite window sizes. For instance, suppose that we have a relational stream `CloseStream(BookID, CloseTime)` describing closing auctions. Since we have `UsedBookBidStream`, defined in Example 1, for used books and `NewBookBidStream`, defined in Example 5, for new books, we can now get the closing price of each book as soon as its auction closes. The following example illustrates our approach.

Example 6. *Return the winning bid's price when a certain auction is closed.*

```
CREATE STREAM BidStream (BookID, BidderID,
    BidPrice, BidTime) AS (
    SELECT BookID, BidderID, BidPrice, BidTime
    FROM UsedBookBidStream
    UNION
    SELECT BookID, BidderID, BidPrice, BidTime
    FROM NewBookBidStream );

CREATE STREAM AuctionBehavior (behavior, id,
    price, time) AS (
    SELECT 'close', BookID, Null, CloseTime
    FROM CloseStream
    UNION
    SELECT 'bid', BookID, BidPrice, BidTime
    FROM BidStream;

SELECT PriceReport(behavior, id, price, time)
FROM AuctionBehavior
```

In this example, bid streams for both used and new books are first unioned into one single stream `BidStream`, which is then unioned with `CloseStream` into a new stream `AuctionBehavior`, ordered by its `time` value. The resulting stream is finally passed to the UDA `PriceReport`, which basically returns the closing price of an auction upon reading the "close" event for the auction (see Section 5.4 for details).

Conversely, we can also transform relational data streams into XML SAX-3 streams by using UDAs. This makes it possible to apply to relational data streams, the FSA-based techniques for processing multiple XPath/XQuery statements in parallel that will be discussed in Sections 5.3 and 6.2.

4 Querying Integrated Data Streams

In Section 3, we outlined how to use ESL to generate relational streams from XML streams, and vice versa. But, in addition to integrating these two types of data streams, we also want to achieve the cooperation of their respective query languages. In particular, we would like to enable XQuery to take full advantage of the advanced OLAP functions now available in SQL:2003.

4.1 Continuous Query Support with ESL

Windows play a major role in relational DSMS and are supported by most data stream systems. ESL adopts the window constructs specified in SQL:2003 OLAP function standards to support continuous queries on windows. These constructs can now be applied on the relational streams, the SAX-3 streams, and the streams derived from those as illustrated by the following example:

Example 7. *Continuously return the total number of bids in the last 10 minutes.*

```
SELECT current_time, COUNT(BidPrice)
        OVER (RANGE 10 MINUTE PRECEDING)
FROM BidStream
```

Here, `BidStream` is the unified data stream created in Example 6, `OVER` is a standard SQL:2003 construct to create a moving window, and `current_time` is a key word representing the current system time.

Thus, Example 7 defines a logical window (i.e., a time-based window) on the input stream. Example 8 shows another important construct used in continuous queries: the count-based physical window.

Example 8. *Continuously return the average bid price of last 5 bids for every book.*

```
SELECT current_time, BookID, avg(BidPrice)
OVER (PARTITION BY BookID ROWS 5 PRECEDING)
FROM BidStream
```

An important advantage of Stream Mill over other DSMSs is that it supports windows on arbitrary UDAs, not just built-in aggregates. Thus, the aggregate avg() in Example 8 can be replaced by any UDA, to express more advanced queries, as we will see in Section 5.4.

4.2 ESL-Defined Functions in XQuery

Incompleteness of XQuery on Streaming Data. The native definition of user-defined functions allows XQuery to achieve Turing-completeness [25]. Unfortunately, the function-definition mechanism of XQuery is blocking and thus not suitable for streaming data. This can be illustrated by the following example showing the definition and invocation of a count-like aggregate:

Example 9. *Return the total number of bids.*

```
DECLARE FUNCTION mycount($bids as xs:AnyType)
AS xs:integer {
    IF (empty($bids)) THEN 0
    ELSE mycount(subsequence($bids,2))+1 }

LET $a IN STREAM(“AllBids.xml”)/Bids/Bid
RETURN mycount($a)
```

The sequence (i.e., the list) of all bids is given to the XQuery function, that then applies the count function to its tail (i.e., the subsequence starting from the second element). As shown in this example, XQuery functions assume that the whole sequence is present and materialized before the computation is started. This computation model is blocking and will not work when we have an infinite stream of records. Namely, the native extensibility mechanisms provided by the current XQuery standards cannot be used to define online aggregates or aggregates using windows such as that of Example 7. The problem is further exacerbated by the lack of explicit grouping constructs in XQuery [17]. Queries with complex GROUP BY/PARTITION BY clauses are actually very difficult to write in XQuery. Moreover, while queries with simple groupings can be readily written in XQuery, they are difficult to implement without multiple scans of the original document [17].

