Partition and Compose: Parallel Complex Event Processing

Martin Hirzel, IBM Research
Tuesday, 17 July 2012
DEBS
CEP ?= Stream Processing?

Event (Stream) Processing

Aggregate
Enrich
Filter
Join
Parse

Complex Event Processing

Use pattern over “simple events” to detect and report “composite events”

→ CEP as an operator in a streaming language?
Background: SPL

- IBM Streams Processing Language
- SPL is the language for InfoSphere Streams (IBM Product)
- This paper is based on System S = research branch of InfoSphere Streams
Scenario: Financial analysis

M-shape (double-top) stock pattern

Series of rising peaks and troughs

Deep drop below start of match

Source: http://www.cs.cornell.edu/bigreddata/cayuga/
M-Shape pattern in SPL

```java
1 stream<MatchT> Matches = MatchRegex(Quotes) {
2   param
3     pattern : ".* rise+ drop+ rise+ drop* deep";
4     partitionBy : symbol;
5   predicates : {
6     rise = price>First(price) && price>=Last(price),
7     drop = price>=First(price) && price<Last(price),
8     deep = price<First(price) && price<Last(price) }
9 output
10    Matches : symbol=symbol, seqNum=First(seqNum),
11       count=Count(), maxPrice=Max(price);
12 }
```

→ Operator only, no extensions to SPL syntax
## Regular expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>(explanation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Identifier</td>
<td>(predicate)</td>
</tr>
<tr>
<td>.</td>
<td>Wildcard</td>
<td>(true predicate)</td>
</tr>
<tr>
<td>$re_1$ $re_2$</td>
<td>Concatenation</td>
<td>($re_1$ followed by $re_2$)</td>
</tr>
<tr>
<td>$re_1$</td>
<td>$re_2$</td>
<td>Disjunction</td>
</tr>
<tr>
<td>$re^*$</td>
<td>Kleene star</td>
<td>(zero or more repetitions)</td>
</tr>
<tr>
<td>$re+$</td>
<td>Kleene plus</td>
<td>(one or more repetitions)</td>
</tr>
<tr>
<td>$re?$</td>
<td>Optional</td>
<td>(zero or one occurrences)</td>
</tr>
<tr>
<td>$(re)$</td>
<td>Grouping</td>
<td>(overrides operator precedence)</td>
</tr>
<tr>
<td></td>
<td>Empty</td>
<td>(consumes no events)</td>
</tr>
</tbody>
</table>

> Pattern language familiar from string matching
## Aggregations

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int32 Count()</code></td>
<td>Number of simple events</td>
</tr>
<tr>
<td><code>&lt;any T&gt; T First(T v)</code></td>
<td>First in match</td>
</tr>
<tr>
<td><code>&lt;any T&gt; T Last(T v)</code></td>
<td>Last in match</td>
</tr>
<tr>
<td><code>&lt;ordered T&gt; T Max(T v)</code></td>
<td>Largest</td>
</tr>
<tr>
<td><code>&lt;ordered T&gt; T Min(T v)</code></td>
<td>Smallest</td>
</tr>
<tr>
<td><code>&lt;numeric T&gt; T Sum(T v)</code></td>
<td>Sum</td>
</tr>
<tr>
<td><code>&lt;numeric T&gt; T Average(T v)</code></td>
<td>Arithmetic mean</td>
</tr>
<tr>
<td><code>float64 Delta(timestamp v)</code></td>
<td>Time since match start</td>
</tr>
<tr>
<td><code>&lt;any T&gt; list&lt;T&gt; Collect(T v)</code></td>
<td>All values as a list</td>
</tr>
</tbody>
</table>

→ **Operator-specific intrinsic functions**
Matching semantics

• **Standard regular expression semantics**
• Non-greedy *(right-minimal)*
• Partition-isolated
• *(Partition-)*Contiguous
• Non-overlapping
  *(submit longest: left-maximal)*
Implementation overview

- All C++ operators in SPL are code generators

- MatchRegex operator invocation
- MatchRegex operator generator
- Automaton
- At compile-time
- At runtime

