Extending a General-Purpose Streaming System for XML

Mark Mendell, Howard Nasgaard
IBM Canada
{mendell,nasgaard}@ca.ibm.com

Eric Bouillet
IBM Ireland
bouillet@ie.ibm.com

Martin Hirzel, Buğra Gedik
IBM USA
{hirzel,bgedik}@us.ibm.com

ABSTRACT

General-purpose streaming systems support diverse application domains with powerful and user-defined stream operators. Most general-purpose streaming systems have their own, non-XML, internal data representation. However, streaming input is often either a sequence of small XML documents, or a scan of a huge document. Prior work on XML streaming focuses on filtering, not transforming, XML, and does not describe how to integrate with a general-purpose streaming system. This paper describes how to integrate an XML transformer with a streaming system by designing a specification syntax that is both consistent with the existing system and familiar to XML users. After type-checking the specification, we compile it to an efficient automaton driven by SAX events. Our approach extends the underlying streaming system with XML support without changing its core architecture, and the same technique could be used for other extensions beyond XML.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability—*Data mapping*; H.2.3 [**Database Management**]: Languages—*Data manipulation languages*

General Terms

Languages

1. INTRODUCTION

Stream processing systems [1, 2, 3, 8] deliver high-throughput and low-latency processing to applications in diverse domains such as telecommunications, transportation, finance, and health-care. Since many applications require domain-specific processing, general streaming systems support powerful user-defined operators. At the same time, many standardized data exchange formats are now based on XML, and streaming XML enables two important use cases: processing a stream of many small XML documents, or processing a huge XML document in a single scan. Our goal is to extend a general-purpose streaming system with XML support, to get the best of both worlds in terms of efficiency, generality, and ease-of-use.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2012, March 26–30, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-0790-1/12/03 ...$10.00

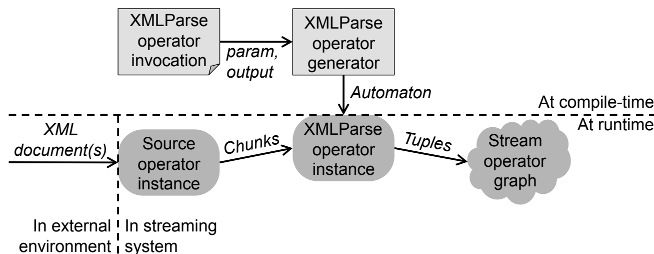

Figure 1: Overview of our approach.

The body of work on streaming XML contains several optimizations [4, 9, 10, 17], but is not general enough for our setting, because most of it focuses on *filtering* XML, as opposed to *transforming* it to a format amenable to a general-purpose streaming system. Work on extending general databases for XML processing focuses on a relational representation with index lookups, but only supports *batch* processing, as opposed to *streaming* XML [5, 7].

Our approach is to add a new operator, `XMLParse`, to a general-purpose streaming system. We use InfoSphere Streams[1] as the underlying system, which comes with its own language SPL[2] [13]. In SPL, streaming operators are implemented by user-defined code generators. Figure 1 illustrates our approach. In the program text, each invocation of the `XMLParse` operator declaratively specifies a transformation from XML documents to SPL tuples. By careful operator design, this specification has a look-and-feel that is familiar to both XML and SPL users. At compile-time, the operator invocation is translated into an automaton. At runtime, the automaton is encapsulated in an operator instance, which reacts to SAX parser events [6] to do efficient data extraction and transformation. Our approach separates the data `Source` from `XMLParse` as separate operator instances, yielding two benefits: one is generality (we uniformly support many kinds of sources, such as reading from files, databases, sockets, etc.), and the other is the ability to handle either many small XML documents or a single giant XML document. Furthermore, `XMLParse` is separated from the downstream operator graph, which consists of instances of library operators or user-defined operators, supporting diverse application domains. `XMLParse` is available on Streams

[1] http://www.ibm.com/software/data/infosphere/streams/
[2] SPL is short for Streams Processing Language.

Exchange, a website for sharing InfoSphere Streams code[3].