Curing the Incompleteness of XQuery on Streaming Data. Therefore, the superior expressive power of XQuery evaporates when processing streaming data. One approach to solve this problem consists in introducing new constructs into XQuery—possibly in a fashion akin to the GROUP BY constructs recently proposed in [17] to assure the effective support of analytics in XQuery. However, besides requiring the addition of new constructs to the current standards, this solution will also complicate the current evaluation model of XQuery FLOWR expressions, which, unlike the XQuery function-definition mechanism, is quite amenable to stream-oriented processing. Therefore, we will instead take the more natural approach of keeping the FLOWR constructs intact and explore mechanisms to introduce non-blocking user-defined functions and aggregates in XQuery. Our solution exploits the fact that XQuery can accept functions defined in external languages—including nonblocking functions. In other systems, the functions to be imported into XQuery would be written in C++ or Java; but the Stream Mill system closely integrates XQuery and ESL and thus allows the importation of SQL:2003 analytical functions as well as window aggregates directly from ESL. This produces a significant simplification for the user and the system alike.

Say for instance that we want to support the same window queries as specified in Section 4.1, on the XML stream `AllBids.XML` consisting of messages similar to Figure 1, all collected under a root element `Bids`. Then, we can use the following statements:

Example 10. *Continuously return the total number of bids in the last 10 minutes (XQuery version of Example 7).*

```
DECLARE FUNCTION mycount($price as xs:real,
    $time as xs:time)
AS ELEMENT (Total_Bids) {
    RETURN SQLXML (
        SELECT XMLElement (Name “Total_Bids”,
            XMLElement (Name “Current_Time”, current_time),
            XMLElement (Name “Number”, count(T.price)
                    OVER (RANGE 10 Minute PRECEDING)))
        FROM STREAM ($price, $time) AS T (price, time) )}

FOR $a IN STREAM(“AllBids.xml”)/Bids/Bid
LET $b := $a/BidPrice
LET $c := $a/BidTime
RETURN mycount($b, $c)
```

In our XQuery extension, the keyword **`SQLXML`** specifies that the XQuery function is actually defined using SQL/XML, which has become a popular XML publishing standard for relational databases [2]; SQL/XML adds to SQL specialized publishing functions for structured output support. Thus we can now call function **`mycount ($b, $c)`**, to return the current total number of bids for the last 10 minutes. Note that Example 10 is the same as Example 7—but it is defined in extended XQuery and returns XML stream.

This extension also allows supporting GROUP BY in XQuery as shown in Examples 11 and 12, below.

Example 11. *Continuously return the average bid price of last 5 bids for every book (Same as Example 8).*

```
DECLARE FUNCTION myavg($id as xs:integer,
    $price as xs:float)
AS ELEMENT (Avg_Price) {
    RETURN SQLXML (
        SELECT XMLElement (Name “Avg_Price”,
            XMLElement (Name “BookID”, T.id),
            XMLElement (Name “Avg”, avg(T.price)
                OVER (PARTITION BY T.id ROWS 5 PRECEDING)))
        FROM STREAM ($id, $price) AS T (id, price) )}

FOR $a IN STREAM(“AllBids.xml”)/Bids/Bid
LET $b := $a/BookID
LET $c := $a/BidPrice
RETURN myavg($b, $c)
```

Using the extended function definition capabilities proposed here for XQuery, all the complex OLAP queries discussed in [17] can readily be supported. Consider for instance the following query from [17]:

Example 12. *Return the average bid price by book (BookID), bidder (BidderID), (BookID, BidderID), and overall.*

```
DECLARE FUNCTION myavg($b as xs:AnyType) {
RETURN SQL
    (SELECT S.BookID, S.BidderID, avg(S.Price)
    FROM Stream ($b/BookID, $b/BidderID, $b/BidPrice)
        AS S (BookID, BidderID, BidPrice)
    GROUP BY CUBE(S.BookID, S.BidderID))}

LET $b IN STREAM("AllBids.xml")/Bids/Bid
RETURN myavg($b)
```

As pointed out in [17], this simple CUBE query can only be expressed in current XQuery by a complex query that scans the input several times to construct the grouping sets. The solution proposed in [17] consists in adding a new construct GROUP BY to the current FLWOR standard of XQuery. The simpler solution proposed here is that of allowing XQuery functions to invoke ESL aggregates. Besides built-in aggregates such as avg() and count(), arbitrary ESL UDAs can also be invoked in this fashion.

5 Defining ESL UDAs

In this section, we discuss ESL user-defined-aggregates (UDAs) in more detail and underline the important role they play in unifying the processing of XML and relational streams. Then, in Section 5.4, We discuss the use of UDAs in more advanced queries.

5.1 Introduction to UDAs

Native definition of UDAs is the kernel of ESL. It empowers ESL to handle more complex streaming queries via a small extension of SQL. Let us use the following very simple example to illustrate the basic idea, which comes from [16]:

Example 13. *Definition of UDA equivalent to standard avg aggregate.*

```
1 AGGREGATE avg(next REAL) : REAL {
2     TABLE state(sum REAL, count REAL);
3     INITIALIZE: {
4         INSERT INTO state VALUES (sum, 1); }
5     ITERATE: {
6         UPDATE state
            SET sum = sum + next, count = count + 1; }
7     TERMINATE: {
8         INSERT INTO RETURN
            SELECT sum/count FROM state; } }
```

