# Partition and Compose: Parallel Complex Event Processing

Martin Hirzel
IBM T.J. Watson Research Center
hirzel@us.ibm.com

## ABSTRACT

Complex event processing uses patterns to detect composite events in streams of simple events. Typically, the events are logically partitioned by some key. For instance, the key can be the stock symbol in stock quotes, the author in tweets, the vehicle in transportation, or the patient in health-care. Composite event patterns often become meaningful only after partitioning. For instance, a pattern over stock quotes is typically meaningful over quotes for the same stock symbol. This paper proposes a pattern syntax and translation scheme organized around the notion of partitions. Besides making patterns meaningful, partitioning also benefits performance, since different keys can be processed in parallel. We have implemented partitioned parallel complex event processing as an extension to IBM's System S high-performance streaming platform. Our experiments with several benchmarks from finance and social media demonstrate processing speeds of up to 830,000 events per second, and substantial speedups for expensive patterns parallelized on multi-core machines as well as multi-machine clusters. Partitioning the event stream before detecting composite events makes event processing both more intuitive and parallel.

## Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Data manipulation languages*; H.2.4 [**Database Management**]: Systems—*Query processing*

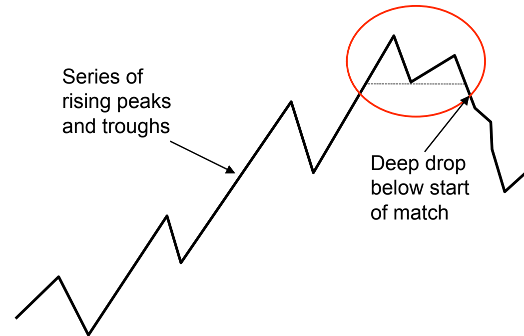
## Keywords

CEP, composite events, stream processing, SPL, pattern matching, regular expressions, automata, parallelism

## 1. INTRODUCTION

The world is becoming more connected, and increasing amounts of continuous data streams are available from domains as diverse as finance, social media, transportation, telecommunications, entertainment, security, and health-care.

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.
*DEBS '12,* July 16–20, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-1315-5 ...$15.00.




(a) Example price curve with match, from [9].

```
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 }
```

(b) SPL source code with `MatchRegex` operator.

**Figure 1: M-shape (double-top) stock pattern, based on Cayuga's financial Subscription 1 [9].**

This has given rise to technologies that process these data streams online for rapid response. One such technology is *stream processing*, which supports many kinds of continuous analytics, such as aggregating, enriching, classifying, annotating, filtering, joining, parsing, etc. of incoming events. Another such technology is *CEP* (complex event processing), which uses patterns over sequences of *simple events* to detect and report *composite events*. The boundaries between stream processing and CEP are not always clear, and the terms are sometimes used interchangeably. The thesis of this paper is that CEP is a special case of stream processing. As evidence for its thesis, this paper describes an implementation of CEP as an operator in a general-purpose stream processing system.

This paper introduces the `MatchRegex` operator, which is an operator for SPL [14], the Streams Processing Language for the System S distributed streaming platform [13]. Figure 1 shows an example of using `MatchRegex` for financial analysis. Like many other SPL operators, `MatchRegex` presents a simple declarative interface to the user, and is in-

ternally implemented via code generation. The declarative interface consists of three parameters: *pattern* is a regular expression over the event stream; *partitionBy* is a key for partitioning the event stream; and *predicates* are boolean expressions for use in the pattern. The partitioning opens the door to *parallelization* on multi-core machines and even multi-machine clusters.

The *MatchRegex* operator has several advantages over prior CEP engines. It performs *incremental aggregation*: it maintains aggregations such as *First*, *Count*, and *Max* from Figure 1 incrementally, storing only the necessary information and updating it one input event at a time. The alternative is keeping a data structure of all matching input events, and computing the aggregation non-incrementally by iterating over these events. The incremental approach is faster and uses less memory. Another advantage is the *integration in a general streaming system*. This makes it possible to use the *MatchRegex* operator for tasks that it is good at, while leaving other tasks, such as XML processing, natural-language text analytics, etc. to other operators designed for those purposes. The operator works with *automatic parallelization*, which yields substantial speedups for slow patterns, without burdening the user with having to parallelize by hand. The operator supports *powerful predicates*, which can even call user-defined functions from the middle of a pattern-match. And finally, the operator uses *standard regular expression* syntax and semantics, making good use of a skill set common among most programmers.

This paper makes the following primary contributions:

- A declarative *syntax* for partitioned CEP patterns that integrates smoothly into the host language (Section 2).
- A translation into *automata* that maintain aggregations incrementally and exploit the partitioning for parallelism (Section 3).
- A suite of static *safety* checks that catch programming errors early and ensure deterministic execution (Section 4).

Overall, this paper demonstrates not only that CEP can be naturally implemented as an operator in a general-purpose streaming system, but it also yields high sequential performance and even higher parallel performance.

## 2. COMPOSITE EVENT SPECIFICATION

The design of the composite-event specification syntax has three objectives: familiarity, expressiveness, and safety. The syntax should be familiar to programmers, so they can pick it up quickly based on what they already know. The syntax should be expressive, so it can easily address common use cases of complex event processing. And the syntax should encourage safe, predictable, and deterministic code, while enabling the compiler to detect and flag errors early. The syntax in this paper resembles the *MATCH_RECOGNIZE* clause proposed to ANSI as an SQL extension [25]. However, we simplified the syntax to make it easier to use and optimize, and we modified it to fit in a library operator without requiring any host-language grammar changes.