We argue that the design from Figure 1 addresses our requirements. First, it is efficient, because we translate the operator invocation into an automaton at compile-time. Second, it is general, because both the source operators and the downstream operators use an industry-strength, general-purpose streaming system. And third, it is easy to use, because we integrate existing syntax from both XML-languages and SPL to obtain a declarative specification. We increase ease-of-use further by also providing intuitive defaults for the data transformation, so that users only need to specify those parts of a transformation explicitly that differ from the common-case defaults.

This paper makes the following primary contributions:

- A declarative extension of a general-purpose streaming system for XML processing.
- An automaton-based approach for efficient transformation (not just filtering) of XML into native formats.

Our approach combines a declarative operator invocation with a user-defined code generator to obtain an efficient operator instance. The bigger picture of this work is that it can serve as a blue-print for extending streaming systems not just with XML support, but also with other additions.

2. BACKGROUND ON SPL

SPL [13, 14] is the programming language for the general-purpose stream processing system InfoSphere Streams. An InfoSphere Streams application is a stream graph, consisting of operator instances (vertices) and streams (directed edges). Streams transport conceptually infinite sequences of discrete data items, which we call *tuples*. Each operator instance is push-driven: each time a tuple arrives on an input stream to an operator instance, it triggers a *firing*. When an operator fires, it consumes the triggering tuple from its input stream, executes some local code, and produces zero or more output tuples on output streams [19]. This data-flow driven model of computation makes it easy to exploit task and pipeline parallelism, and indeed, InfoSphere Streams usually runs on clusters of several commodity workstations.

Kind	Example type	Example literal
Bool	**boolean**	**true**
Number	**int32**	42
String	**rstring**	"answer"
Tuple	**tuple**<**int32** *q*, **rstring** *a*>	{*q*=42, *a*="?"}
List	**list**<**float64**>	[1.618, 3.141]
Map	**map**<**rstring, int32**>	{"phi": 2, "pi": 3}

Figure 2: A few SPL types.

SPL is strongly typed and statically typed. Figure 2 lists some representative types in SPL. There are several primitive types (**boolean**, **int32**, **rstring**, etc.), as well as several composite types (**tuple**, **list**, **map**, etc.). SPL provides a literal syntax for all types, including composite types: list literals are written in [...], tuple literals are written in {...} with field[4] assignments of the form *ID* = *expr*, and map literals are written in {...} with mappings of the form

[3]http://www.ibm.com/developerworks/wikis/display/streams
[4]The SPL specification refers to fields as attributes, but here we call them fields to avoid confusion with XML attributes.

```
1 stream<rstring sym, float64 price> AvgPrice
2 = Aggregate(Quote) {
3   window Quote       : sliding, count(8), partitioned;
4   param  partitionBy : sym;
5   output AvgPrice    : sym = Last(sym),
6                        price = Average(price);
7 }
```

(a) *Aggregate* operator invocation (SPL).

```
8 void process(Tuple& tIn, int port) {
9   Partition p = window.getPartition(<%=getKey()%>);
10  Tuple tOld = p.pop_front();
11  <% genStateRemovalCode(tOld) %>
12  p.push_back(tIn);
13  <% genStateAdditionCode(tIn) %>
14  Tuple tOut;
15  <% genOutputAssignmentCode(tOut) %>
16  submit(tOut, 0);
17 }
```

(b) Simplified *Aggregate* operator definition (code generator).

```
18 void process(Tuple& tIn, int port) {
19   Partition p = window.getPartition(tIn.sym);
20   Tuple tOld = p.pop_front();
21   total[tIn.sym]-=tOld.total; count[tIn.sym]--;
22   p.push_back(tIn);
23   total[tIn.sym]+=tIn.total; count[tIn.sym]++;
24   Tuple tOut;
25   tOut.sym=tIn.sym;
26   tOut.price=total[tIn.sym]/count[tIn.sym];
27   submit(tOut, 0);
28 }
```

(c) Simplified *Aggregate* operator instance (generated C++).

Figure 3: Operator code generation.

expr : *expr*. The SPL type system is fully nested, meaning that composite types can contain other composite types. This paper describes how to transform data from XML to SPL types. We will use the nested composite types and literals to specify these transformations.

