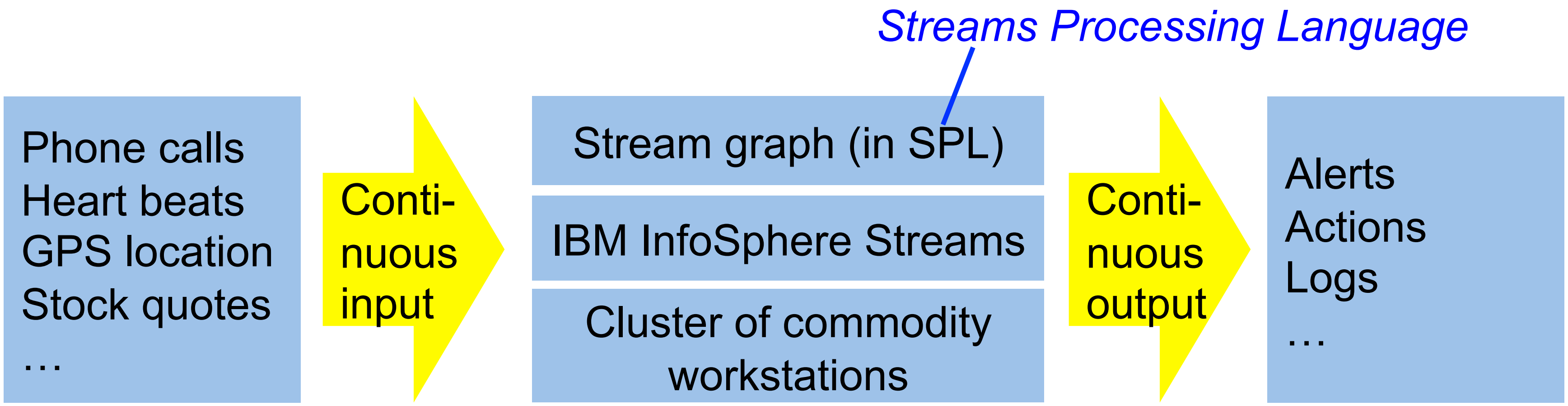
Extending a General-Purpose Streaming System for XML

Mark Mendell, Howard Nasgaard Eric Bouillet Martin Hirzel, Bugra Gedik

IBM Canada IBM Ireland IBM USA

EDBT 2012

General-Purpose Streaming System



Streams Processing Language

Phone calls
Heart beats
GPS location
Stock quotes
…

Conti-nuous input

Stream graph (in SPL)

IBM InfoSphere Streams

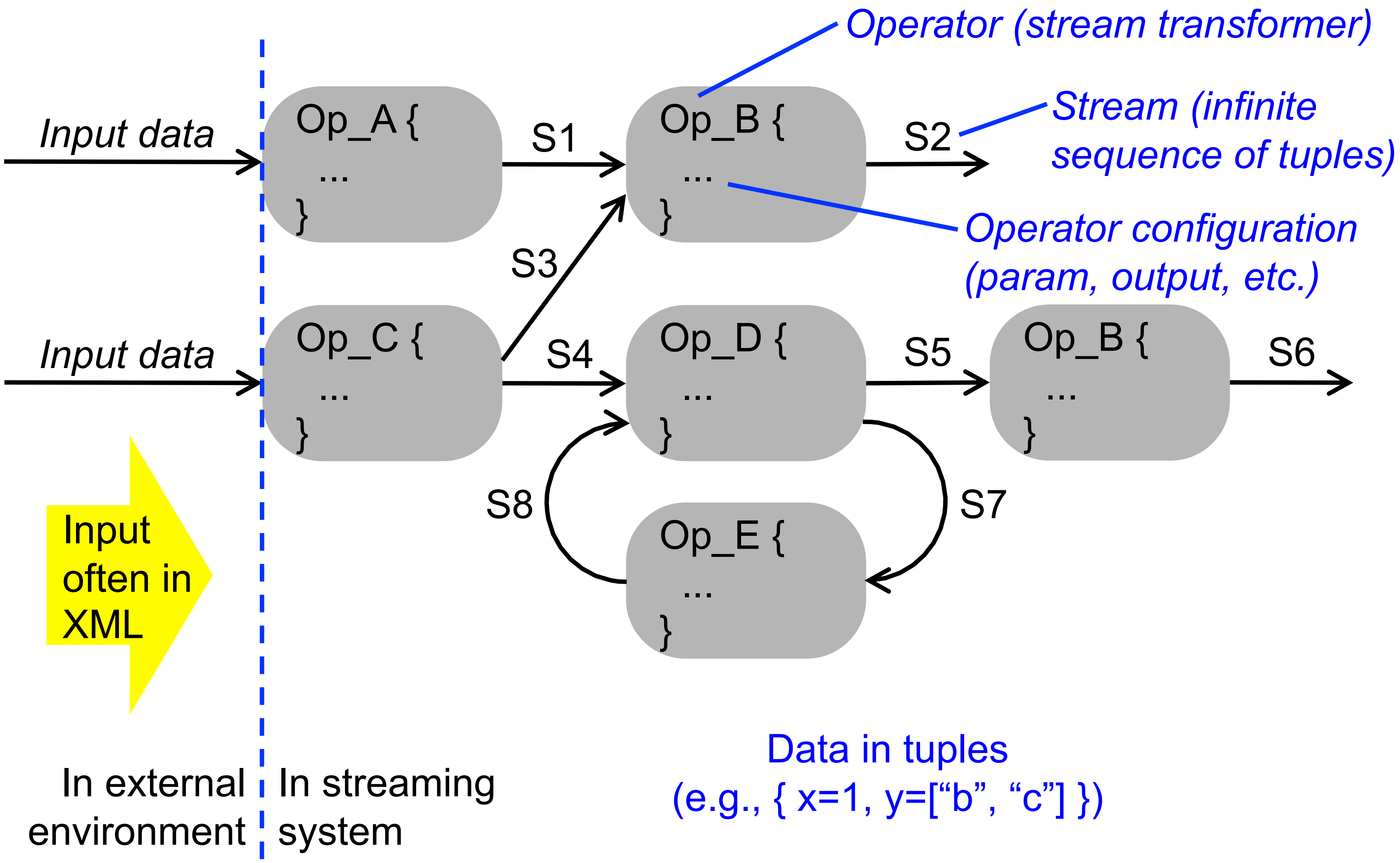
Cluster of commodity workstations

Conti-nuous output

Alerts
Actions
Logs
…

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

Stream Graphs in SPL



Operator (stream transformer)

Stream (infinite sequence of tuples)

Operator configuration (param, output, etc.)