### 2.1 Syntax Overview

As the example in Figure 1 shows, a composite event specification consists of a signature (Line 1), a pattern (Line 3), a partition key (Line 4), predicates (Lines 5-8), and output assignments (Lines 10 + 11). The following discussion dissects each of these components.

| Syntax | Description | (explanation) |
|---|---|---|
| *id* | Identifier | (predicate) |
| . | Wildcard | (true predicate) |
| $re_1\ re_2$ | Concatenation | ($re_1$ followed by $re_2$) |
| $re_1$ \| $re_2$ | Disjunction | ($re_1$ or $re_2$) |
| $re$* | Kleene star | (zero or more repetitions) |
| $re$+ | Kleene plus | (one or more repetitions) |
| $re$? | Optional | (zero or one occurrences) |
| ($re$) | Grouping | (overrides operator precedence) |
| | Empty | (consumes no events) |

**Table 1: Regular expressions supported in patterns.**

Line 1 from Figure 1 shows the ***signature***:

```
1 stream<MatchT> Matches = MatchRegex(Quotes) {
```

The signature specifies the type (*MatchT*) and name (*Matches*) of the output stream, the operator (*MatchRegex*), and the input stream (*Quotes*). Conceptually, the *MatchRegex* operator is a stream transformer that transforms an input stream of simple events into an output stream of composite events. Since signatures for all (library or user-defined) SPL operators follow the same syntax, the syntax is familiar to SPL programmers [14]. SPL is a statically typed language, and the type of the output stream improves safety. SPL has an expressive type system, including various primitive types for numbers, strings, enumerations, booleans, and timestamps, as well as composite types such as lists, maps, or tuples. (A *tuple* in SPL is a record of *attributes*, where each attribute has a name and a type. In SPL terminology, the unit of communication on a stream is a tuple, but this paper refers to tuples on streams as *events* instead for consistency with the event-processing literature.)

Line 3 from Figure 1 shows the ***pattern***:

```
3 pattern : ". rise+ drop+ rise+ drop* deep";
```

The pattern is a regular expression over an alphabet of predicates. Regular expressions are familiar to most computer professionals, since most popular programming languages support regular expressions over strings, and most schools teach regular expressions as part of the computer science core curriculum. For ease of use, *MatchRegex* supports the usual regular expression features shown in Table 1. In particular, it supports Kleene closure for high expressiveness.

Line 4 from Figure 1 shows the ***partition key***:

```
4 partitionBy : symbol;
```

In this example, the key is the *symbol* attribute from the input stream of simple events. In general, the key can consist of multiple attributes, separated by commas.

Lines 5-8 from Figure 1 show the ***predicates***:

```
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) };
```

The SPL language supports similar expressions as most C-like languages, shown in Table 2. This syntax is familiar to most programmers. The main novelty here is the use of aggregations (*First* and *Last*). Aggregations look like function calls and are discussed in detail in Section 2.2.

| Syntax | Description | (explanation) |
|---|---|---|
| *id* | Identifier | (attribute of event) |
| *id* ( $e_1, \ldots, e_n$ ) | Call | (intrinsic or SPL function) |
| $e_1$ *op* $e_2$ | Infix operator | (logic, arithmetic, or comparison) |
| *op* *e* | Prefix operator | (logic or arithmetic) |
| $e_1$ [ $e_2$ ] | Subscript | (list or map access) |
| ( *e* ) | Grouping | (overrides operator precedence) |
| *lit* | Literal | (boolean, number, or string value) |

**Table 2: Some of the SPL expressions supported in predicates and output assignments.**

```
 1 boolean tagsEq(rstring tweet1, rstring tweet2) {
 2   list<rstring> xs = parseAndSortTags(tweet1);
 3   list<rstring> ys = parseAndSortTags(tweet2);
 4   return xs == ys;
 5 }
 6 stream<MatchT> Matches = MatchRegex(Tweets) {
 7  param
 8   pattern     : ". sameTags+ sameTags5th";
 9   partitionBy : author;
10   predicates  : {
11    sameTags    = tagsEq(content, First(content)),
12    sameTags5th = Count() == 5 &&
13                  tagsEq(content, First(content)) };
14  output
15   Matches : count=Count(), all=Collect(content);
16 }
```

**Figure 2: Social-media analysis pattern.**

Besides aggregations, predicates can also call normal SPL functions. Figure 2 shows an example with an SPL function *tagsEq* (Lines 1-5) called from predicates in a pattern (Lines 11+13). The only restriction is that for the operator to be safely parallelized, predicates must be deterministic and side-effect free, which the SPL compiler checks automatically.

Lines 10+11 from Figure 1 show the ***output assignments***:

```
10 Matches : symbol=symbol, seqNum=First(seqNum),
11           count=Count(), maxPrice=Max(price);
```

When a composite event has been detected, the output assignments set its attributes. Like the predicates, the output assignments also use SPL expressions. Output assignments use the same general syntax in all (library or user-defined) SPL operators, thus improving familiarity. Again, for safety, parallelization only happens if the compiler can show that the output assignments are deterministic and side-effect free; this is usually the case.

## 2.2 Aggregations

The running example from Figure 1 contains several aggregations, such as *First* and *Max*. Table 3 lists the full set of aggregations that the *MatchRegex* operator supports. The function parameters *v* refer to an attribute of the input simple events. For instance, if *ts* is an attribute with a timestamp, then *Delta*(*ts*) returns the difference between the current *ts* and *ts* at the start of the match. Most of the aggregation functions are *generic*: they have a type parameter *T*. For example, the argument *v* in `<ordered T> T Max(T v)` can be an attribute of any ordered type (number, timestamp, enum, or string), and the aggregation returns the same type.

Aggregations are *operator-specific intrinsic functions*. The *MatchRegex* operator, like many SPL operators, is implemented by code generation. In other words, the operator internally has a mini-compiler, which generates C++ code based on the operator parameters and other meta-information such as types and output assignments. This mini-compiler has special knowledge of *intrinsic* functions, and treats them specially by generating custom code for them. Specifically, aggregations in the *MatchRegex* operator are translated into code for incrementally updating a partial match each time it is extended by one more simple event. This section focuses on the semantics of aggregations that the user needs to know, deferring implementation details to Section 3.

| Prototype | Description |
|---|---|
| `int32 Count()` | Number of simple events |
| `<any T> T First(T v)` | First in match |
| `<any T> T Last(T v)` | Last in match |
| `<ordered T> T Max(T v)` | Largest |
| `<ordered T> T Min(T v)` | Smallest |
| `<numeric T> T Sum(T v)` | Sum |
| `<numeric T> T Average(T v)` | Arithmetic mean |
| `float64 Delta(timestamp v)` | Time since match start |
| `<any T> list<T> Collect(T v)` | All values as a list |

**Table 3: Operator-specific intrinsic functions supported in predicates and output assignments.**

Consider the predicate fragment *price >= Last*(*price*). A bare identifier, such as *price*, refers to an attribute of the current simple event. An aggregator call, such as *Last*(*price*), in a predicate refers to a value computed from the simple events in the partial match so far. That means that this predicate fragment checks whether the current simple event contains a higher price than that of the last (i.e., previous) simple event.