A new UDA is specified by providing definition of an INITIALIZE computation (line 3-4), an ITERATE computation (line 5-6), and a TERMINATE computation (line 7-8), in a single procedure written in SQL. Example 13 defines an aggregate equivalent to the standard AVG aggregate in SQL. Line 2 declares a local table, **state**, where the sum and count of values processed so far, are kept. Furthermore, while in this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. Here, INITIALIZE block inserts the values taken from the input stream into the **state** and sets the **`count`** to 1. The ITERATE block updates the tuple in **state** by adding the new input value to the **`sum`** and 1 to the **`count`**. The TERMINATE block returns the ratio between the sum and the count as the final result of the computation by the "INSERT INTO RETURN" statement in line 8. The TERMINATE block is executed right after all the input tuples have been processed. The definition of all three statement blocks, INITIALIZE, ITERATE, and TERMINATE, in one procedure allows supporting the declaration of their shared tables (the **state** table in this example). Because UDAs can also be supported on moving windows, this avg() UDA can actually be used in Example 7 of Section 4.1.

In fact, we can move the "INSERT INTO RETURN" statement to the INITIALIZE and ITERATE blocks from the TERMINATE block, and make the TERMINATE block empty. This modified UDA returns the average of the tuples seen so far, after processing each tuple from the input. This new UDA is actually a non-blocking UDA, which is clearly identified by the fact that its TERMINATE block is empty. This generic aggregate definition mechanism allows tremendous flexibility and power. In fact, ESL with UDAs is Turing-complete and allows natively specifying complex mining functions on data streams. Advanced applications of UDAs are shown in [16].

5.2 Defining a UDA to Flatten XML Streams

In Section 3.3, we discussed how to use ESL to transform XML SAX-3 event streams into regular relational streams, where a Flatten() UDA is called in ESL to do the transformation.

To achieve this, what we need is to detect an attribute event for **`BidTime`**, and three consecutive start of element events for **`BookID`**, **`BidderID`**, and **`BidPrice`**. Then, we flatten their values into one tuple, which follows the relational schema of Example 1, whereas all other SAX-3 events are discarded. (The UDA assumes that the SAX stream wrapper already wraps the text value of a leaf element into its **`value`** field, thus we get SAX-3 events such as ('start', 'BookID', '100001')).

Example 14 defines such a UDA to flatten SAX-3 events into relational tuples. In this UDA, a local table **Temp** is defined, where the BookID, BidderID, BidPrice and BidTime values are kept.

Example 14. *Definition of UDA to flatten SAX-3 event streams into relational streams.*

```
AGGREGATE Flatten (event varchar(10),
    name varchar(50), value varchar(50))
    : (int, char(10), real, timestamp) {
    TABLE Temp(BookID int, BidderID char(10),
        BidPrice real, BidTime timestamp);
    INITIALIZE: { }
    ITERATE: {
        INSERT INTO Temp VALUES (Null, Null,
            Null, CAST(value, timestamp))
            WHERE event = 'attr' AND name = 'BidTime';
        UPDATE Temp SET BookID = CAST(value, int)
            WHERE event = 'start' AND name = 'BookID';
        UPDATE Temp SET BidderID = value
            WHERE event = 'start' AND name = 'BidderID';
        UPDATE Temp
            SET BidPrice = CAST(value, real)
            WHERE event = 'start' AND name = 'BidPrice';
        INSERT INTO RETURN
            SELECT * FROM Temp
            WHERE BookID != NULL
            AND BidderID != NULL
            AND BidPrice != NULL
            AND BidTime != NULL;
        DELETE FROM Temp WHERE SQLCODE = 0; }
    TERMINATE: { } }
```

In the ITERATE block of this UDA, a new tuple is inserted in the **`Temp`** table when a **`BidTime`** attribute event is read, and its value (casted to the correct data type) is stored in the **`BidTime`** field. When the input is the start of element for either **`BookID`**, or **`BidderID`**, or **`BidPrice`**, the corresponding field of the tuple is updated. The tuple in **`Temp`** will be returned once each field has been filled with a value. Once a tuple in **`Temp`** has been returned (an event that sets the system variable SQLCODE to 0), it is no longer needed and it is deleted.

The TERMINATE block is empty in this example, since satisfied tuples have already been returned in ITERATE. Therefore, this UDA is non-blocking; indeed it serves as a stream transducer, where streams are piped in and piped out.

Similar UDAs can be defined to flatten any number of elements. For example, if we are only interested in bid price and time, we can write a UDA which generates (BidPrice, BidTime) pairs and feeds into another 2-field relational stream.

5.3 Generating XML Streams from Relational Streams Using UDAs

As discussed in previous sections, relational UDAs are based on SQL, and return relational streams. In Section 5.2, we showed how to define a UDA to transform XML SAX-3 event streams into relational streams, in order to unify the two kinds of data streams. Similarly, we must be able to transform relational streams to XML streams. In general, two approaches can be taken to address this requirement.

The first approach consists in using SQL to assemble and return SAX-3 events in the output. For the simple UDA of Example 13, this involves returning three tuples: a start tuple, a value tuple, and an end tuple (assuming that no attribute is needed). The triplets produced by this UDA upon invocation (on a window, since this is a blocking UDA) can be passed to an output wrapper, which takes in the SAX-3 event streams and outputs XML stream messages.