Input data

Input data

Op_A {
 …
}

Op_B {
 …
}

Op_C {
 …
}

Op_D {
 …
}

Op_B {
 …
}

Op_E {
 …
}

S1

S2

S3

S4

S5

S6

S7

S8

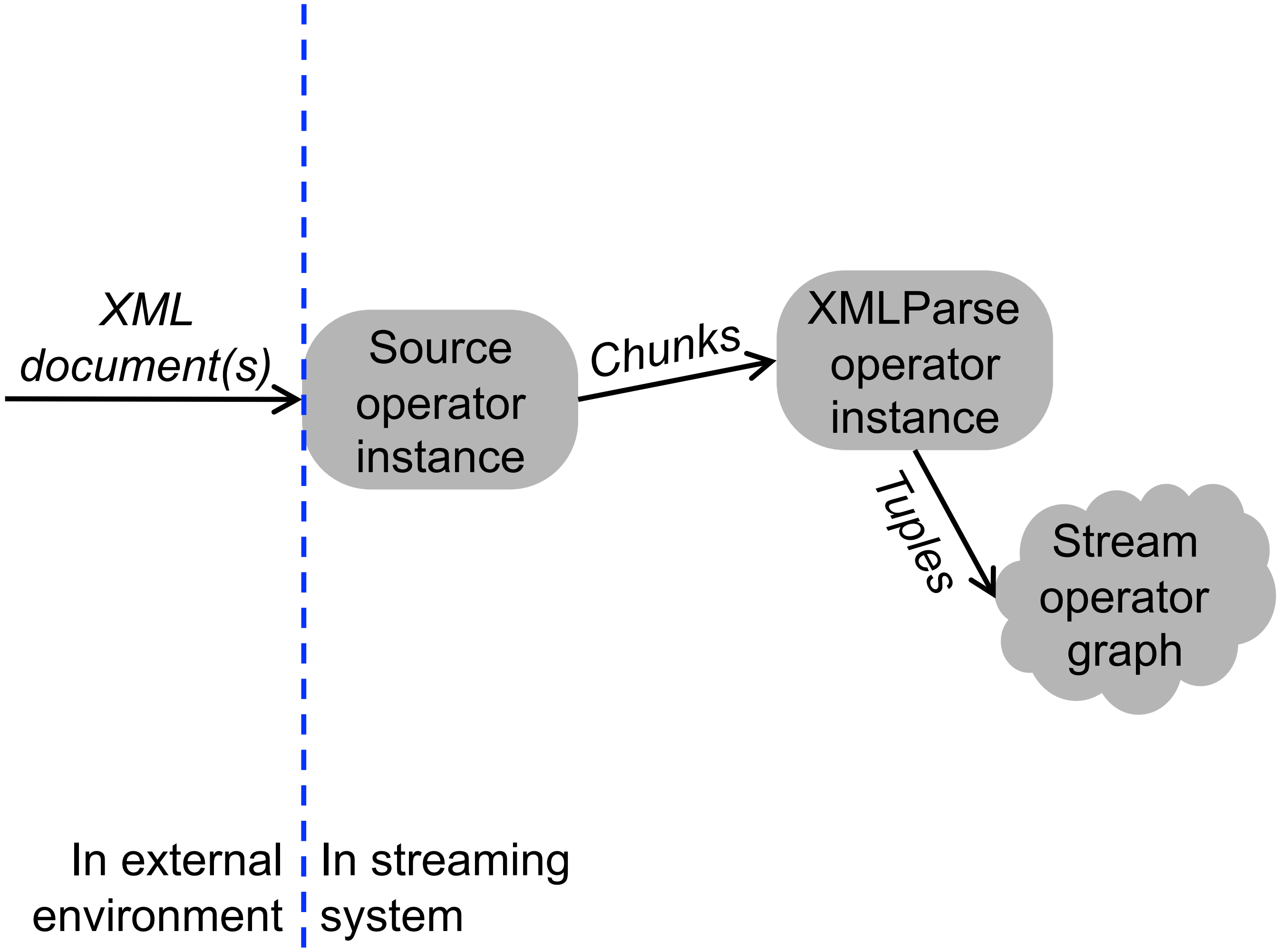
Input often in XML

In external environment

In streaming system

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

XML Support as an Operator



XML document(s)