The central concept of SPL is the *operator invocation*. An operator invocation specifies topology (input and output streams) and configures an operator (by providing some or all of the **window**, **param**, **output**, and other clauses). Figure 3(a) shows an example. The topology has one output stream *AvgPrice* and one input stream *Quote*. Line 1 shows the type for tuples on the output stream; Line 3 configures the window for the input stream; Line 4 provides a parameter specifying a partitioning key; and Lines 5-6 configure output assignments for the fields of the output tuple. Note that this operator invocation is declarative: it specifies only *what* the operator should do, but not *how* it is implemented. The output assignments call the following generic operator-specific intrinsic functions:

```
<any T> T Last(T v)              //last value in window
<numeric T> T Average(T v)   //average over window
```

These intrinsics are implemented by code generation.

An *operator definition* for SPL is a code generator, which generates different code depending on the configuration in the operator invocation. Figure 3(b) shows a simplified definition for the *Aggregate* operator. It consists of C++ code with *blanks*, which are fragments of scripting code in **<%...%>** escapes. When the code generator runs, it copies C++ code unchanged, whereas for blanks, it runs the script fragments and replaces them by their output. Figure 3(c) shows the

```
1 stream<rstring d, list<rstring> e> T = XMLParse(X) {
2   param  trigger : "/a/b";
3   output T       : d = XPath("c/d/text()"),
4                    e = XPathList("c/e/text()");
5 }
```

(a) *XMLParse* operator invocation (SPL code).

```
6 <a>
7   <b><c><d>X</d> <e>11</e> <e>12</e></c></b>
8   <b><c><d>Y</d>                     </c></b>
9 </a>
10 <a>
11   <b><c><d>Z</d> <e>31</e>           </c></b>
12 </a>
```

(b) Sample input data (XML documents).

```
13 { d = "X", e = [ "11", "12" ] }
14 { d = "Y", e = [            ] }
15 { d = "Z", e = [ "31"       ] }
```

(c) Sample output data (SPL tuples).

Figure 4: Simple operator invocation.

generated C++ code, where blanks have been filled in based on the concrete operator invocation. This example shows only a highly simplified version of the *Aggregate* operator in the SPL library. The full implementation generates different code for different kinds of windows, provides more operator-specific intrinsics, etc. But the important thing is that this code generation framework is available to developers of all operators, including in the standard library, in third-party libraries, or in special-purpose applications. Invocations for all operators in SPL, including *XMLParse* and *Aggregate*, use the syntax illustrated in Figure 3(a) to specify topology and configure the operator.

3. OPERATOR INVOCATIONS

This section describes the API of the *XMLParse* operator.

3.1 Extracting Flat Tuples

Figure 4(a) shows SPL code that invokes the *XMLParse* operator. Line 1 indicates that the input stream with the raw data is *X*, and the output stream *T* carries SPL tuples with two fields **rstring** *d* and **list**<**rstring**> *e*. Line 2 specifies the *trigger* parameter, which is an absolute XPath indicating that every */a/b* subtree of the XML input should trigger one SPL tuple as output. This trigger means that in the example input data in Figure 4(b), Lines 7, 8, and 11 each trigger one tuple. Going back to the SPL code, Lines 3 and 4 specify how the fields *d* and *e* of the output tuples are assigned. The output assignments use operator-specific intrinsic functions with the following signatures:

```
rstring XPath(rstring subTrig)
list<rstring> XPathList(rstring subTrig)
```

The sub-triggers are relative XPaths, which use the main trigger as the context node. Function *XPath()* extracts a primitive item, and *XPathList()* extracts a list of items. Figure 4(c) shows the resulting output tuples.

We defer the description of how the *XMLParse* operator works internally to Section 4. For now, we focus on what interface it provides to the user. From the example, we can make several observations. First, the code does not require any extensions to SPL; it only uses language features familiar from other SPL operators such as *Aggregate*. Second, the code is strongly typed and statically typed. Third, the code uses familiar syntax for XPath expressions in SPL strings. And fourth, the semantics are intuitive, with absolute paths for triggers that extract tuples, and relative paths for sub-triggers that extract fields. There are a few additional features, which we will discuss below. But first, we clarify what subset of XPath is supported.

```
mainTrigger  ::= absolutePath
               | mainTrigger '|' mainTrigger
subTrigger   ::= suffix
               | prefix '/' suffix
               | subTrigger '|' subTrigger
prefix       ::= . | relativePath
suffix       ::= '@' ID | 'text()'
absolutePath ::= '/' relativePath
relativePath ::= ID | relativePath '/' ID
```

Figure 5: Grammar for the supported XPath subset.