The second approach consists in taking advantage of the fact that SQL/XML standards are supported in ESL. As a result, we can use SQL/XML to output XML-structured elements encoded as SAX-3 streams. For Example 13, for instance, we can just replace its current TERMINATE code with:

```
INSERT INTO RETURN
SELECT XMLElement(Name "Avg", sum/count)
FROM state;
```

This statement takes the output values and restructures them to produce XML SAX-3 triplets. This second approach leads to simpler statements and avoids the risk of malformed XML document.

5.4 Advanced Queries on Data Streams

ESL also supports advanced sequence and time-series queries that represent another important application area for data streams.

Example 15. *Continuously return all the patterns of two consecutive bids that raise the previous bid by at least 50%.*

```
SELECT BookID, IncreaseDetect(BidPrice)
FROM BidStream GROUP BY BookID;

AGGREGATE IncreaseDetect (BidPrice real) : real {
    TABLE temp (price real, rank int);
    INITIALIZE: {
        INSERT INTO temp
            VALUES (BidPrice, 1), (BidPrice, 2); }
    ITERATE: {
        INSERT INTO temp VALUES (BidPrice, 3);
        INSERT INTO RETURN
            SELECT t1.price, t2.price, t3.price
            FROM temp t1, temp t2, temp t3
            WHERE t1.rank = 1 AND t2.rank = 2
            AND t3.rank = 3 AND t1.price * 1.5 <= t2.price
            AND t2.price * 1.5 <= t3.price;
        DELETE FROM temp t WHERE t.rank = 1;
        UPDATE temp t SET t.rank = t.rank -1; }
    Terminate: { } }
```

In this example, the **`BidPrice`** is extracted from the original **`BidStream`** and passed to the UDA **`IncreaseDetect`**, which detects price increases. A temporary table **`temp`** holds the last three bids; when two consecutive bid prices exceed the previous bid by 1.5, the UDA returns the content of **`temp`**, i.e., the sequence of three successive price increase by 50% or more. This example illustrates the ability of UDAs to support state-based computations. Without UDAs, or similar state-based operators, detecting a sequence of n events normally takes $n - 1$ self-joins, which is a very complicated operation on data streams.

Let us now return to Example 6 in Section 3.3, where we pass the combined stream **`AuctionBehavior`** to a PriceReport() UDA to determine the winning price. The definition of this UDA is given below:

Example 16. *Return the winning bid's price when a certain auction is closed (UDA which is called in Example 6).*

```
AGGREGATE PriceReport (behavior varchar, id int, price real, time
timestamp) : (int, real, timestamp) {
    TABLE temp (item int, currentprice real);
    INITIALIZE: {
    ITERATE: { }
        UPDATE temp
            SET currentprice = price
            WHERE behavior = 'bid' and item = id;
        INSERT INTO temp VALUES (id, price)
            WHERE behavior = 'bid' AND SQLCODE != 0;
        INSERT INTO RETURN
            SELECT t.item, t.currentprice, time
            FROM temp t
            WHERE behavior = 'close' AND t.item = id;
        DELETE FROM temp t
            WHERE SQLCODE = 0 AND t.item = id; }
    Terminate: { } }
```

In Example 16, the first two SQL statements in ITERATE are used to update current highest bid price for each item. The last two SQL statements in ITERATE are used to output the winning price when the auction of a certain item closes. UDAs such as those in Examples 15 and 16 are non-blocking, and can be used directly in ESL, or in XQuery functions as discussed in Section 4.2.

6 FSA-Based Support for XPath/XQuery

In Sections 3.3 and 5.2, we discussed how to transform XML streams into relational streams, where the input XML messages are flattened into relational tuples, as shown in Examples 5 and 14. Here, the schema for the input XML, consisting of bids as those of Figure 1, is rather simple and regular, and our queries are also simple. In general, we must deal with complex XML structures and arbitrary queries.

```
<Auction>
 <Book BookID = "100001">
  <Title>A Complete Guide to DB2</Title>
  <Author>Don Chamberlin</Author>
  <Content>
    <Chapter>
      <Title>Introduction</Title>
    </Chapter>...
  </Content>
 </Book>
</Auction>
```

Figure 2. A sample XML message for an auction

For instance, suppose a user is interested in writing queries to extract 'books which have a **`<Title>`** descendant element containing keyword "DB2"', from XML messages such as that of Figure 2. The user would normally prefer to express his/her query in XQuery/XPath, rather than writing a flatten UDA such as that of Example 14. Indeed, Stream Mill allows users to write continuous queries directly using XQuery/XPath. These queries are then supported using UDAs that implement the FSA-based approach of YFilter [28] for performance and parallel execution.

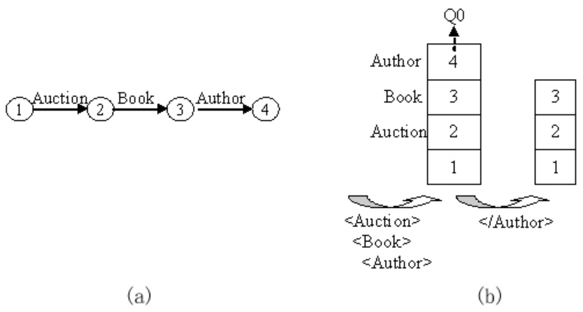

Figure 3. FSA for Q0: */Auction/Book/Author*

6.1 UDA Simulation of Basic FSA

We will begin with a very simple example to explain the idea of FSA-based XPath processing as in [28], using UDAs written in SQL. For better performance these aggregates have eventually been implemented as system-defined aggregates (SDAs), that make full use of Stream Mill's internal optimization techniques.