Source operator instance

Chunks

XMLParse operator instance

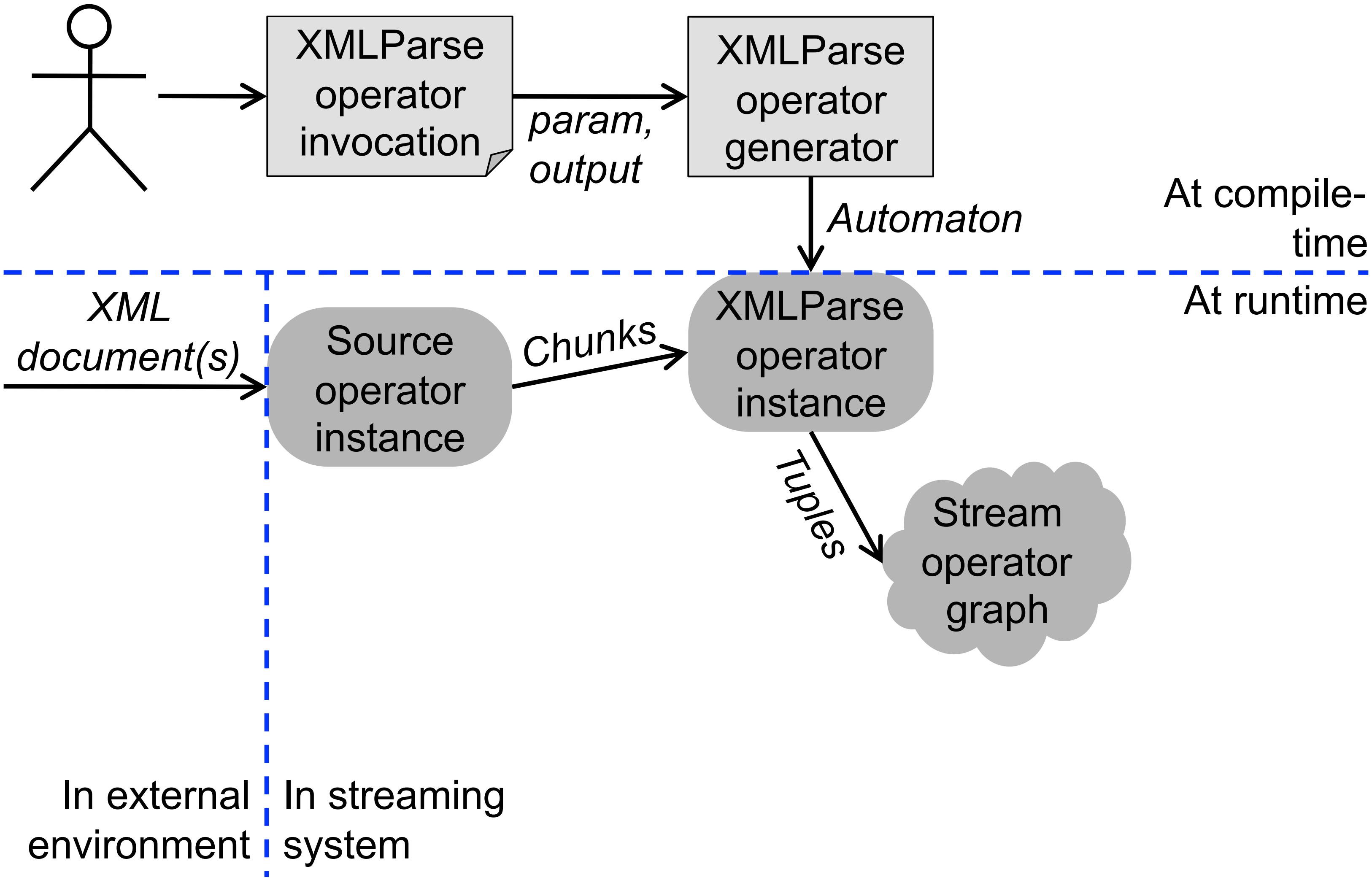
Tuples

Stream operator graph

In external environment

In streaming system

Code Generation



XMLParse operator invocation

param, output

XMLParse operator generator

Automaton

At compile-time

At runtime

XML document(s)