Consider the output assignment *maxPrice = Max*(*price*). The left-hand-side identifier, *maxPrice*, refers to an attribute of the matched output composite event. An aggregator call, such as *Max*(*price*), in an output assignment refers to a value computed from all simple events in the completed match.

Aggregations such as the ones that *MatchRegex* supports are familiar to database programmers. For example, in SQL, one might find the following:

```
SELECT Max(price) as maxPrice FROM Quotes WHERE ...
```

The set of supported aggregators in Table 3 includes most familiar cases. Furthermore, the *Collect* aggregator returns a list of all values for a particular attribute in a match. That maximizes expressiveness, since users can compute their own aggregations from that list. For example, Line 15 in Figure 2 assigns *all* the contents of all matching tweets, which can then be processed further by a down-stream operator.

The implementation of aggregation requires state, but that is hidden from the user. It is handled internally by the generated code so as not to interfere with safe parallelization.

## 2.3 Semantics

The main thing the user needs to know about *MatchRegex* patterns is that standard regular expression semantics apply. The pattern simply matches the input sequence of simple events in arrival order, one predicate at a time. Reusing widely-understood semantics reduces surprises. For instance, windows and time are not handled as a separate feature. In-

stead, they are handled with the existing features via the following idiom: define a predicate that refers to time (for example, `notTooLong = Delta(ts) < 0.5`), then use that predicate in the pattern (for example, `notTooLong*`).

That said, there are a few semantic choices worth mentioning: matching is non-greedy, partition-isolated, and partition-contiguous, and completed matches are non-overlapping. The rest of this section explains what that means and gives the rationale behind each design decision.

Matching is ***non-greedy***: for example, the pattern `a+` matches right away after the first simple event that satisfies predicate `a`. This property is also known as *right-minimality*. It leads to good responsiveness in a real-time setting: if matching were greedy, the operator would have to wait until the next event that *fails* predicate `a` before it can report the match for `a+`. Depending on the application, this delay may be unacceptable. However, if a user desires greedy matching, they can emulate it by an idiom that requires no syntax change: add an explicit predicate at the end of the pattern. For instance, `largeSize priceRise+ priceDrop` stops only after the price is done rising.

Matching is ***partition-isolated*** if a `partitionBy` parameter is specified. The semantics of partitioning are as if there were a separate replica of the operator for each partition (but the implementation uses maps instead to keep just a reasonable number of parallel operators). This means that there is no interference whatsoever between partial matches from different partitions. This design choice is central to this paper, as it ensures both simple intuitive semantics and parallelizability.

Matching is (partition-)***contiguous***. Within a single partition, every simple event is matched against an explicit predicate from the regular expression, and no event is implicitly skipped within a match. This choice keeps the semantics simple. If non-contiguous matching is desired, it can easily be emulated by the idiom of inserting an explicit predicate in the middle of the pattern. For example, `. notTooLong* largeIncrease` uses `notTooLong` as a skip predicate between the first and the last simple event.

Completed matches are ***non-overlapping*** in their partition. For example, consider the pattern `a+ b` and a sequence of input events $\langle x, a_1, a_2, b\rangle$ satisfying predicates $\neg$`a a a b`. There are two possible matches, a shorter one $\langle a_2, b\rangle$ and a longer one $\langle a_1, a_2, b\rangle$. The `MatchRegex` operator only reports the longer one, and discards all other partial matches when it does so. Reporting the longest match is known as *left-maximality* [18], and discarding all other partial matches is known as *skip-past-last* behavior [25]. In other words, the operator always starts over from a clean slate after reporting a match. This is consistent with the behavior of regular expressions over strings in most popular programming languages. For example, after detecting the token `then` in a string, users probably have no interest in the overlapping tokens `hen` or `he`. There are several potential causes for overlapping matches, such as overlapping individual predicates, and disjunction or closure in the regular expression. Whatever the cause, `MatchRegex` reports only matches for non-overlapping subsequences of each partition of the input stream. Note that there may be overlaps across partitions, to avoid interference.

There are not many use cases where the non-overlapping semantics are a restriction, but at least one occurs in the Cayuga benchmarks [9]. The use case aggregates over a sliding window, which, by definition, is overlapping. In SPL, sliding-window aggregations are more naturally expressed using other operators, hence there is no need for the `MatchRegex` operator to support overlapping matches.

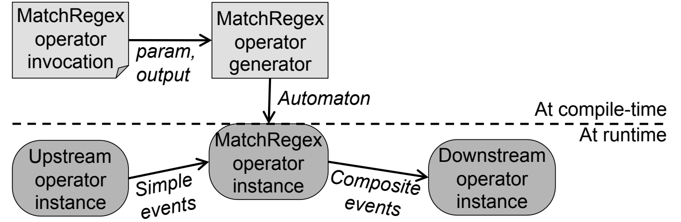



**Figure 3: `MatchRegex` operator code generation.**

# 3. PARTITIONED PARALLEL AUTOMATA

The `MatchRegex` operator works by code generation, as illustrated in Figure 3. The input to the code generator consists of the parameters, output assignments, etc. from an operator invocation such as the one for the M-shape pattern in Figure 1. At compile-time, the code generator translates the pattern into an automaton, and generates C++ code for it. At runtime, the generated C++ code serves as a stream transformer that consumes simple events and produces composite events. The underlying code-generation infrastructure is available to all SPL operator developers, and used widely for library and user-defined operators alike.

The rest of this section takes a closer look at the implementation, starting from the automaton followed by partitioning and finally auto-parallelization.

## 3.1 Translating Patterns to Automata

Using standard regular expressions as patterns, besides making the most of users' existing skill sets, has an additional advantage: it makes it possible to use standard textbook algorithms for translating patterns to automata. The `MatchRegex` operator uses Algorithm 3.36 from the "Dragon book" (second edition) [4]. If the underlying alphabet is non-overlapping, this algorithm converts a regular expression directly to a DFA (deterministic finite automaton). In the `MatchRegex` operator, the alphabet consists of user-defined predicates, which may be overlapping. For example, Figure 2 has overlapping predicates `sameTags` and `sameTags5th`. Therefore, the algorithm yields an NFA (non-deterministic finite automaton). However, unlike NFAs in general, the NFA in our case has useful additional properties: it is free of epsilon-transitions (in other words, the automaton makes exactly one state transition per input event), and each predicate appears on only one transition emanating from each state (in other words, it needs to be evaluated only once per event per partial match).