Suppose that an XPath query Q0, */Auction/Book/Author*, is issued by the user to filter the input documents. A simple FSA can be built to simulate the processing of Q0 based on the input SAX events, as in Figure 3 (a), where the name on every edge represents the triggering element name between two states.

A runtime stack is maintained to keep the active states while we read in SAX events. The transition between states is triggered by the start event of an element, whereby new active states are pushed on top of the stack; similarly, the end event will perform backtracking by popping out the active states from the top of the stack. Every time we reach the accept state, we find an answer to the XPath and can return it on the fly. The runtime stack based on the XML input of Figure 2 is shown in Figure 3 (b).

Based on the non-blocking calculation feature of relational UDAs, as well as its Turing-complete expressive power, we aim at building a simulation for the execution of FSA, where we can bridge the different computational styles of FSA and SQL, and share them under the same relational data model.

We use (i) a **Transition** table to store the transition graph, such as that of Figure 3 (a), and (ii) a **State** table to store the states of FSA (in the **`SType`** column, 'i' means initial state, 'm' means middle state, and 'a' means accept state). The table schema and sample content are as follows:

State Table

SID	SType	QID
1	i	-
2	m	-
3	m	-
4	a	Q0

Transition Table

SrcID	symbol	DestID
1	Auction	2
2	Book	3
3	Author	4

Observe that in **State** table above, only the accept state has a **QID** value giving the ID for the satisfied query when the FSA reaches that state. Here, again, we will generate a SAX-3 event stream as a result of incoming XML messages.

Transition and backtracking are done in the ITERATE state for every input event in **SAX-3** stream. The basic UDA definition is shown in Example 17; observe that the code in every state can be implemented in just one or two SQL statements.

Example 17. *UDA simulation for simple XPath queries.*

```
Table State (SID int, SType char(10), QID char(2));
Table Transition (srcID int, symbol char(50), destID int);

Aggregate FSA (event char(10), name char(50), value char(10)):
(QID char(2)) {
    TABLE RuntimeStack (level int, SID int);
    TABLE StackTop (level int); /*Used to decide the stack top*/
    INITIALIZE: {
        /* Push initial state in State table into
            RuntimeStack table*/ }
    ITERATE: {
        /*start of element handler*/
            /* If (the top state in RuntimeStack is 's1',
                    AND input event is 'start',
                    AND input event name is 'e'
                    AND entry (s1, e, s2) is in Transition table)
                Push s2 on top of RuntimeStack */
        /*decide accept state*/
            /* If (the top state in RuntimeStack is an "accept" state
                    in State table)
                Return the QID of that state in State table */
        /*end of element handler*/
            /* If (the input event name is 'end')
                Delete the top state in RuntimeStack table */}
    TERMINATE: { } }
```

This UDA only returns the IDs of the queries that are satisfied, but this basic scheme can be extended to output SAX-3 events of matched elements.

Notice that TERMINATE state is empty in the previous UDA, and satisfied queries are returned on the fly. Therefore, this UDA is actually a non-blocking UDA, which pipelines the input and the output streams. This is exactly what we need for streaming XML processing.

6.2 Multiple Complex XPath/XQueries

The previous section described how to filter XML documents for a simple XPath query (/Auction/Book/Author), however several extensions are required to enable parallel processing of more complex XPaths as in YFilter.

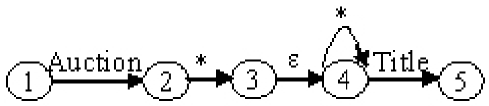

Figure 4. FSA for */Auction/*//Title*

6.2.1 Multiple XPath Queries with Wild-card '*' and Descendent Axis '//'

As discussed previously, the XPath statement /Auction/Book/Author is translated into the FSA of Figure 3(a), where '/Auction' corresponds to the transition from state 1 to state 2. The presence of wild cards in our query statements do not change this overall translation scheme by much; for instance, if the above query is changed to /*/Book/Author, then the transition from state 1 to state 2 in Figure 3(a) will have '*' as its label. However, as pointed out in [28], the use of the descendant axis brings additional complications to this translation. For instance, if the above query is changed to //Auction/Book/Author, then an extra state, state 0, is needed, along with a transition from state 0 to state 1 with label ϵ, and a transition from state 1 to itself with label '*'. Furthermore, in this case, state 0 will become the start state of the FSA. Thus, integration of these advanced constructs requires some extensions to the FSA translation process. This is also illustrated by the example shown in Figure 4 that gives the FSA for XPath /Auction/*//Title.