Source operator instance

Chunks

XMLParse operator instance

Tuples

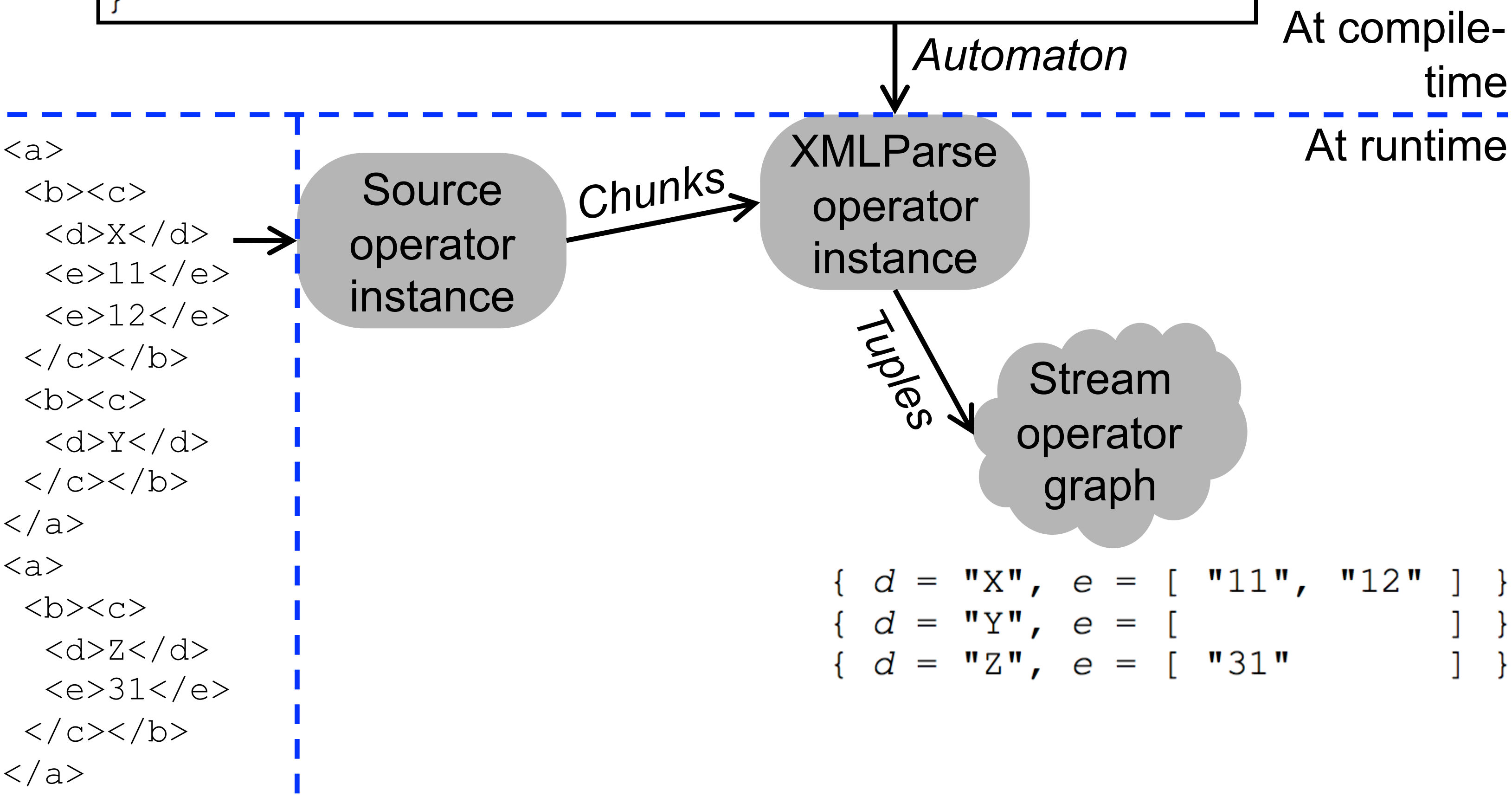
Stream operator graph

In external environment

In streaming system

Declarative Operator Configuration

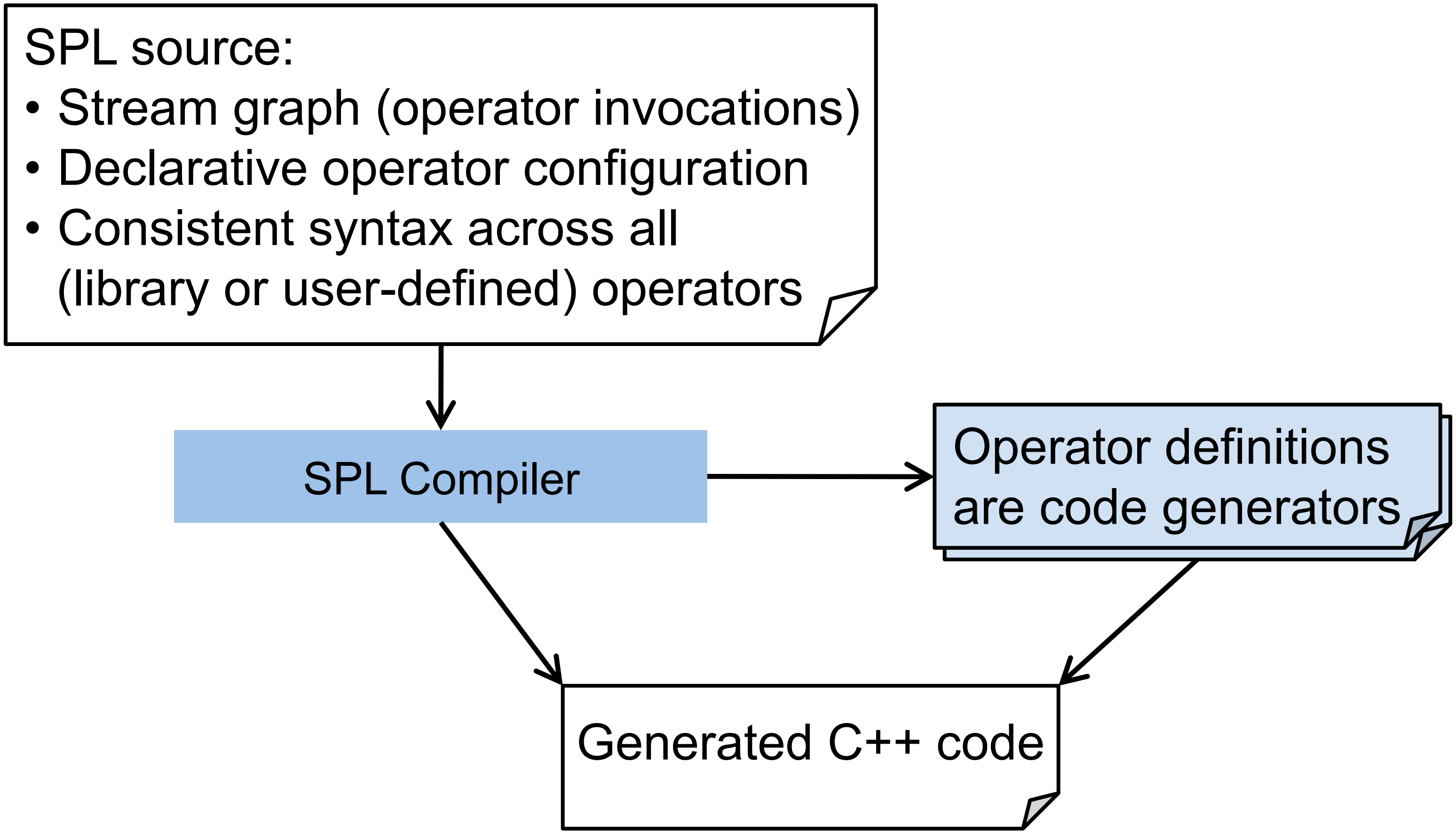
```
stream<rstring d, list<rstring> e> T = XMLParse(X) {
  param  trigger : "/a/b";
  output T       : d = XPath("c/d/text()"),
                   e = XPathList("c/e/text()");
}
```



```
<a>
 <b><c>
  <d>X</d>
  <e>11</e>
  <e>12</e>
 </c></b>
 <b><c>
  <d>Y</d>
 </c></b>
</a>
<a>
 <b><c>
  <d>Z</d>
  <e>31</e>
 </c></b>
</a>
```

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

Background: SPL Compiler



SPL source:

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

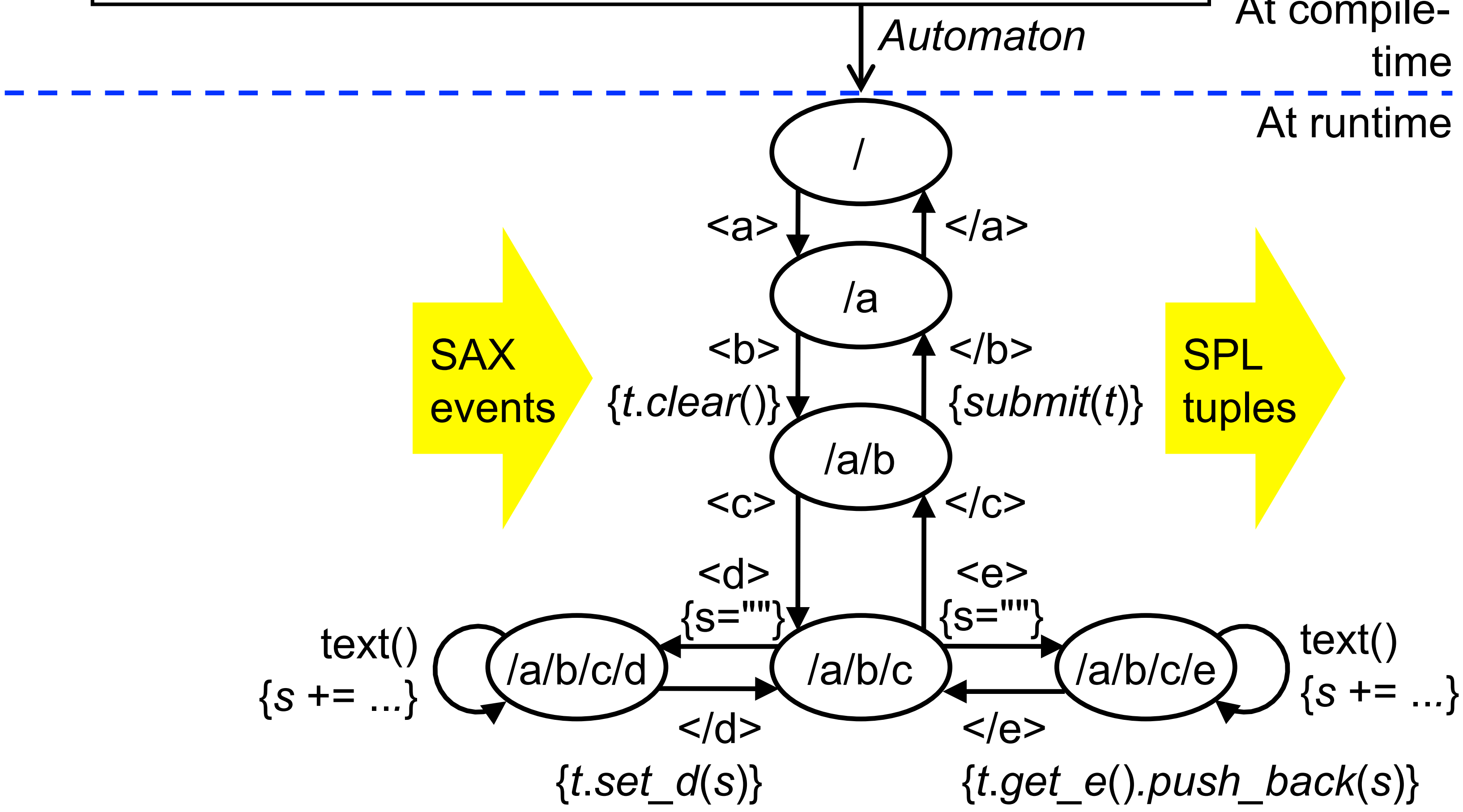
SPL Compiler

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()");
}
```

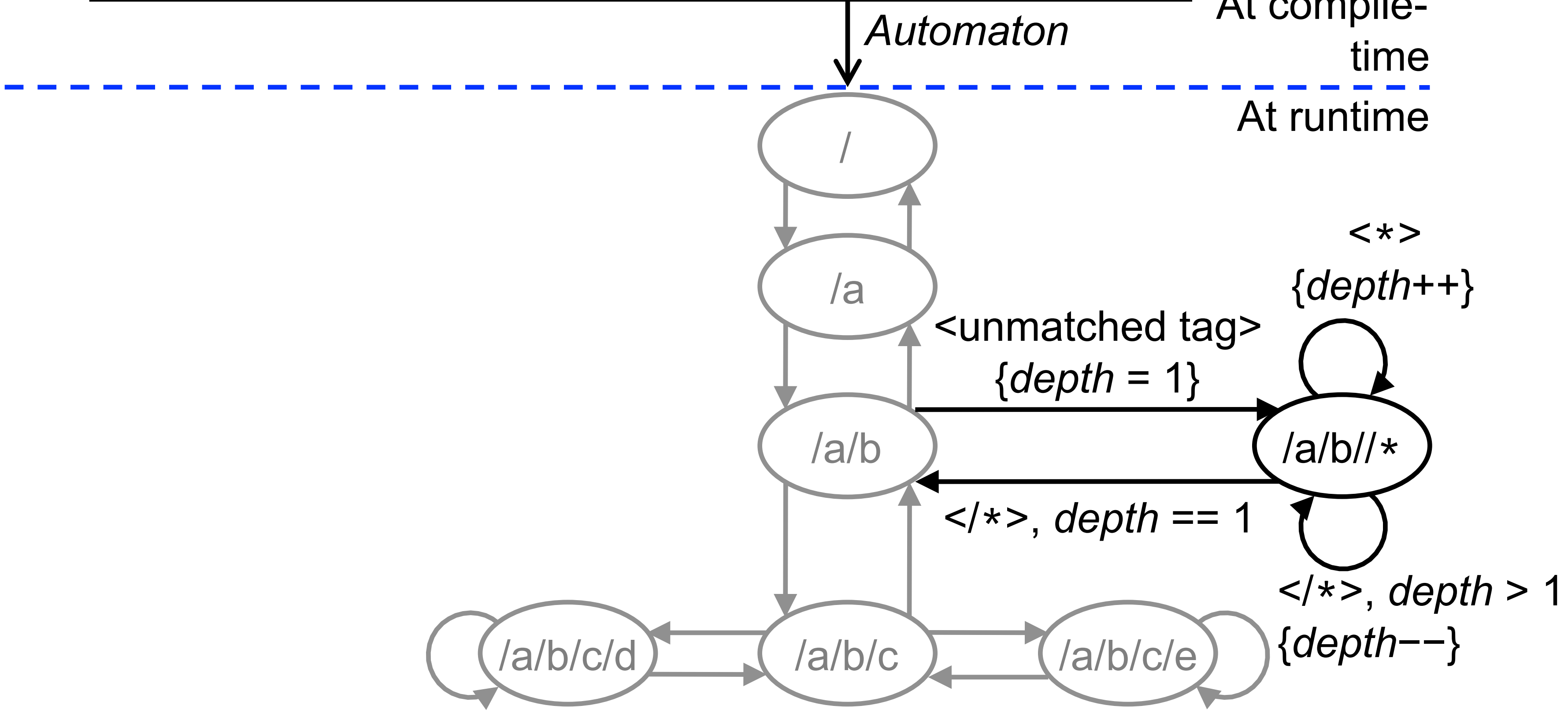


Observations on the Automaton

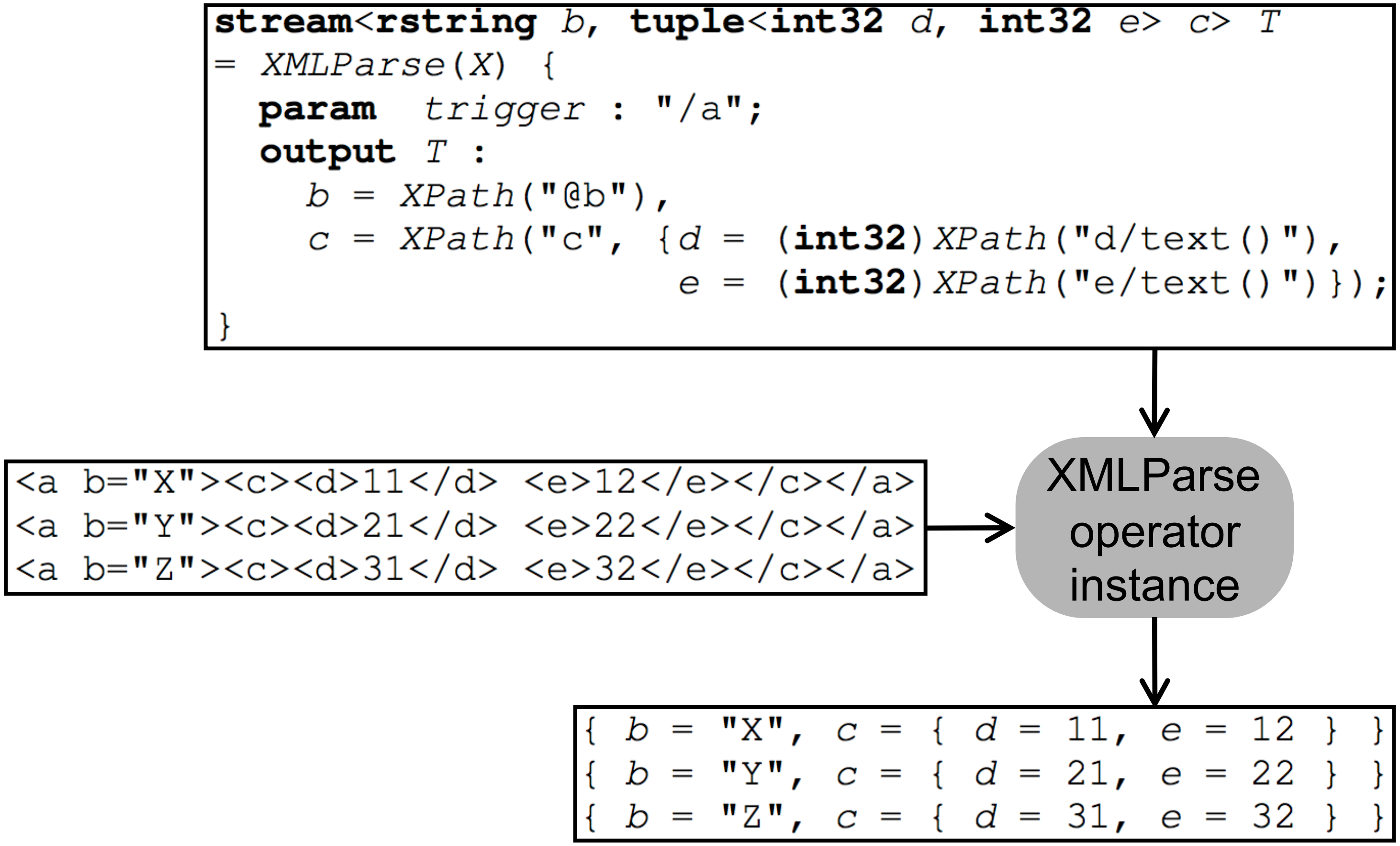
- Memory efficient:
  Avoids in-memory tree representation for XML
- Comprehensive:
  Filtering + data extraction + transformation
- Incremental:
  Each SAX event triggers a constant-time action

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()");
}
```