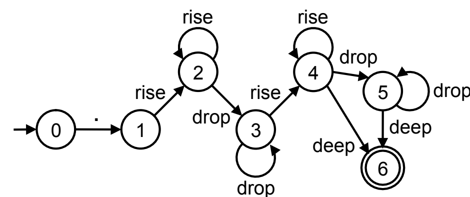
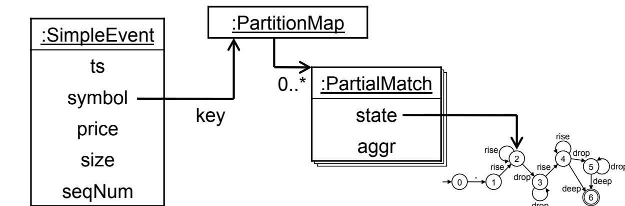



**Figure 4: Automaton for M-shape pattern `". rise+ drop+ rise+ drop* deep"` from Figure 1.**

Figure 4 shows the automaton for the running example. State 0 is the starting state. Each transition edge is labeled with a predicate, which is a boolean expression over the current input event and the partial match so far. To reach a complete match, the automaton must arrive at State 6, which is the accepting state. At this point, the *MatchRegex* operator submits a composite event.

## 3.2 Partitioning the Automata




```
type PartitionMap = map<Key, list<PartialMatch>>
type Key = tuple<rstring symbol>
type PartialMatch = tuple<int32 state, Aggr aggr>
type Aggr = tuple<
  int32 count,
  tuple<uint32 first,uint32 last,uint32 max> price,
  tuple<uint32 first> seqNum>
```

**Figure 5: Data structures for partitioned matching.**

In the presence of a *partitionBy* parameter, the *MatchRegex* operator performs partitioned pattern matching. Figure 5 shows the relevant data structures. One or more attributes of the input simple event are designated as a key (in this case, the stock *symbol*). The key is used to retrieve a list of partial matches from the partition-map. Each partial match in the list consists of an NFA state and some aggregation information *aggr*.

The aggregation information *aggr* incrementally maintains anything needed by the aggregators in predicates or output assignments. For instance, *maxPrice = Max(price)* is an output assignment that requires the maximum of attribute *price*, maintained incrementally in *aggr.price.max*. Each time a new input event has a higher *price*, the aggregation is updated. Performing aggregation incrementally one simple event at a time benefits both responsiveness and memory consumption. It improves responsiveness, because the computation is spread out evenly, instead of happening all at once when a match is completed. And it reduces memory consumption, because the operator does not actually store any past simple events. Storing past simple events is unnecessary, since the *aggr* field of the partial match summarizes all the necessary information.

Figure 6 shows a simplified version of the code generated from the running example M-shape pattern. Each incoming simple event triggers a call to the *process* function (Line 1), and each outgoing composite event is a call to the *submit* function (Line 32). Lines 2-7 retrieve the list of partial matches from the partition-map. Lines 9-11 create new partial matches for transitions from the start state. Lines 12-28 update old partial matches for transitions from other states. And, if any partial matches have reached an accepting state, Lines 29-34 submit the longest one and clear the list of partial matches for this partition. As Section 5 will demonstrate, this code performs well in practice: even without parallelization, the M-shape pattern reaches throughputs above 700,000 events per second.

```
 1 void process(SimpleEvent& evt) {
 2   Key key(evt.symbol);
 3   PartialMatchList& oldPms =
 4     partitionMap.has(key) ? *partitionMap.get(key)
 5                           : *new PartialMatchList();
 6   PartialMatchList& newPms = *new PartialMatchList();
 7   partitionMap.put(key, &newPms);
 8   int longestAccepting = -1;
 9 /*create new partial matches*/
10   if (/*predicate .*/)
11     newPms.add(new PartialMatch(1, evt));
12 /*update existing partial matches*/
13   for (int i=0, n=oldPms.size(); i<n; i++) {
14     PartialMatch& pm = *oldPms.get(i);
15     switch (pm.state) {
16       case 0: {
17         if (/*predicate .*/)
18           newPms.add(new PartialMatch(1, evt, pm));
19         break; }
20       /*similar cases for states 1-4*/
21       case 5: {
22         if (/*predicate drop*/)
23           newPms.add(new PartialMatch(5, evt, pm));
24         if (/*predicate deep*/) {
25           newPms.add(new PartialMatch(6, evt, pm));
26           /*update longestAccepting if longer*/ }
27         break; } } /*end of switch*/
28     delete &oldPms; } /*end of for*/
29 /*if any accepting match, submit longest and clear*/
30   if (longestAccepting != -1) {
31     PartialMatch& pm = *newPms.get(longestAccepting);
32     submitEvent(evt.symbol, pm.aggr.seqNum.first,
33                 pm.aggr.count, pm.aggr.price.max);
34     newPms.clear(); }
35 }
```

**Figure 6: Generated C++ code.**

## 3.3 Parallelizing the Automata

Not all patterns are as inexpensive as the M-shape finance pattern from the running example. For instance, patterns over Twitter messages may contain predicates that use costly text manipulation. In applications where the pattern is expensive, the *MatchRegex* operator can benefit from parallelization on multiple cores or machines [19].

Figure 7 illustrates the *MatchRegex* operator before (top) and after (bottom) parallelization. The idea is to split the stream of simple events into $N$ channels; send each channel to a separate replica of *MatchRegex*; and merge the $N$ channels of composite events back into a single stream. The same attribute(s) that serve as the key for the partition-map also serve as the key for the hash-split. This guarantees that if two simple events have the same key, they are sent to the same replica, thus satisfying partition contiguity. Our current implementation does not include any special provisions for load balancing in the presence of skew [7] or fault-tolerance [21], beyond what System S already provides out of the box.

The *MatchRegex* operator may not be the only parallelizable operator in the stream graph. For example, a Twitter application may first parse XML strings with raw tweets into tweet events, and then use a pattern to find composite events in the stream. Parallelizing both *ParseTweet* and *MatchRegex* yields two back-to-back parallel segments. Rather than merging all channels from the first segment only to split them again right away, System S handles this with a shuffle topology, as shown in Figure 8.