The ability of supporting parallel processing of a large number of queries represents an important technology developed as a result of several research efforts on streaming XML data. In particular, Yfilter [28] presents a technique for combining FSA that solves this problem efficiently, by assuring that the prefixes of the different XPath statements are combined and shared as much as possible. As shown in [29], SQL extended with UDAs is Turing Complete, and can thus easily support UDA that represents such combined FSA. We use the Transition table of Example 17 to record the transitions of the combined FSA. This Transition table can be built incrementally as new queries are added/deleted. The addition of the wild-card, descendant axis, and parallel processing of multiple XPaths require small extensions to the UDA of Example 17; the details are omitted here due to space constraints. This combined-FSA UDA presents many opportunities for optimization, for instance use of indexes and memory tables. Furthermore, the UDA can also be supported as a system built-in function, in which case it can be written in an external programming language and can take advantage of specialized optimization techniques.

6.2.2 XPath/XQueries with Branch Queries

Besides the simple linear XPaths we have discussed above, our XPath statements can actually contain structure-based or value-based predicates. For example,

/Books/Book[Author]/Title will return book titles only for those books that contain author information. This XPath expression can be divided into the two subexpressions Q1: /Books/Book/Author, and Q2: /Books/Book/Title. Before a result can be returned to the output, the sub-results produced by these two subexpressions must be joined on their shared **Book** node.

Building on the techniques presented above, we can now extend our UDAs to process complex XPath and XQuery statements by decomposing each query into separate XPaths. But in this case, in addition to returning the satisfied query ids and the corresponding SAX-3 events, we also need to return, for each matched element, an internal element id that is needed later for joins. Such unique id can be easily maintained by creating an id table with just one tuple, and incrementing it every time a new start-of-element event is detected. For instance, say that the queries Q1 and Q2 discussed above are issued on the following XML input, where the number associated with each start element represents the internal element id:

$< Books >^1$
  $< Book >^2$
    $< Title >^3$ A Complete Guide to DB2 $< /Title >$
    $< Author >^4$ Don Chamberlin $< /Author >$
    $< Content >^5$...$< /Content >$
  $< /Book >$
  $< Book >^6$
    $< Title >^7$ Advanced Database Systems $< /Title >$
  $< /Book >$
$< /Books >$

The result set for Q1 (/Auction/Book/Author) contains only one match, namely, {1-2-4}, where the numbers represent the satisfied element ids. Similarly, the result set for Q2 (/Auction/Book/Title) is {1-2-3, 1-6-7}. However, since the two XPaths need to be joined on their second element, 1-2-4 and 1-2-3 match and 1-2-3 is returned as output, but there is no match for 1-6-7, which thus produces no output.

Our basic UDA can be easily extended to cope with this join requirement, and return all the satisfied element ids for each XPath, as well as the SAX-3 events for the output. All those element ids are then passed to another UDA to perform the join operations and output the SAX-3 events that satisfy the query. Likewise, XQuery statements are first mapped into several XPath statements joined on some common nodes, and then processed with techniques similar to those discussed above.

7 System Implementation and Performance

The Stream Mill system unifies relational data streams and XML streams using the architecture outlined in Figure 5. Stream Mill supports the conversion of XML SAX-3 events to flat relational tuples and vice versa, and this paves the way to a closer cooperation of relational and XML query languages, resulting in greater power and flexibility.

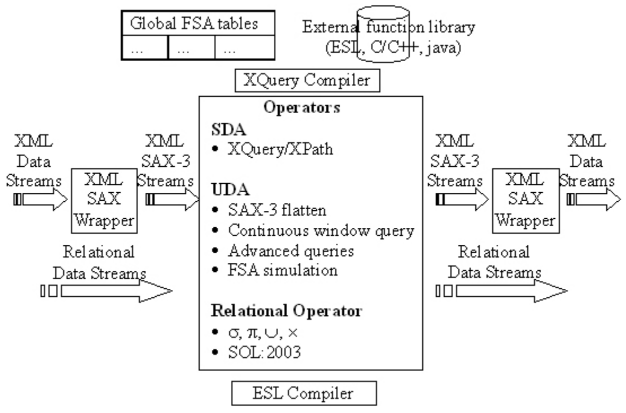

Figure 5. Architecture for unifying XML data streams and Relational data streams

Stream Mill supports normal selection, projection, union, and special join operations on relational data streams. The system also integrates SQL:2003 OLAP standards applicable to data streams. The most distinctive feature of the Stream Mill system is its support for native UDAs that allow advanced continuous queries on relational streams. These functionalities can now be used on flattened SAX-3 streams.

To achieve the full integration of relational and XML streams and their query languages, the Stream Mill system also supports XQuery so that users can express queries on streaming XML documents by using standard XQuery. However, XQuery currently fails to provide good support for analytics, data mining, and many other functions that are now available in SQL:2003, or in ESL using our extended UDAs [11]. Therefore, we enable XQuery to call ESL UDAs and allow it to take full advantage of these powerful facilities.

Given an XQuery statement, Stream Mill first decomposes it into several XPaths; these XPaths are added to the parallel XPath processor, which is a UDA similar to that of Example 17 and generate the content in global FSA tables. The results of these XPaths queries are joined to determine the results of the original XQuery. Furthermore, specialized external functions and UDAs can be called from XQuery, and the compiler supports dynamic incorporation of such functions, and the insertion or deletion of queries that are processed in parallel by Yfilter UDAs.