Upstream operator instance → MatchRegex operator instance
- Simple events

MatchRegex operator instance → Downstream operator instance
- Composite events
Automaton

\[ \text{. rise}^+ \text{ drop}^+ \text{ rise}^+ \text{ drop}^* \text{ deep} \]

\[ \begin{array}{c}
\text{Update and filter} \\
\text{partial match}
\end{array} \]

\[ \begin{array}{c}
\text{Create new} \\
\text{partial match}
\end{array} \]

\[ \begin{array}{c}
\text{Report completed} \\
\text{match and flush}
\end{array} \]

\[ \Rightarrow \text{NFA (non-deterministic finite automaton)} \]
Partitioning

\[
\text{type} \quad \text{PartitionMap} = \text{map}<\text{Key}, \text{list}<\text{PartialMatch}>>
\]
\[
\text{type} \quad \text{Key} = \text{tuple}<\text{rstring} \; \text{symbol}>
\]
\[
\text{type} \quad \text{PartialMatch} = \text{tuple}<\text{int32} \; \text{state, Aggr} \; \text{aggr}>
\]
\[
\text{type} \quad \text{Aggr} = \text{tuple}<
\begin{align*}
\text{int32} \; & \text{count}, \\
\text{tuple}<\text{uint32} \; & \text{first}, \text{uint32} \; \text{last}, \text{uint32} \; \text{max}> \; \text{price}, \\
\text{tuple}<\text{uint32} \; & \text{first}> \; \text{seqNum}>
\end{align*}
\]

\[
\begin{array}{c}
\text{ts} \\
\text{symbol} \\
\text{price} \\
\text{size} \\
\text{seqNum}
\end{array}
\]

\[
\begin{array}{c}
\text{state} \\
\text{aggr}
\end{array}
\]
```cpp
void process(SimpleEvent& evt) {
    Key key(evt.symbol);
    PartialMatchList& oldPms = 
        partitionMap.has(key) ? *partitionMap.get(key) : *new PartialMatchList();
    PartialMatchList& newPms = *new PartialMatchList();
    partitionMap.put(key, &newPms);
    int longestAccepting = -1;
    /*create new partial matches*/
    if (/*predicate .*/)
        newPms.add(new PartialMatch(1, evt));
    /*update existing partial matches*/
    for (int i=0, n=oldPms.size(); i<n; i++) {
        PartialMatch& pm = *oldPms.get(i);
        switch (pm.state) {
            case 0: {
                if (/*predicate .*/)
                    newPms.add(new PartialMatch(1, evt, pm));
                break;
            }
            /*similar cases for states 1-4*/
            case 5: {
                if (/*predicate drop*/)
                    newPms.add(new PartialMatch(5, evt, pm));
                if (/*predicate deep*/) {
                    newPms.add(new PartialMatch(6, evt, pm));
                    /*update longestAccepting if longer*/
                }
                break; }
        } /*end of switch*/
    delete &oldPms; } /*end of for*/
    /*if any accepting match, submit longest and clear*/
    if (longestAccepting != -1) {
        PartialMatch& pm = *newPms.get(longestAccepting);
        submitEvent(evt.symbol, pm.aggr.seqNum.first,
            pm.aggr.count, pm.aggr.price.max);
        newPms.clear(); }
}
```
Parallelization

→ Schneider et al. [PACT’12]
Safety and determinism

• SPL compiler checks …
  – Syntax and names in expressions
  – Expression and function types
• MatchRegex operator checks …
  – Syntax and names in regular expression pattern
  – Starting predicate aggregation-free
• Auto-parallelizer checks …
  – Partitioning
  – Absence of stateful expressions
  – Sequence numbers and pulses

→ Enables simple output validation with “diff”
## Data sets ...

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Key</th>
<th># Keys</th>
<th># Events</th>
<th>Logical time</th>
</tr>
</thead>
<tbody>
<tr>
<td>finance</td>
<td>Trade</td>
<td>symbol</td>
<td>390</td>
<td>10,000,000</td>
<td>2 h 01 min</td>
</tr>
<tr>
<td>twitter</td>
<td>Tweet</td>
<td>author</td>
<td>6,142</td>
<td>200,000</td>
<td>36 h 40 min</td>
</tr>
</tbody>
</table>

```haskell
type Trade = tuple<
    timestamp ts, rstring symbol,
    uint32 price, uint32 size, uint32 seqNum>;

type Tweet = tuple<
    uint64 id, timestamp ts, rstring author,
    rstring content>;
```