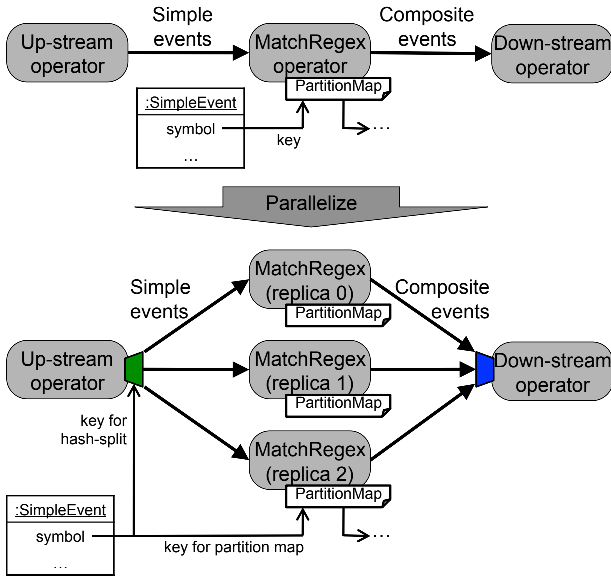



**Figure 7: Parallelizing the operator graph.**

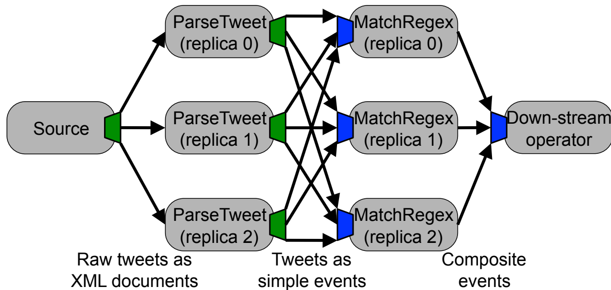



**Figure 8: Parallelized operator graph with shuffle.**

Using partitions for parallelization has several advantages. In our implementation, the parallelization is fully automatic: it takes place in the compiler and runtime system, and requires no user interaction [19]. Since the semantics guarantee partition isolation, replicas of the *MatchRegex* operator can maintain disjoint state, and no synchronization is required. In fact, the replicas do not even need shared memory, and can instead be distributed over different machines in a cluster. The amount of available parallelism is generally determined by the number of distinct keys. This number is typically in the hundreds or thousands, and thus easily supports small to moderate degrees of parallelism even in the presence of skew. Section 5 will show experiments with up to 32 parallel channels. Finally, our implementation guarantees deterministic semantics: given the same stream of simple events, *MatchRegex* always produces the same stream of composite events, irrespective of whether and how much it was parallelized. This determinism guarantee is the subject of the next section.

# 4. SAFETY AND DETERMINISM

The *MatchRegex* operator and its implementation satisfy a number of desirable semantic properties, including partition-isolation, uniqueness of the longest match, behavior preservation when parallelized, and liveness. This section explains why each property holds.

Before discussing the advanced semantic properties, it is worth mentioning that the SPL compiler and the *MatchRegex* mini-compiler implement several conventional compile-time checks. They check for syntax errors (such as an unknown token in a regular expression), name errors (such as an unknown identifier), and type errors (such as using a string where a boolean is expected). The *MatchRegex* mini-compiler also checks that transitions from the start state in the automaton are not labeled with predicates involving aggregation, because at the start state, there is no previous partial match with aggregate information.

***Partition-isolation:*** *If the* `partitionBy` *parameter is specified, there is no cross-partition interference.* This property was introduced in Section 2.3; here, we establish why it holds. As described in Section 3.2, the operator implicitly keeps track of partial matches with their NFA-state and aggregation information in a partition map. This data structure isolates implicit state of partitions from each other. Hence, the only way to violate partition-isolation would be for the user to maintain explicit state and share that across partitions. This requires writing a variable during processing for one partition, then reading the same variable during processing for another partition. In SPL, variables can be written either via an expression with side effects, such as *v++*, or via a function. The SPL type system requires any function with side effects on global state to be declared as **`stateful`**, and any written function parameters to be declared as **`mutable`** [14]. To guarantee partition-isolation, the *MatchRegex* operator checks, at compile-time, that there are no expressions with side effects, calls to stateful functions, or calls with mutable parameters in the predicates or output assignments. □

***Uniqueness of longest match:*** *If multiple matches have the same length, their aggregation information is the same.* As soon as any partial match reaches an accepting state, the *MatchRegex* operator reports the longest accepting match. The *length* of a match is the number of simple events it spans. It is possible for multiple accepting matches to be "longest". However, in that case, the operator can report any one of them, since they all have identical aggregation information, and the resulting composite event is therefore identical. This property holds for three reasons. The first reason is (partition-)contiguity: within a partition, no simple events are skipped implicitly. The second reason is complete aggregation: all events in the partial match contribute to the aggregation information (see Section 2.3). In contrast, aggregations in the *MATCH_RECOGNIZE* proposal [25] refer to subsequences of events, increasing expressiveness but reducing simplicity and optimizability. The third reason is that the NFA construction algorithm guarantees that each transition in the NFA is labeled with exactly one predicate, and thus consumes one simple event (see Section 3.1). □

***Result-set preservation:*** *The same input events yield the same output events irrespective of parallelization.* Thanks to partition-isolation, an output event for a key $k$ is affected only by simple events for that same key $k$. As discussed in Section 3.3, both the hash-split for parallelization and the partition-map use the same key. Thanks to partition-contiguity, that means that if a particular replica receives at least one input event for key $k$, that replica receives *all* input events for key $k$. It will therefore compute the same

output events. Note that even the time aggregate *Delta* is deterministic, since it relies on logical time in an event attribute, not on physical machine time. □

***Result-order preservation:*** *The order of events in the output stream is the same irrespective of parallelization.* If the *MatchRegex* operator is parallelized, each replica computes output matches for a subset of the keys. However, since the replicas run in parallel, there is a potential race condition: two output events computed by different replicas might arrive at the merge out of order. The System S runtime system resolves this issue via sequence numbers [19]. The split attaches a sequence number to each input event. When the *MatchRegex* operator detects an output event, it attaches the sequence number of the last simple event. And finally, the merge orders output events by sequence numbers, and strips the sequence numbers back off the data items before forwarding them to the downstream operator. □

***Liveness:*** *Every output event gets reported within a bounded amount of time.* If the *MatchRegex* operator is not parallelized, this property is guaranteed by eager matching, as discussed in Section 2.3. If the operator is parallelized, there is a potential problem: if the events for all keys in one replica $r_i$ do not result in any output events, then the merge must hold all output events from the other replicas, in case an input event with a lower sequence number shows up from replica $r_i$. In theory, this could cause indefinite delays or even deadlocks [16]. In practice, System S bounds the delays by using *pulses* [19]. Every once in a while, the split sends a pulse to all replicas. The pulse does not affect pattern matching, but each replica of the *MatchRegex* operator forwards the pulse. When the merge receives pulses from all replicas, it submits all pending output events. □

Taken together, these properties allow users to write predictable and deterministic applications. Parallelization only affects performance, it does not change the output. Determinism is often useful for testing, and in fact, we used it in all the performance experiments for Section 5 to validate that the functional behavior of the applications was correct.