Nested Tuples



```
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()")});
}
```

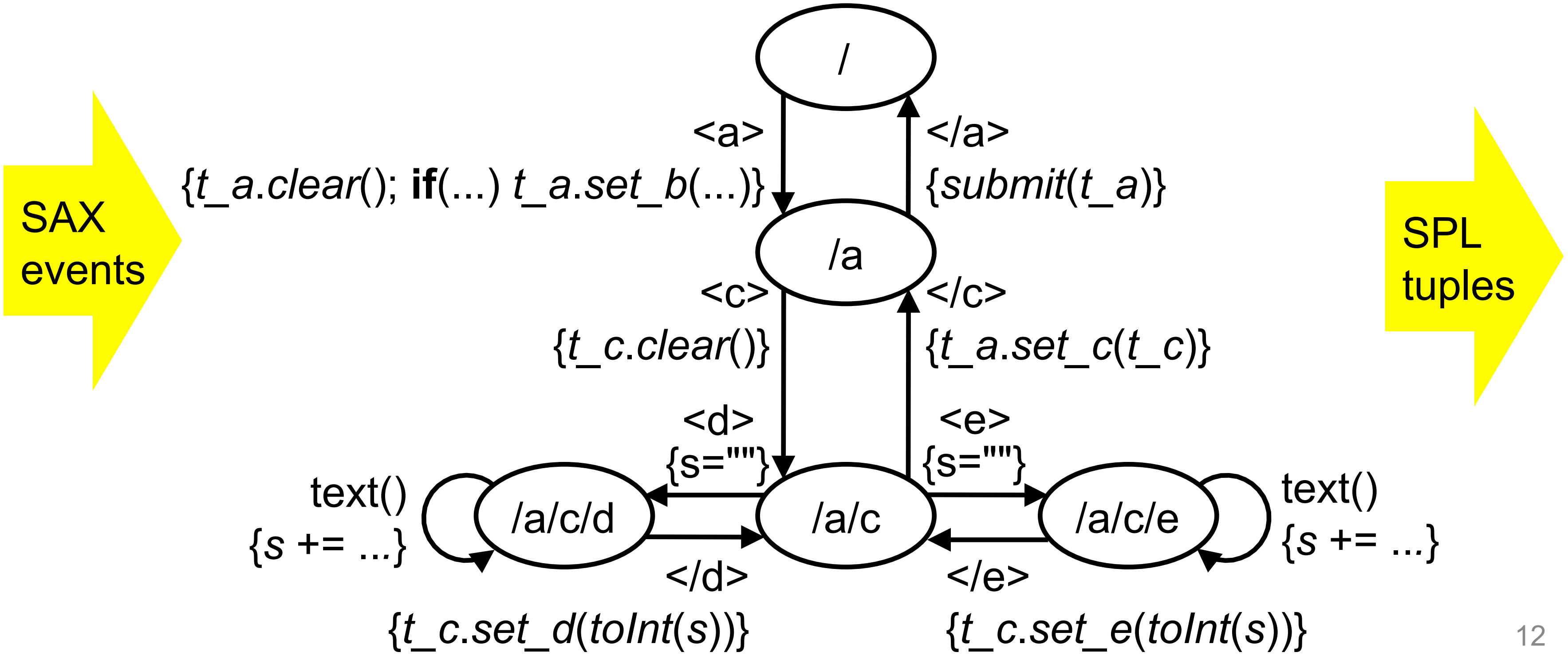
```
<a b="X"><c><d>11</d> <e>12</e></c></a>
<a b="Y"><c><d>21</d> <e>22</e></c></a>
<a b="Z"><c><d>31</d> <e>32</e></c></a>
```

XMLParse operator instance

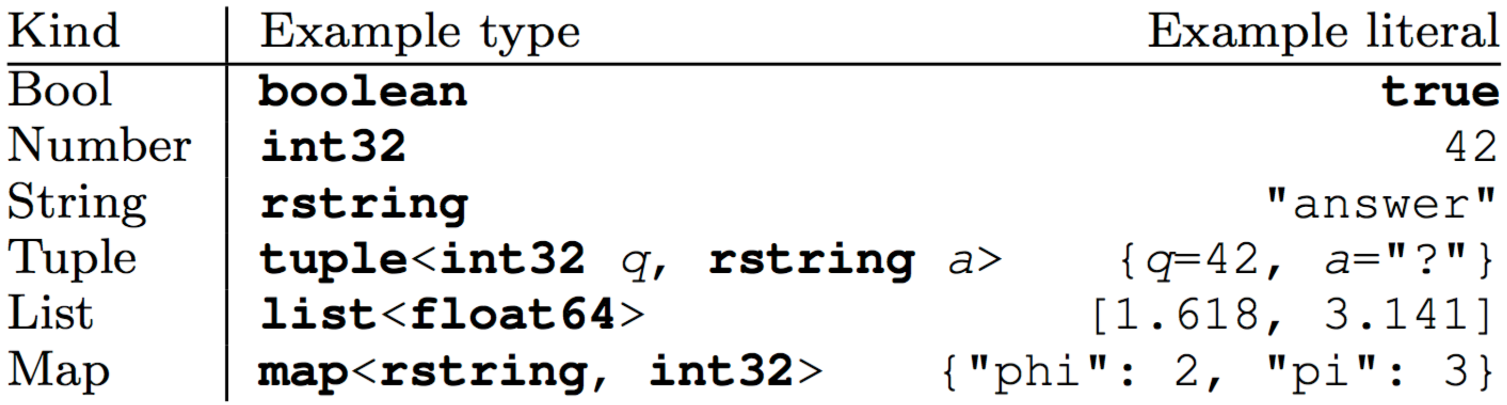
```
{ b = "X", c = { d = 11, e = 12 } }
{ b = "Y", c = { d = 21, e = 22 } }
{ b = "Z", c = { d = 31, e = 32 } }
```

Automaton for Nested Tuples

```
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()")});
}
```