## ... and benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Pattern</th>
<th>Description</th>
<th>Selectivity</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>finance0</td>
<td>largeSize rise+ priceDrop</td>
<td>Large trade followed by peak</td>
<td>1.71 %</td>
<td>MatchRegex only</td>
</tr>
<tr>
<td>finance1</td>
<td>. rise+ drop+ rise+ drop* deep</td>
<td>M-shape (double top)</td>
<td>0.31 %</td>
<td>MatchRegex only</td>
</tr>
<tr>
<td>finance2</td>
<td>. rise* riseEnd flat* flatEnd</td>
<td>Rise then flat with time window</td>
<td>2.72 %</td>
<td>MatchRegex only</td>
</tr>
<tr>
<td>finance3</td>
<td>divergence</td>
<td>Price substantially above VWAP</td>
<td>0.03 %</td>
<td>VWAP → MatchRegex</td>
</tr>
<tr>
<td>finance4</td>
<td>hi gap* lo</td>
<td>Max of hi smaller than min of lo</td>
<td>5.15 %</td>
<td>MinMax → MatchRegex</td>
</tr>
<tr>
<td>finance5</td>
<td>. notTooLong* largeIncrease</td>
<td>Large increase with time window</td>
<td>0.04 %</td>
<td>MatchRegex only</td>
</tr>
<tr>
<td>twitter0</td>
<td>(None)</td>
<td>Parse tweet only, no matching</td>
<td>100.00 %</td>
<td>ParseTweet only</td>
</tr>
<tr>
<td>twitter1</td>
<td>. sameTags+ sameTags5th</td>
<td>Five tweets with identical tags</td>
<td>14.07 %</td>
<td>ParseTweet → MatchRegex</td>
</tr>
<tr>
<td>twitter2</td>
<td>.+ disjointTags</td>
<td>Different first vs. last tags</td>
<td>2.15 %</td>
<td>ParseTweet → MatchRegex</td>
</tr>
</tbody>
</table>
Absolute throughput in events per second

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1 : Non-parallel</th>
<th>Width : Best parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>finance0</td>
<td>1 : 778,584 ± 8.5%</td>
<td>1 : 778,584 ± 8.5%</td>
</tr>
<tr>
<td>finance1</td>
<td>1 : 708,118 ± 16.3%</td>
<td>1 : 708,118 ± 16.3%</td>
</tr>
<tr>
<td>finance2</td>
<td>1 : 832,469 ± 15.4%</td>
<td>1 : 832,469 ± 15.4%</td>
</tr>
<tr>
<td>finance3</td>
<td>1 : 229,090 ± 5.7%</td>
<td>4 : 732,837 ± 15.0%</td>
</tr>
<tr>
<td>finance4</td>
<td>1 : 371,753 ± 9.6%</td>
<td>4 : 670,781 ± 18.5%</td>
</tr>
<tr>
<td>finance5</td>
<td>1 : 235,672 ± 4.2%</td>
<td>8 : 737,656 ± 7.1%</td>
</tr>
<tr>
<td>twitter0</td>
<td>1 : 6,488 ± 8.1%</td>
<td>32 : 127,720 ± 42.6%</td>
</tr>
<tr>
<td>twitter1</td>
<td>1 : 6,267 ± 6.8%</td>
<td>16 : 66,492 ± 22.3%</td>
</tr>
<tr>
<td>twitter2</td>
<td>1 : 316 ± 0.8%</td>
<td>32 : 2,378 ± 2.0%</td>
</tr>
</tbody>
</table>

⇒ Large speedup when low sequential throughput
Speedups

1 Machine x 8 Cores

4 Machines x 8 Cores = 32

Motivates elasticity and auto-width controller
## Related work

<table>
<thead>
<tr>
<th>Engine / language</th>
<th>Complex events</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>NiagaraCQ / XML-QL</td>
<td>Algebraic</td>
<td>No</td>
</tr>
<tr>
<td>SQL-TS</td>
<td>Back-tracking</td>
<td>No</td>
</tr>
<tr>
<td>Amit</td>
<td>Back-tracking</td>
<td>No</td>
</tr>
<tr>
<td>NFA\textsuperscript{b} / SASE</td>
<td>Automaton</td>
<td>No</td>
</tr>
<tr>
<td>MATCH_RECOGNIZE</td>
<td>ANSI proposal</td>
<td>No</td>
</tr>
<tr>
<td>EventScript</td>
<td>Automaton</td>
<td>No</td>
</tr>
<tr>
<td>Cayuga / CEL</td>
<td>Automaton</td>
<td>Yes, by hand</td>
</tr>
<tr>
<td>EventJava</td>
<td>Index data structures</td>
<td>Yes, per task</td>
</tr>
<tr>
<td>[Woods,Teubner VLDB]</td>
<td>Automaton</td>
<td>Yes, on FPGA</td>
</tr>
<tr>
<td>This paper</td>
<td>Automaton</td>
<td>Yes, partitioned</td>
</tr>
</tbody>
</table>
Conclusions

• CEP as an SPL operator
  – Use CEP for pattern matching
  – Use other operators for filtering, enrichment, parsing, joining, etc.

• Up to 830K events/second
  – Incremental aggregation
  – C++ code generation
  – Parallelism (up to 14x speedup)
Backup
Shuffle in twitter02 and twitter03

Source

ParseTweet (replica 0)

ParseTweet (replica 1)

ParseTweet (replica 2)

MatchRegex (replica 0)

MatchRegex (replica 1)

MatchRegex (replica 2)

Down-stream operator

Raw tweets as XML documents

Tweets as simple events

Composite events