# 5. RESULTS

This section uses several benchmark applications from finance and social media to measure the performance of the *MatchRegex* operator. These benchmarks also illustrate what kind of applications can be easily expressed. Section 5.1 describes the methodology, and Section 5.2 reports the performance results.

## 5.1 Methodology

All experiments for this paper ran on machines with two Intel Xeon processors, where each processor has four cores, for a total of eight cores per machine. Each machine has a clock-speed of 3 GHz and a main memory of 64 GB. For all experiments, the input source and output sink ran on different machines and communicated with the core benchmark via high-speed ethernet. Since the machines were not reserved for our experiments, there was a certain amount of noise from other system activity. To compensate for noise, each data point is the arithmetic mean of ten runs.

Table 4 characterizes the input data sets. The finance data set consists of 10,000 NASDAQ trades repeated 1,000 times with adjusted timestamps, totaling 10,000,000 simple events. The simple events have type *Trade*, with attributes for the timestamp, stock symbol (e.g., "IBM"), price (in cents), size (in shares), and a sequence number. There are 390 different traded symbols in the data set. The twitter data set consists of 10,000 tweets repeated 20 times, totaling 200,000 simple events. The tweets are selected by querying the 25 most common hash-tags, such as #business, #writers, etc. The data set contains tweets from 6,142 different authors. Raw tweets come in as strings of XML conforming to the Atom syndication format. After parsing, the simple events have type *Tweet*, with attributes for the ID, timestamp, author, and message content. Using data sets based on real stock trades or tweets, respectively, yields distributions and time series that are representative of what might happen in practical deployments.

| Name | Type | Key | # Keys | # Events | Logical time |
|---|---|---|---|---|---|
| finance | Trade | symbol | 390 | 10,000,000 | 2 h 01 min |
| twitter | Tweet | author | 6,142 | 200,000 | 36 h 40 min |

```
type Trade = tuple<
  timestamp ts, rstring symbol,
  uint32 price, uint32 size, uint32 seqNum>;
type Tweet = tuple<
  uint64 id, timestamp ts, rstring author,
  rstring content>;
```

**Table 4: Input data sets.**

Table 5 characterizes the benchmarks. The first six benchmarks use the finance data set, and the remaining three benchmarks use the twitter data set.

Benchmark finance0 is based on the pattern used for performance experiments by Agrawal et al. [3]. Benchmarks finance1 thru finance5 are adapted from the Cayuga webpage [9]. Specifically, they correspond to Subscription 1 thru 5, respectively, of their application scenario "technical analysis for stock investors". Cayuga supports a feature called *resubscription*, where the composite events detected by a first pattern are fed as simple events into a second pattern. This is used by finance3 and finance4. The first pattern computes a sliding average, which requires overlapping matches that *MatchRegex* does not support. Instead, we replaced the first pattern by a more conventional aggregation operator, taking advantage of the underlying general-purpose streaming system. Our compiler was able to parallelize both operators.

Each of the twitter benchmark starts with a *ParseTweet* operator, which turns a raw tweet (XML string) into a *Tweet* event (see Table 4). This operator is expensive but parallelizable. To isolate the parsing overhead, twitter0 only does parsing without pattern-matching, whereas twitter1 and twitter2 do parsing plus matching a pattern each. Since both parsing and matching are parallelizable, the topology contains a shuffle, as shown in Figure 8.

The *selectivity* column in Table 5 measures the ratio of output events to input events. For example, finance0 produces 0.0171 output events per input event. Since twitter0 does no filtering, it is not selective, and produces exactly one output event per input event.

All benchmarks in this section use partitioning. All six finance benchmarks are based on prior work, where they were partitioned as well. The twitter benchmarks are not based on prior work, but are naturally partitioned by authors.

## 5.2 Performance

This section presents results for absolute throughput, as well as speedups on a single eight-core machine and on a shared-nothing cluster of four eight-core machines.

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

**Table 5: Benchmarks.**

| Benchmark | 1 : Non-parallel | Width : Best parallel |
|---|---|---|
| finance0 | 1 : 778,584 ± 8.5% | 1 : 778,584 ± 8.5% |
| finance1 | 1 : 708,118 ±16.3% | 1 : 708,118 ± 16.3% |
| finance2 | 1 : 832,469 ±15.4% | 1 : 832,469 ± 15.4% |
| finance3 | 1 : 229,090 ± 5.7% | 4 : 732,837 ± 15.0% |
| finance4 | 1 : 371,753 ± 9.6% | 4 : 670,781 ± 18.5% |
| finance5 | 1 : 235,672 ± 4.2% | 8 : 737,656 ± 7.1% |
| twitter0 | 1 : 6,488 ± 8.1% | 32 : 87,158 ± 3.7% |
| twitter1 | 1 : 6,267 ± 6.8% | 16 : 66,492 ± 22.3% |
| twitter2 | 1 : 316 ± 0.8% | 32 : 2,378 ± 2.0% |

**Table 6: Absolute throughput in events/second.**

Table 6 presents absolute throughputs to explore the questions: How fast is the *MatchRegex* operator? And what is the best degree of parallelism? Each entry is of the form *width* : *throughput* ± *standardDeviation*%. The width is the number of parallel channels, in other words, the number of replicas of parallelized operators; the throughput is the average number of input events per second over ten runs for each data point; and the standard deviation is computed over the same ten runs.