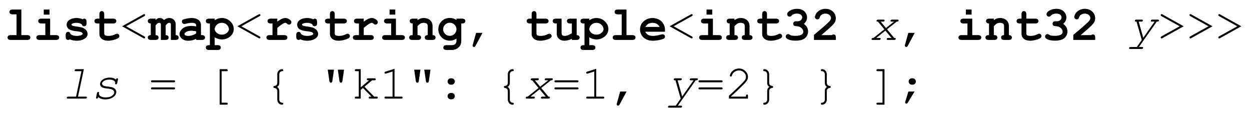


Background: SPL Type System



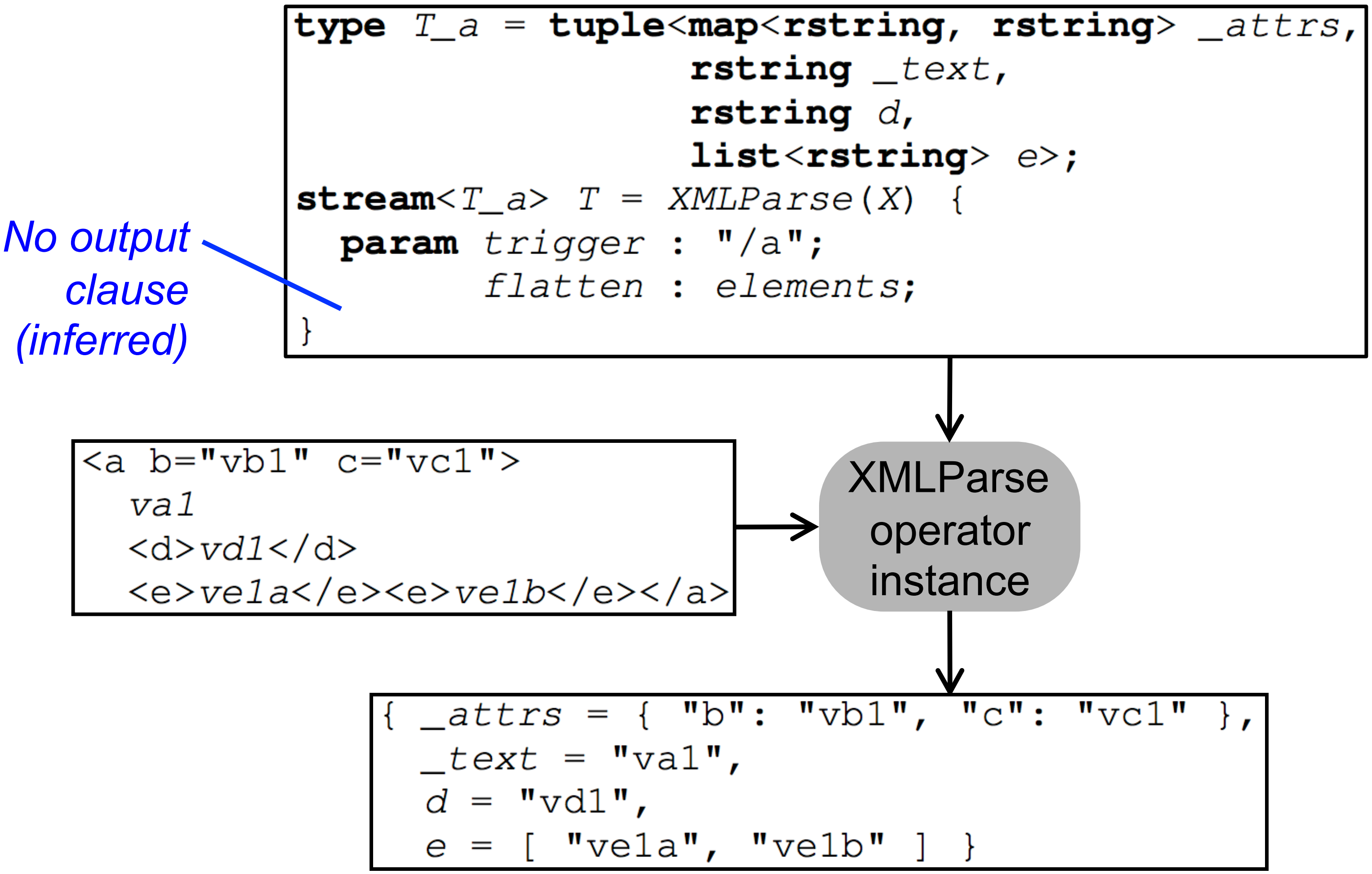
Kind	Example type	Example literal
Bool	**boolean**	**true**
Number	**int32**	42
String	**rstring**	"answer"
Tuple	**tuple**<**int32** *q*, **rstring** *a*>	{*q*=42, *a*="?"}
List	**list**<**float64**>	[1.618, 3.141]
Map	**map**<**rstring**, **int32**>	{"phi": 2, "pi": 3}

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



```
list<map<rstring, tuple<int32 x, int32 y>>>
  ls = [ { "k1": {x=1, y=2} } ];
```

Implicit Conversions



```
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;
}
```

No output clause (inferred)