7.1 Performance Study

ESL is the application language for the Stream Mill system which provides full support for ESL queries via functions such as compilation, optimization, query scheduling, load balancing, in-memory tables, hash-based indexes, R-

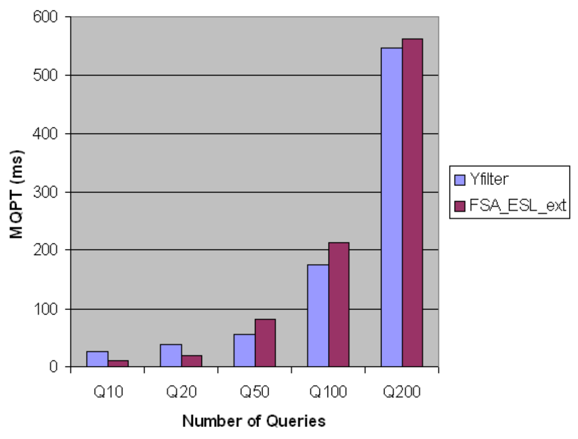

Figure 6. Scalability Test Results

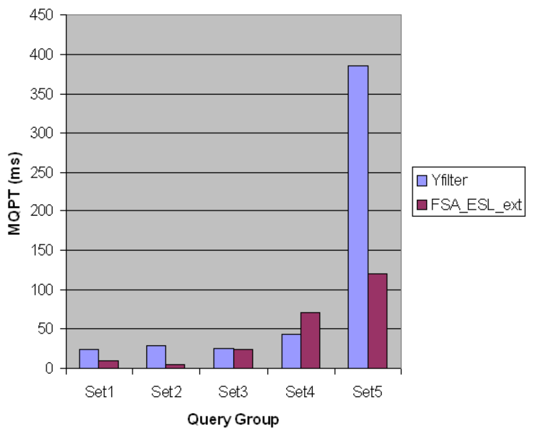

Figure 7. Effect of Different Types of Queries

tree based indexes, and performance monitors [12].

Three implementation approaches were explored for the parallel processing of XPath statements. The first option is to use ESL to define UDAs that simulate the Yfilter FSA. As a second alternative, the same UDAs can be written in programming languages such as C/C++, to achieve better performance. Thirdly, system-defined aggregates (SDAs) can be built to support the Yfilter FSA and achieve optimal performance and scalability. In our experiments, we measured the performance obtained with the second approach, i.e., UDAs written in an external PL, and compared it to that of the YFilter demo system [3]. In terms of performance, this a middle-road solution that achieves better performance and scalability than ESL-coded UDAs, but it not as good as that expected from SDAs. The results of our experiments are reported in Figures 6 and 7, where FSA_ESL_ext denotes the C++ coded UDAs.

All of the experiments reported here were performed on a P4 2.4GHz processor with 1GB memory running JVM 1.4.1 on Linux Red Hat 8.0 machine. As discussed in [28], we use multi query processing time (MQPT), which includes the filtering time but not the document parsing time, as the measure of the efficiency. We perform different experiments to test the scalability and performance for different types of queries. Our experiments suggest that, on the average, the performance of our C++ coded UDAs is comparable to that of the original YFilter, although it can be better or worse for a particular types of queries.

Scalability Test In this test we check the performance of the systems for increasing number of queries. Figure 6 shows filtering times for increasing number of queries: 10, 20, 50, 100, and 200. The filtering times for YFilter and FSA_ESL_ext increase at almost similar rate. This shows that both systems are equally scalable.

Effect of Different Types of Queries Next, we check the effect of different types of queries. In this experiment, we take five different groups of queries, each group consisting of a set of 50 queries as follows:

Set1: simple queries that do not contain a wild-card characters or a '//' descendant axis

Set2: queries with a 0.2 probability of wild-cards

Set3: queries with a 0.2 probability of '//' descendant axes

Set4: queries with a 0.15 probability of wild-card and a 0.5 probability of '//' descendant axes,

Set5: queries with a 0.4 probability of having wild-cards combined with '//' descendant axes.

Figure 7 shows the MQPTs for the respective implementations. FSA_ESL_ext has high MQPT for set4, but YFilter has much higher MQPT for set5. The experiments suggest that, on the average, the UDA-based implementation in the Stream Mill system is comparable to the specialized implementation of YFilter as far as filtering times and scalability are concerned.

We have also implemented our FSA simulator of YFilter as SDAs using C++ and integrated them in Stream Mill. Although this work is still in progress and various optimization improvements remain to be added, our preliminary results show that the performance of our SDAs is always better than that of YFilter implemented in Java—from 2 times to 3.5 times faster, depending on different query groups. This difference is what one would normally expect for respective implementations in Java and C++. In conclusion, building an implementation of XML and XQuery in Stream Mill produces performance results comparable to those obtainable by building a complete new system from scratch, while greatly reducing the effort spent in design and development.

8 Conclusion

In this paper, we have presented the approach used by Stream Mill [12] to unify the processing of XML streams and relational data streams into one DSMS. Although very desirable from an application viewpoint, such a unification has not been achieved by other DSMSs. Indeed, while it is straightforward to represent SAX events as relational streams, supporting complex queries (e.g., those expressed using XQuery) on such streams represents a difficult research problem.