The highest throughput is 832,469 events per second for finance2. This is an extremely good number for CEP engines, to our knowledge exceeded only by the Woods-Teubner-Alonso CEP engine on FPGAs [23]. The *MatchRegex* operator achieves this good performance even without parallelization thanks to its incremental aggregation, which obviates the need to store and join old simple events. Overall, the fastest non-parallel benchmarks are finance0, finance1, and finance2, and parallelizing them yields no further speedup, since matching is already so fast that it does not constitute a bottleneck. Perhaps surprisingly, these three fastest benchmarks have the most sophisticated regular-expression patterns in our benchmark suite, indicating that extra states in the automaton incur no extra cost. At the other end of the spectrum, the three twitter benchmarks are slowest. For twitter0 and twitter1, that can be blamed on parsing, not pattern matching, as demonstrated by twitter0, which does only parsing. As of the time of this writing, the twitter.com website advertises 250 million tweets per day, which averages to 2,893 tweets per second, so the parallel throughput of 66,492 tweets per second in twitter1 is more than fast enough. The twitter2 benchmark is the slowest. Its regular expression starts with `.+`, which means that every event initiates a new partial match. But since twitter2 has a selectivity of 2.15%, only about 1 in 47 events end a partial match, leading to a lot of in-progress matches for the operator to handle. This is exacerbated by the fact that the *disjointTags* predicate involves further parsing of the tweet contents to extract hash-tags, and comparing sets of hash-tags to determine disjointness. To get better twitter2 results, users could precompute the tag sets using a separate operator and store them in an attribute.

Looking at the results after parallelization, all finance benchmarks reach throughputs exceeding 600,000 events per second. Depending on the benchmark, the best parallel width ranges from 1, 4, or 8 to 16 or 32. This motivates online elastic scaling as a fruitful area for future work, which we are actively pursuing. Overall, Table 6 demonstrates that aside from twitter2, which has both a slow pattern and slow predicates, the *MatchRegex* operator achieves high throughputs.

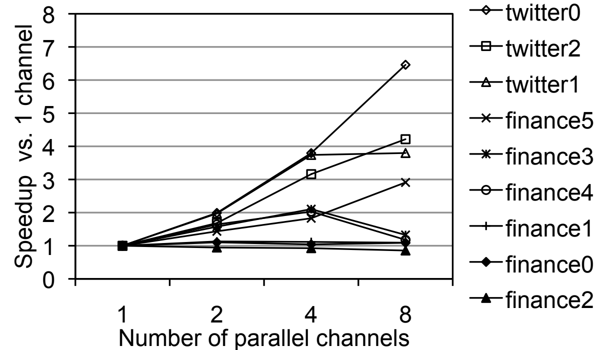



**Figure 9: Speedup from parallelizing on a single multi-core machine with 8 cores.**

Figure 9 presents speedups on an eight-core machine to explore the question: How much does non-distributed parallelism help? The x-axis shows the width, i.e., the number of operator replicas. The y-axis shows the speedup, i.e., the throughput at width $X$ normalized to the throughput at width 1 for the same benchmark. In this graph, the benchmarks are ordered by their 8-way speedup to make the mapping between the curves and the legend easy to read. The biggest speedup factors are 6.5, 4.2, and 3.8 for the three twitter benchmarks. Those are the three benchmarks with the worst absolute performance when non-parallel. Although with 8-way parallelism on an 8-way machine, a perfect speedup would be 8×, this did not happen in practice due to resource contention on the memory hierarchy and network bandwidth to the source and sink. Among the finance benchmarks, the biggest speedup factors were 2.9 for finance5 on 8 cores and 2.1 for finance3 on 4 cores. Again, those two finance benchmarks have the lowest throughput when non-parallel. The finance2 benchmark experiences the worst speedup factor of 0.9. This is not too bad, and is

probably caused by a combination of parallelization overhead, noise, and load imbalance. Overall, the multi-core results demonstrate healthy speedups for the benchmarks that need it most, without unduly slowing down other benchmarks that already perform well.

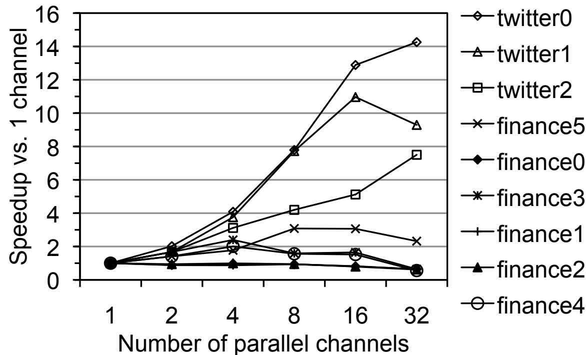



**Figure 10: Speedup from parallelizing on a cluster of 4 machines with 8 cores each = total of 32 cores.**

Figure 10 presents speedups on a shared-nothing cluster of four eight-core machines to explore the question: How much does distributed parallelism help? The x-axis shows parallel width, the y-axis shows speedups, and benchmarks are ordered by their 32-way speedup. The three twitter benchmarks experience the biggest speedup factors of 14.3, 11.0, and 7.5, which is much faster than parallelizing them on a single eight-core machine only. In the range of 1-8 channels, the results are better than in Figure 9, because the same number of channels are distributed over more machines. But again, there is no perfect speedup, because multiple replicas compete for memory hierarchy and network resources on each machine. The finance4 application experiences the worst speedup of 0.6 at width 32, indicating that the multi-machine cluster is more vulnerable to parallelization overhead and noise than a single multi-core machine. This motivates future work on load balancing. Overall, the results show that distributed parallelism can yield substantially higher speedups than shared-memory parallelism. The `MatchRegex` operator is well-suited to distributed execution thanks to its semantic choices. In particular, the partition-isolation property means that no shared memory is required for its data structures.

# 6. RELATED WORK

This section surveys related work on event and stream processing, focusing in particular on language design, matching techniques, partitioning, and parallelism.

Early CEP (complex event processing) systems were not parallel. One of the earliest CEP systems was *NiagaraCQ* [10]. NiagaraCQ used XML-QL as its query language, which was one of the precursors of XQuery. It implemented pattern matching using a graph of algebraic operators. The NiagaraCQ paper made no mention of partitioning. *SQL-TS* extended SQL with pattern-matching over row sequences [18], and included syntax for partitioning. Another early system was *Amit* [2]. Amit queries were themselves written as XML documents, and implemented via back-tracking. Amit performed partitioning based on equality conditions on keys. *SASE* was a CEP language implemented via NFA$^b$ automata (non-deterministic finite automata with buffers) [24, 3]. Like Amit, SASE performed partitioning based on equality conditions on keys, but the language directly supported partitioning by providing specialized syntax for it. The `MATCH_RECOGNIZE` proposal extended SQL-TS with several additional features, most importantly, full-fledged regular expressions and aggregation [25]. Like its predecessor SQL-TS, it supported partitioning. Finally, *EventScript* specified patterns as regular expressions interspersed with action blocks [11]. EventScript patterns were implemented as DFAs (deterministic finite automata), and EventScript specified partitioning in a group clause. We observe that most of these systems supported partitioning in one form or another, but none of them were parallel.