Figure 5 illustrates the syntax allowed for paths in the main trigger and its sub-triggers. Based on the expected use cases, we only support the child axis '/', since doing so simplifies the semantics exposed to the user, and also leads to a more efficient automaton after code generation. Possible future extensions could adapt prior work on XML filtering, such as filter predicates, which would further boost performance by avoiding the generation of unneeded data. A sub-trigger can extract either text or an XML attribute. The or-operator '|' separates alternative triggers. It specifies that either trigger can match, or when used with *XPathList*, that subtrees from both triggers contribute to the SPL list.

3.2 Extracting Nested Tuples

An important advanced use case of the *XMLParse* operator is to transform structured XML documents into nested SPL tuples. This is accomplished by overloading the operator-specific intrinsic functions with generic binary variants:

```
<tuple T> T XPath(rstring subTrig, T spec)
<tuple T> list<T> XPathList(rstring subTrig, T spec)
```

Besides a sub-trigger, these functions use a tuple literal to specify how fields of the sub-tuple are to be extracted. Figure 6 illustrates such an example, where field *c* of the output tuple is itself a tuple with nested fields *d* and *e*. Lines 6-7 show how the nested fields are assigned with another relative sub-trigger. The absolute trigger is the concatenation of the main trigger with the sub-triggers, for example, "/a" / "c" / "d/text()".

As mentioned before, all this is strongly typed. That may seem surprising, considering that the **output** clause holds a *specification* of what to extract, but the compiler checks that it conforms to the type of the *actual data* to be extracted. The reason why this works lies deep in SPL's type system. First, SPL supports tuple literals, and infers their type from their fields. We arrange for the tuple literal in the specification to have the same type as the output tuple. Second, SPL uses structural equivalence, meaning two tuple types are the same if they have the same fields. And third, the generic functions infer their result type from the specification in the second argument. Thanks to the strong static typing, the compiler yields informative error messages for mistakes in specifications of XML data transformation.

```
1 stream<rstring b, tuple<int32 d, int32 e> c> T
2 = XMLParse(X) {
3   param  trigger : "/a";
4   output T :
5     b = XPath("@b"),
6     c = XPath("c", {d = (int32)XPath("d/text()"),
7                     e = (int32)XPath("e/text()")});
8 }
```

(a) *XMLParse* operator invocation (SPL code).

```
9  <a b="X"><c><d>11</d> <e>12</e></c></a>
10 <a b="Y"><c><d>21</d> <e>22</e></c></a>
11 <a b="Z"><c><d>31</d> <e>32</e></c></a>
```

(b) Sample input data (XML documents).

```
12 { b = "X", c = { d = 11, e = 12 } }
13 { b = "Y", c = { d = 21, e = 22 } }
14 { b = "Z", c = { d = 31, e = 32 } }
```

(c) Sample output data (SPL tuples).

Figure 6: Operator invocation with nested tuples.

3.3 Implicit Conversions

So far, we have only seen invocations of the *XMLParse* operator where the user explicitly specified how to convert from XML documents to SPL tuples. But in some cases, users can keep their code more concise by doing certain conversions implicitly, as long as the compiler can infer the conversions from the shape of the output tuple. To this end, we define a default representation for an XML element:

```
tuple< map<rstring, rstring> _attrs,
       rstring _text /*,
       additional fields for nested elements */ >
```

The *_attrs* field represents all the XML attributes as a map from names to values, and the *_text* field represents the XML element's text. Any additional SPL fields are mapped one-to-one to nested XML elements: for example, if the SPL output tuple has a field **tuple**<**rstring** *_text*> *d*, it captures the text of a nested input XML element <*d*>.

```
1 type T_a = tuple<map<rstring, rstring> _attrs,
2                  rstring _text,
3                  rstring d,
4                  list<rstring> e>;
```

(a) *T_a* type definition (SPL code).

```
5 stream<T_a> T = XMLParse(X) {
6   param trigger : "/a";
7         flatten : elements;
8 }
```

(b) *XMLParse* operator invocation (SPL code).

```
9  <a b="vb1" c="vc1">
10   va1
11   <d>vd1</d>
12   <e>ve1a</e><e>ve1b</e></a>
```

(c) Sample input data (XML document).

```
13 { _attrs = { "b": "vb1", "c": "vc1" },
14   _text = "va1",
15   d = "vd1",
16   e = [ "ve1a", "ve1b" ] }
```

(d) Sample output data (SPL tuple).