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

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

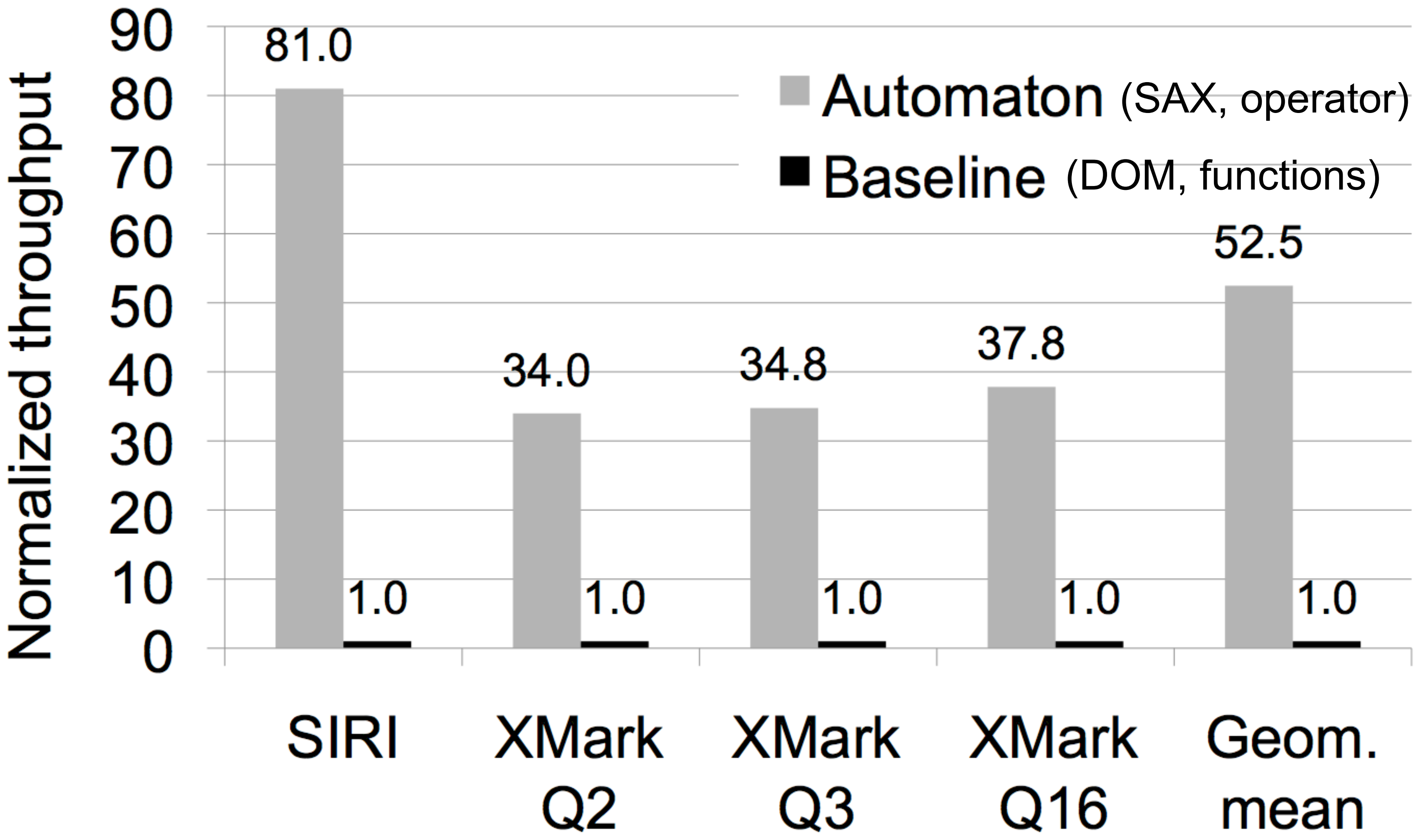
Experimental Methodology

- Baseline: in SPL but without our operator
  - Preprocess data to one input line per main trigger

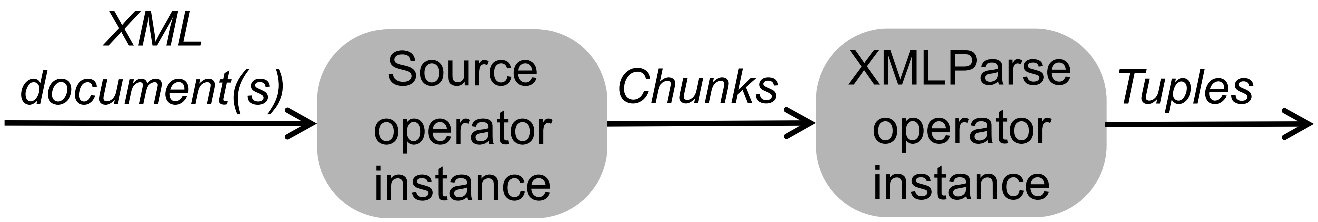
```
for each input line:
    parse XML into DOM tree
    for each sub-trigger:
        extract data from tree with XPath function
```

- 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



Effect of Chunk Size on Throughput



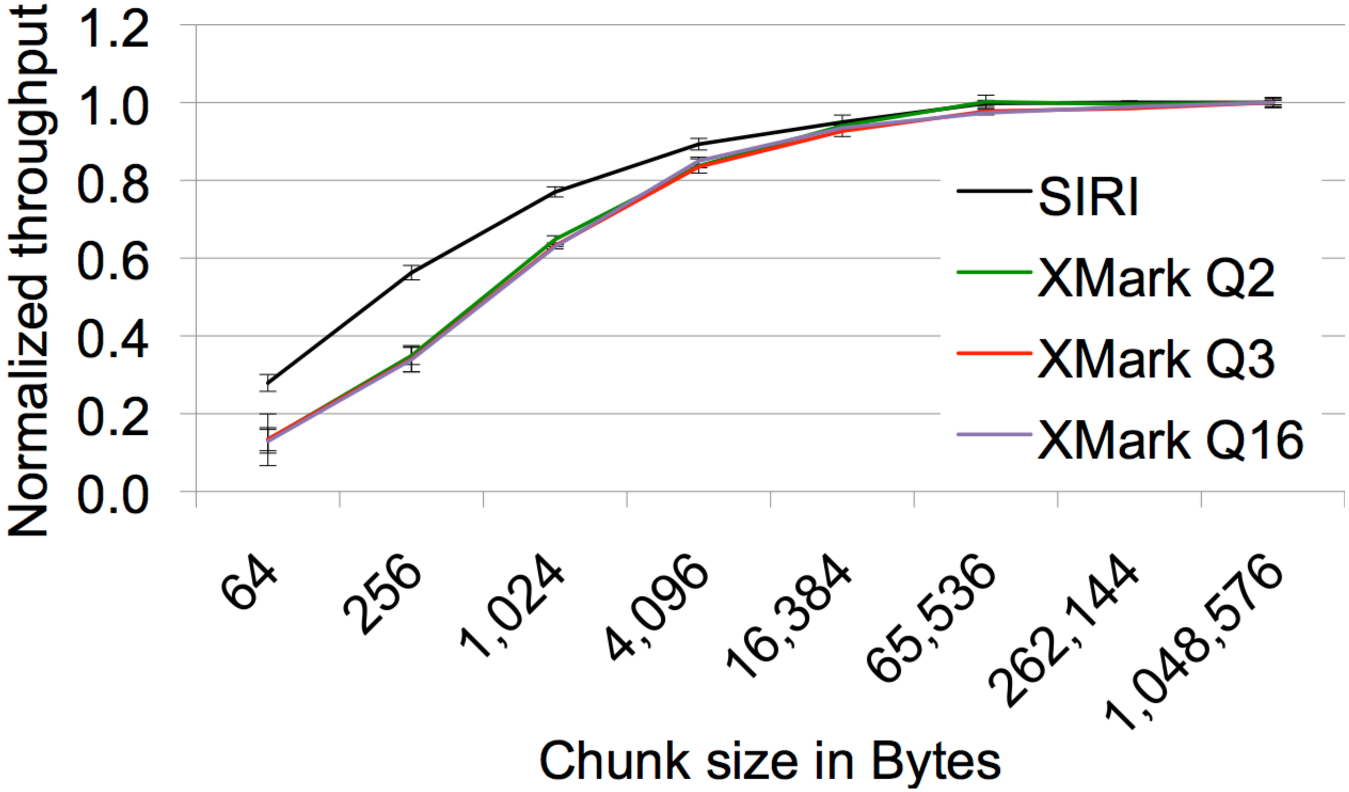
XML document(s)

Source operator instance

Chunks

XMLParse operator instance

Tuples



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/