Early general streaming languages did not directly support CEP patterns. *CQL* (the Continuous Query Language) was a dialect of SQL for streaming, implemented via algebraic operators [6]. The CQL paper was silent on the topic of parallelism. *Borealis* is a streaming system where users can graphically compose (mostly) relational streaming operators [1]. Not only is Borealis parallel, it is also distributed. *SPL* (the Streams Processing Language) [14], formerly known as *SPADE* [13], is a language for describing graphs of stream operators. SPL comes with a code-generation infrastructure for synthesizing general operators in C++ [17]. The graph-of-operators paradigm enables SPL to capture various different flavors of stream processing [22]. SPL runs on System S, which, like Borealis, is parallel and distributed [5]. This paper introduces a parallelizable CEP operator for SPL.

A few recent CEP systems started exploring parallelism. Brenna et al. published a study where they distributed *Cayuga* via a “set of scripts to create configuration files that contain the mapping between Cayuga stream identifiers on the query level to multicast groups” [8]. Their baseline system, Cayuga, is an algebraic CEP system with partitions based on equality conditions on keys [12]. While their system requires separate configuration files for parallelization, our system fully automates parallelization. The *Next* system also takes an algebraic approach to CEP, and scales by automatically placing different operators of the same pattern on different hosts [20]. However, unlike our work, Next does not parallelize based on partitions, and computes no aggregate information for detected events. *EventJava* supports CEP as patterns guarding event methods [15]. It detects equality conditions on keys as partitions and uses them to build sophisticated index data structures. Like EventJava, our system can also deploy different event handlers on different machines, but in addition, our system also parallelizes individual patterns, which EventJava does not do. Woods, Teubner, and Alonso showed how to implement CEP on FPGAs [23]. They use regular expressions for patterns, make partitions explicit, and implement patterns via NFAs. Their matching engine exploits fine-grained parallelism on the FPGA to evaluate many predicates simultaneously and perform many state machine transitions simultaneously. However, unlike our work, they lack partition-parallelism, and they only perform pattern detection, without computing aggregate information for detected events.

# 7. CONCLUSIONS

This paper describes a CEP operator in a general-purpose streaming system. The user specifies patterns as regular ex-

pressions with predicates and aggregations. The operator is implemented by a code-generator, which translates the pattern into an automaton, and implements the aggregations incrementally, saving both time and space. Acknowledging that most CEP matching is naturally partitioned, the user can declaratively specify a partitioning key, and the operator exploits that key for parallelization. This paper demonstrates that the resulting generated code is fast, reaching throughputs up to 830,000 events per second for cheap patterns, and yielding up to 14× parallel speedups for expensive patterns.

By bringing together complex event processing with general stream processing, this paper opens the door to more expressive continuous analytics. The `MatchRegex` operator, taken by itself, is simpler than a full-fledged CEP engine. But the surrounding streaming system supplements it in two ways: by letting it interact with other operators, and by letting it exploit common runtime services. Other operators can perform additional functionality both before pattern matching (e.g., parsing, classifying, aggregating) and after pattern matching (e.g., joining, reporting, or even additional instances of the `MatchRegex` operator). The runtime system can provide common optimizations (such as fusion and parallelization) and other services (such as visualization and management). Keeping the operator simple reduces the implementation burden, and leads to leaner semantics, making CEP easier to use.


## Acknowledgments

Thank you to Richard King, Scott Schneider, John Morar, Mark Mendell, and Robert Soulé for feedback on drafts of this paper. I thank Buğra Gedik for pointing me to the work of Woods, Teubner, and Alonso [23], which inspired the syntax in this paper, and the anonymous reviewers for their constructive criticism and for pointing out that the syntax in the `MATCH_RECOGNIZE` proposal [25] is even more similar. Thanks to the entire System S team for their encouragement and feedback for the operator described in this paper.


# 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)*, pages 277–289, 2005.

[2] A. Adi and O. Etzion. Amit – the situation manager. *Journal on Very Large Data Bases (VLDB J.)*, pages 177–203, 2004.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *International Conference on Management of Data (SIGMOD)*, pages 147–160, 2008.

[4] A. Aho, M. S. Lam, R. Sethi, and J. Ullman. *Compilers: principles, techniques, & tools*. Addison-Wesley, second edition, 2007.

[5] 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)*, pages 27–37, 2006.

[6] 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.)*, pages 121–142, 2006.

[7] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 10–22, 1999.

[8] L. Brenna, J. Gehrke, D. Johansen, and M. Hong. Distributed event stream processing with non-deterministic finite automata. In *Conference on Distributed Event-Based Systems (DEBS)*, 2009.

[9] Cayuga webpage with benchmark descriptions. http://www.cs.cornell.edu/bigreddata/cayuga/.

[10] 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)*, pages 379–390, 2000.

[11] N. H. Cohen and K. T. Kalleberg. EventScript: An event-processing language based on regular expressions with actions. In *Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 111–120, 2008.

[12] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Conference on Innovative Data Systems Research (CIDR)*, pages 412–422, 2007.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *International Conference on Management of Data (SIGMOD)*, pages 1123–1134, 2008.

[14] 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.

[15] K. R. Jayaram and P. Eugster. Scalable efficient composite event detection. In *Coordination Models and Languages*, pages 168–182, 2010.

[16] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 243–252, 2010.

[17] M. Mendell, H. Nasgaard, E. Bouillet, M. Hirzel, and B. Gedik. Extending a general-purpose streaming system for XML. In *Extending Database Technology (EDBT)*, 2012.

[18] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Principles of Database Systems (PODS)*, pages 71–81, 2001.

[19] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[20] N. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Conference on Distributed Event-Based Systems (DEBS)*, 2009.

[21] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *International Conference on Management of Data (SIGMOD)*, pages 827–838, 2004.

[22] 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)*, pages 507–528, 2010.

[23] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. In *Very Large Data Bases (VLDB)*, pages 660–669, 2010.

[24] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *International Conference on Management of Data (SIGMOD)*, pages 407–418, 2006.

[25] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern matching in sequences of rows. Technical report, ANSI Standard Proposal, 2007.