Figure 7: Operator invocation with inferred output.

Figure 7 illustrates the default conversion scheme in action. The main point to note here is that the type definition in Figure 7(a) contains all the necessary information so that the operator invocation in Figure 7(b) does not need any explicit output assignments. The code generator simply infers implicit output assignments. In general, the user can also mix and match explicit and implicit specifications: any missing explicit specification gets automatically filled in by an implicit specification, assuming the output tuple type has the matching fields. Either way, the specification is declarative, uses familiar XPath syntax, and supports static typing.

4. DATA TRANSFORMATION AUTOMATON

This section describes the code generated for transforming XML documents to SPL tuples. The input to the *XMLParse* code-generator is an operator invocation, which declaratively specifies the data transformation. The output of the code-generator is an operator instance, which implements an automaton. Whenever the SPL compiler encounters an operator invocation, it runs the code generator corresponding to that operator. At launch time, InfoSphere Streams deploys the generated operator instances on a cluster of commodity workstations.

A SAX (Simple API for XML) parser transforms an XML document into a sequence of event handler calls [6]. There are three kinds of events:

- <x>: A *start-tag* event has a payload with the element name x and the XML attributes, if any. The corresponding event handler in *XMLParse* updates the state, initializes data, and records attributes if needed.
- `text()`: A *character-data* event has a payload with characters. The corresponding *XMLParse* event handler records the text if needed.
- </x>: An *end-tag* event has a payload with the element name x. The corresponding event handler in *XMLParse* updates the state, and either assigns or submits buffered data, depending on whether the event corresponds to a main-trigger or a sub-trigger.

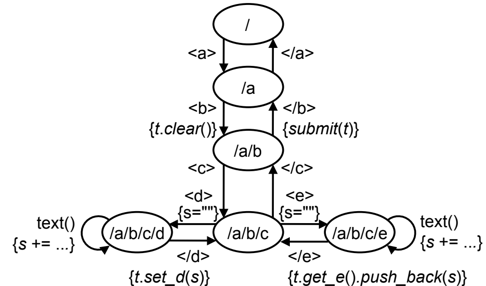

Figure 8: Automaton for invocation from Figure 4.

Figure 8 shows an automaton generated for an *XMLParse* operator invocation. Ovals are states, labeled with an absolute XPath for the subtree they correspond to. Edges are state transitions, labeled with a *guard* and an optional *action*. The guard is a SAX event; for example, guard `</b>` checks whether the event is an end tag for element `b`. The action is a piece of code; for example, action `{submit(t)}` submits tuple `t`. Because Figure 4 Line 2 specifies the main trigger as `"/a/b"`, Figure 8 submits a tuple when it is in state `/a/b` and sees closing tag `</b>`. Because Figure 4 Line 3 assigns `d = XPath("c/d/text()")`, Figure 8

records the text in state /a/b/c/d, and stores it in field *t.d* at closing tag </d>. And because Figure 4 Line 4 assigns *e = XPathList*("c/e/text()"), Figure 8 records the text in state /a/b/c/e, and appends it to field *t.e* at </e>.

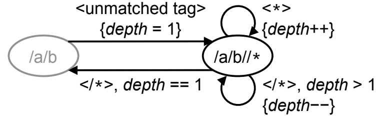

Figure 9: State for skipping unmatched subtree.

The automaton in Figure 8 is simplified: it only shows states and transitions that are relevant for the data transformation, and omits states and transitions that "skip" unmatched XML subtrees. Figure 9 shows such an omitted state /a/b//*. The transition to this state is guarded by unmatched tags, implemented by an **else** clause in the generated code. The state uses an integer *depth*, which it increments on start-tags and decrements on end-tags. The transition back from the skip-state to the main automaton is guarded by </*> for any closing tag and *depth*==1 to check that the unmatched subtree is finished.

In general, whereas the main portion of a *XMLParse* automaton is a DFA (deterministic finite automaton), the portion for skipping unmatched subtrees is not strictly speaking a DFA (because it uses counting and predicates on guards), but still deterministic (because guards are unique and complete). Furthermore, while our automata bear some resemblance to prior work on XML filtering [4, 10, 17], our automata go beyond filtering and also do transformation.

