Extending a General-Purpose Streaming System for XML

Mark Mendell, Howard Nasgaard
IBM Canada

Eric Bouillet
IBM Ireland

Martin Hirzel, Bugra Gedik
IBM USA

EDBT 2012
General-Purpose Streaming System

Phone calls
Heart beats
GPS location
Stock quotes
...

Continuous input

Stream graph (in SPL)
IBM InfoSphere Streams
Cluster of commodity workstations

Continuous output

Alerts
Actions
Logs
...

• Long-running applications
• Aggregate, enrich, filter, join, classify, …
• High-throughput, low-latency
• Library of reusable stream operators
• Inherent parallelism

Streams Processing Language
Stream Graphs in SPL

Input data

Op_A {
  ...
}

Op_B {
  ...
}

Op_C {
  ...
}

Op_D {
  ...
}

Op_E {
  ...
}

Op_B {
  ...
}

S1 -> S2
S3 -> S4
S5 -> S6
S7 -> S8

Operator (stream transformer)

Stream (infinite sequence of tuples)

Operator configuration (param, output, etc.)

Data in tuples
(e.g., \{ x=1, y=["b", "c"] \})

Input often in XML

In external environment
In streaming system

Input data
XML Support as an Operator

Source operator instance

XML document(s)

Chunks

XMLParse operator instance

Tuples

Stream operator graph

In external environment

In streaming system
Code Generation

XMLParse operator invocation

XMLParse operator generator

Source operator instance

XMLParse operator instance

Chunks

Tuples

Stream operator graph

At compile-time

At runtime

In external environment

In streaming system

XML document(s)

param, output

Automaton
Declarative Operator Configuration

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

At compile-time

Automaton

Source operator instance

Chunks

XMLParse operator instance

Tuples

Stream operator graph

{ d = "X", e = [ "11", "12" ] }

{ d = "Y", e = [ ] }

{ d = "Z", e = [ "31" ] }

At runtime
Background: SPL Compiler

SPL source:
- Stream graph (operator invocations)
- Declarative operator configuration
- Consistent syntax across all (library or user-defined) operators

Operator definitions are code generators

Generated C++ code
From SPL Source to Automaton

stream<rstring d, list<rstring> e> T = XMLParse(X) {
  param trigger : "/a/b";
  output T : d = XPath("c/d/text()"),
              e = XPathList("c/e/text()");
}
Observations on the Automaton

• Memory efficient:
  Avoids in-memory tree representation for XML
• Comprehensive:
  Filtering + data extraction + transformation
• Incremental:
  Each SAX event triggers a constant-time action
Skipping Unmatched Subtrees

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

At compile-time

```
/
/a
/a/b
   
   <unmatched tag>
   {depth = 1}
/a/b/
   
   <//*>, depth == 1
   
   <//*>, depth > 1
   
   {depth--}
/a/b//
   
   <*/>, depth == 1
/a/b/c/d
/a/b/c
/a/b/c/e
```

At runtime

```
 Automaton
```

At compile-time
Nested Tuples

```cpp
stream<rstring b, tuple<int32 d, int32 e> c> T = XMLParse(X) {
    param trigger : "/a";
    output T :
        b = XPath("@b"),
        c = XPath("c", {d = (int32)XPath("d/text()"),
                        e = (int32)XPath("e/text()"))});
}
```

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

XMLParse operator instance

```
{ b = "X", c = { d = 11, e = 12 } }
{ b = "Y", c = { d = 21, e = 22 } }
{ b = "Z", c = { d = 31, e = 32 } }
```
Automaton for Nested Tuples

```cpp
stream<rstring b, tuple<int32 d, int32 e> c> T = XMLParse(X) {
    param trigger : "/a";
    output T :
    b = XPath("@b"),
    c = XPath("c"),
    d = (int32)XPath("d/text()"),
    e = (int32)XPath("e/text()"));
}
```

SAX events

SPL tuples
Background: SPL Type System

- Strongly typed
- Statically typed
- Nested types and literals, e.g.:

\[
\text{list<map<rstring, tuple<int32 x, int32 y>>>}
\]

\[
l_1 = [ \{ "k1": \{ x=1, y=2 \} \} ];
\]
Explicit Conversions

```plaintext
type T_a = tuple<map<rstring, rstring> _attrs,
               rstring _text,
               rstring d,
               list<rstring> e>;
stream<T_a> T = XMLParse(X) {
    param trigger : "/a";
    flatten : elements;
}

<a b="vbl" c="vcl">
  val
  <d>vd1</d>
  <e>vela</e><e>velb</e></a>

{ _attrs = { "b": "vbl", "c": "vcl" },
  _text = "val",
  d = "vd1",
  e = [ "vela", "velb" ] }
Experimental Methodology

• Baseline: in SPL but without our operator
  – Preprocess data to one input line per main trigger
    for each input line:
      parse XML into DOM tree
    for each sub-trigger:
      extract data from tree with XPath function

• SIRI: Many small XML documents
  – Location updates for public transportation

• XMark: One huge XML document
  – Synthetic auction information
  – Picked queries without joins

• All measurements include load time
Throughput vs. Baseline

- **Automaton (SAX, operator)**
- **Baseline (DOM, functions)**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Normalized Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIRI</td>
<td>1.0</td>
</tr>
<tr>
<td>XMark Q2</td>
<td>34.0</td>
</tr>
<tr>
<td>XMark Q3</td>
<td>34.8</td>
</tr>
<tr>
<td>XMark Q16</td>
<td>37.8</td>
</tr>
<tr>
<td>Geom. mean</td>
<td>52.5</td>
</tr>
</tbody>
</table>
Effect of Chunk Size on Throughput

XML document(s) → Source operator instance → Chunks → XMLParse operator instance → Tuples

Graph showing the normalized throughput as a function of chunk size in bytes for different tests (SIRI, XMark Q2, XMark Q3, XMark Q16).
Conclusions

• Use an automaton not just for XML filtering, but also for transformation
• For efficiency, use code generation
• In SPL, users can write their own operators as code generators
• To learn more about SPL: http://publib.boulder.ibm.com/infocenter/streams/v2r0/