With an innovative approach, we have shown that this problem can be solved by exploiting UDAs, rather than joins as suggested by previous authors. The experience with Stream Mill shows that ESL UDAs can actually support FSA-based XQuery/XPath processing of XML streams with performance that is comparable to that of dedicated implementations of YFilter.

Relational data streams and XML data streams can now be easily transformed from one to the other; moreover, different query languages, such as SQL and XQuery, can cooperate in the same application, by processing streams and producing answers in either format. Furthermore, we provided simple mechanisms whereby complex aggregates with windows can be defined in ESL and imported into XQuery. These constructs have proved extremely useful for relational data streams, but they are not supported in XQuery, nor can they be added easily—e.g., the native function-definition mechanisms now provided by XQuery are blocking. Streaming XML applications can now benefit from these powerful ESL constructs and from their efficient implementation provided by Stream Mill [12].

References

[1] SQL 2003 Standard Support in Oracle Database 10g. World Wide Web, otn.oracle.com/products/database/application_development/pdf/SQL_2003_TWP.pdf.

[2] SQL/XML. World Wide Web, http://www.sqlx.org.

[3] YFilter 1.0 Release. World Wide Web, http://yfilter.cs.berkeley.edu/code_release.htm.

[4] A. Arasu, S. Babu, and J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. World Wide Web, http://dbpubs.stanford.edu/pub/2002-57, 2002.

[5] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Framework for Logical XQuery Optimization. In *VLDB*, pages 168–179, 2004.

[6] A. Eisenberg, J. Melton, K. Kulkarni, et al. SQL:2003 Has Been Published. In *SIGMOD Record*, volume 33, pages 119–126, 2004.

[7] A. K. Gupta and D. Suciu. Streaming Processing of XPath Queries with Predicates. In *SIGMOD*, pages 419–430, 2003.

[8] C. Barton, P. Charles, D. Goyal, et al. Streaming XPath Processing with Forward and Backward Axes. In *ICDE*, pages 455–466, 2003.

[9] C. Chan, P. Felber, et al. Efficient Filtering of XML Documents with XPath Expressions. In *ICDE*, pages 235–244, 2002.

[10] C. Koch, S. Scherzinger, N. Schweikardt, and et al. Schema-based Schedulng of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *VLDB*, pages 228–239, 2004.

[11] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A Native Extension of SQL for Mining Data Streams. In *SIGMOD Demo*, pages 873–875, 2005.

[12] C. Zaniolo, R. Luo, H. Wang, et al. Stream Mill: Bringing Power and Generality to Data Stream Management Systems. World Wide Web, http://wis.cs.ucla.edu/stream-mill/index.html.

[13] D. Carney, U. Cetintemel, M. Cherniack, et al. Monitoring Streams - a New Class of Data Management Applications. In *VLDB*, pages 215–226, 2002.

[14] D. Florescu, C. Hillery, D. Kossmann, et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB*, pages 997–1008, 2003.

[15] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD*, pages 431–442, 2003.

[16] H. Wang and C. Zaniolo. ATLaS: a Native Extension of SQL for Data Minining. In *SDM*, pages 130–141, 2003.

[17] K. S. Beyer, D. Chamberlin, L. Colby, et al. Extending XQuery for Analytics. In *SIGMOD*, 2005.

[18] K. S. Beyer, R.J. Cochrane, L.S. Colby, et al. XQuery for Analytics: Challenges and Requirements. In *XIME-P*, pages 3–8, 2004.

[19] L. Golab and M. T. Ozsu. Issues in Data Stream Management. In *SIGMOD Record*, volume 32, pages 5–14, 2003.

[20] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. In *TKDE*, volume 11, pages 610–628, 1999.

[21] N. Bruno, L. Gravano, et al. Navigation- vs. Index-Based XML Multi-Query Processing. In *ICDE*, pages 139–150, 2003.

[22] N. Koudas and D. Srivastava. Data Stream Query Processing: A Tutorial. In *VLDB*, page 1149, 2003.

[23] R. Bourret. XML and Databases. World Wide Web, http://www.saxproject.org.

[24] R. R. Bordawekar, C. A. Lang. Analytical Processing of XML Documents: Opportunities and Challenges. In *SIGMOD Record, Vol 34. No. 2*, 2005.

[25] S. Kepser. A Proof of the Turing-Completeness of XSLT and XQuery. In *Technical report SFB 441, Eberhard Karls Universitat Tubingen*, 2002.

[26] S. Madden, M. A. Shah, J. M. Hellerstein, et al. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD*, pages 49–60, 2002.

[27] T. Fiebig and G. Moekotte. Algebraic XML Construction in Natix. In *WISE*, pages 212–221, 2001.

[28] Y. Diao, M. Altinel, M. Franklin, et al. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. In *TODS*, pages 467–516, 2003.

[29] Y. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *VLDB*, pages 492–503, 2004.

[30] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML Query Engine for Network-Bound Data. In *VLDB Journal*, volume 11, pages 380–402, 2002.