So far, we have seen an example of the generated code, but we have not yet seen the code-generator algorithm. It performs the following steps:

- Make implicit conversions explicit. This step implements the policies from Section 3.3 that infer how to transform XML entities to SPL fields when the user has omitted an explicit specification.
- Separate alternative triggers. This step implements or-operators '|' by expanding them out to all combinations.
- Make sub-triggers absolute. As mentioned in Sections 3.1 and 3.2, a sub-trigger is a relative XPath, whose context node comes from the main trigger or an enclosing sub-trigger. This step concatenates these paths as appropriate to make them all absolute.
- Combine all triggers to a single automaton. This step loops over all trigger paths, and has an inner loop over the levels in each path. At each level, it either reuses an existing automaton state and transition guard or adds new ones if they do not yet exist, and then adds actions.
- Add skipping states and transitions for unmatched subtrees, as illustrated in Figure 9.
- Generate code for the event handlers. This step fills in the three handlers for <*x*>, text(), and </*x*> SAX events. The generated code for tag-handlers consists of an outer **switch** statement with one case for each state, and an inner **if** statement with one else-if clause for each guard. The clauses contain the code for state transitions, data recording, and tuple submission.

Figure 10 shows another example of a generated automaton. This automaton transforms nested tuples, not just flat tuples. It corresponds to the operator invocation in Figure 6,

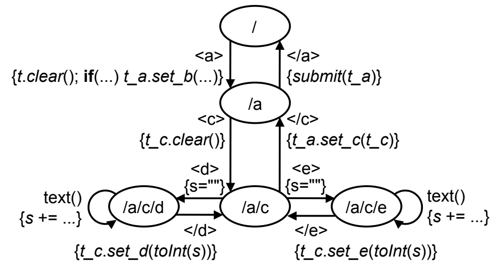

Figure 10: Automaton for invocation from Figure 6.

which has multiple levels of nested sub-triggers. The actions buffer the main tuple in *t_a*, and the nested tuple in *t_c*. The transition from state /a/c at guard </c> simply stores *t_c* in field *t_a.c*. Besides nested tuples, this automaton also illustrates XML attributes. The transition from state / at guard <a> checks whether the opening tag contains attribute @b, and if yes, stores it in a field *t_a.b*.

This section described how to generate efficient automata for *XMLParse* invocations. The *XMLParse* operator is an important extension for our operator library, but it relies only on APIs that are available to all operator developers. It thus demonstrates the versatility of our general-purpose streaming language SPL and its compiler.

5. RESULTS

We evaluate the use cases of (1) continuous processing of many small documents and (2) single-scan processing of one huge document. The case of many small documents is represented by SIRI (Service Interface for Real-Time Information[5]), a protocol for sending public transport vehicle GPS positions and arrival times for passenger information display. We have built a SIRI-based application for the city of Dublin, which has been running non-stop in production for several months. Here, we just measure the performance of the ingest through *XMLParse*. The case of a huge document is represented by XMark [18]. The XMark document describes auctions with people, items, bids, etc. We pick three representative XMark queries that perform no joins and are thus a good fit for streaming. The original intent of XMark is evaluation of mass-storage systems, which typically separate loading from processing; on the other hand, we focus on transformation, hence our measurements include load time.

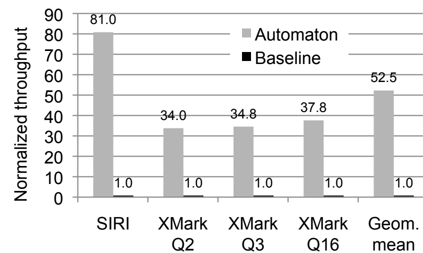

Figure 11: Throughput relative to baseline.

Figure 11 shows the performance of the automaton (the result of code generation in *XMLParse*), compared to the performance of a baseline that uses XPath function calls. For the baseline, we preprocessed the input, so each line contains a document for exactly one main trigger. At runtime,

[5]http://www.kizoom.com/standards/siri/

the baseline parses each line into a DOM tree, and then uses XQilla functions to evaluate the XPaths for the sub-triggers. We conducted all experiments with fused operators. For our experiments, we streamed the data from disk. Even though the baseline has less work to do, since its input is already preprocessed, the automaton performs between 34 and 81 times better, since it uses a SAX parser and processes events in a single scan, whereas the baseline constructs DOM trees and traverses them.

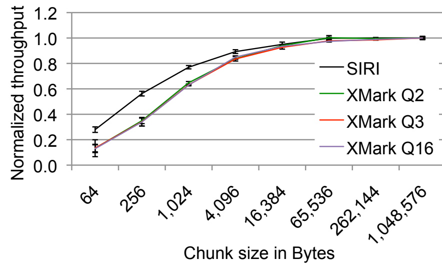

Figure 12: Effect of chunk size on throughput.

As shown in Figure 1, the *Source* operator instance sends the raw input one chunk at a time to the *XMLParse* operator instance. Figure 12 shows the effect of the chunk size on throughput. As chunk size grows, throughput improves, because operator firing costs and SAX parser boundary cases are amortized over more data. The point of diminishing returns is around 16KB, at which point larger chunks have little impact on throughput. We therefore used 16KB chunks for the numbers in Figure 11.

6. RELATED WORK

Our work extends a general-purpose streaming language with a declarative syntax for XML processing, and is hence related to other efforts on extending languages for XML. XDuce is a small general-purpose language designed for XML processing that comes with type system innovations [15]. XJ extends Java with standard-conformant support for both XML schema and XPath [11]. And LINQ extends C# with features that have a similar feel as XPath and XQuery [16]. Unlike our paper, the related work does not focus on streaming languages. Furthermore, our approach gets the benefits of static typing and efficient code generation without requiring changes to our core language SPL.

Our work transforms streaming XML into streams of SPL tuples, and is hence related to other efforts on streaming XML processing. NiagaraCQ does many-query filtering, but instead of streams of XML, it focuses on streams of updates to XML files [9]. The $\chi\alpha o\varsigma$ algorithm does filtering for any axes, including backward axes [4]. XSQ uses an automaton to perform not just filtering, but also simple aggregation, but not the kind of full-fledged transformation that our work supports [17]. The XPush machine supports many-query filtering with predicates [10]. Similarly to the related work, we translate XPath expressions to automata for streaming XML processing. But our work focuses on transformation, not just filtering, to make it usable as an extension to a general-purpose streaming system. MDQ transforms XML for data integration, but works on trees, instead of driving an automaton via SAX events [12].

7. CONCLUSIONS

This paper presents the *XMLParse* operator, which bridges the gap from XML inputs to general-purpose streaming systems. The operator is easy to use, because its syntax is familiar, declarative, and strongly typed. The operator is fast, because it uses code generation to synthesize an automaton. Since the implementation does not require changes to the core architecture of the streaming system, it can serve as a sample for other extensions.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.
[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Workshop on Data Mining Standards, Services and Platforms (DM-SSP)*, 2006.
[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 15(2), 2006.
[4] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *International Conference on Data Engineering (ICDE)*, 2003.
[5] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Demo at International Conference on Management of Data (SIGMOD-Demo)*, 2006.
[6] D. Brownell. *SAX2*. O'Reilly, 2002.
[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *International Conference on Management of Data (SIGMOD)*, 2002.
[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.
[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *International Conference on Management of Data (SIGMOD)*, 2000.
[10] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *International Conference on Management of Data (SIGMOD)*, 2003.
[11] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in Java. In *International World Wide Web Conferences (WWW)*, 2005.
[12] M. Hentschel, L. Haas, and R. Miller. Just-in-time data integration in action. In *Demo at Very Large Data Bases (VLDB-Demo)*, 2010.
[13] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. SPL Streams Processing Language Specification. Technical Report RC24897, IBM Research, 2009.
[14] M. Hirzel and B. Gedik. Streams that compose using macros that oblige. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2012.
[15] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *International World Wide Web Conferences (WWW)*, 2000.
[16] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling objects, relations, and XML in the .NET framework. In *Industrial Sessions at the International Conference on Management of Data (SIGMOD)*, 2006.
[17] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *International Conference on Management of Data (SIGMOD)*, 2003.
[18] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Very Large Data Bases (VLDB)*, 2002.
[19] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, 2010.