RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing

Cagri Balkesen, Nihal Dindar, Matthias Wetter, Nesime Tatbul

ETH Zurich, Switzerland

{cagri.balkesen, dindarn, wetterma, tatbul}@inf.ethz.ch

ABSTRACT

Recognition of patterns in event streams has become important in many application areas of Complex Event Processing (CEP) including financial markets, electronic healthcare systems, and security monitoring systems. In most applications, patterns have to be detected continuously and in real-time over streams that are generated at very high rates, imposing high-performance requirements on the underlying CEP system. For scaling CEP systems to increasing workloads, parallel pattern matching techniques that can exploit multi-core processing opportunities are needed. In this paper, we propose RIP - a Run-based Intra-query Parallelism technique for scalable pattern matching over event streams. RIP distributes input events that belong to individual run instances of a pattern's Finite State Machine (FSM) to different processing units, thereby providing fine-grained partitioned data parallelism. We compare RIP to a state-based alternative which partitions individual FSM states to different processing units instead. Our experiments demonstrate that RIP's partitioned parallelism approach outperforms the pipelined parallelism approach of this state-based alternative, achieving near-linear scalability that is independent from the query pattern definition.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query Processing

Keywords

CEP; Pattern Matching; Stream Processing; Parallelism

1. INTRODUCTION

Complex Event Processing (CEP) has become a critical technology with a wide range of well-known application domains from financial trading to health care. An essential capability for CEP systems is the ability to match patterns over sequences of events. These patterns are typically specified as regular expressions with event variables that are then used to define predicates over individual event occurrences as well as correlations across them. As such, pattern queries can be arbitrarily complex, imposing high computational complexity over the CEP systems that are executing them.

In real-time CEP applications, there is an additional need to detect patterns continuously over streaming event sequences. The time-sensitive nature of events as well as the need to keep up with potentially very high event arrival rates in such settings further exacerbate the performance challenges faced by CEP systems.

For CEP systems to be able to cope with ever-increasing input and query workloads, they must be equipped with techniques that can use modern computing technologies to their advantage. In particular, over the past several years, we have seen an uprising trend in multi-core processing technologies, which presents a ripe opportunity for ensuring high-throughput CEP.

In this paper, the research question we aim to answer is how to exploit inherent parallelism in modern multi-core CPU architectures for scalable processing of CEP queries over event streams. We focus on a common subset of continuous `MATCH-RECOGNIZE` queries [18] and follow a query execution model based on Finite State Machines (FSMs). Given a query, our goal is to map its FSM execution onto parallel processing units in a multi-core machine. Furthermore, this should be done in a way that scales processing throughput (*i.e.*, the number of input events that can be processed per unit time) with the increasing number of cores, as close to the ideal linear scale-up target as possible.

We propose a novel solution, RIP - a Run-based Intra-query Parallelism technique. RIP distributes input events that belong to individual run instances of a query's FSM to different processing units, thereby providing fine-grained partitioned data parallelism that is independent from the query pattern definition. As we show with a detailed experimental study that is also verified with real-world data workloads, RIP achieves throughput scalability that is very close to the ideal. Furthermore, we compare RIP to a sequential baseline as well as an alternative parallel approach that partitions individual FSM states (as opposed to whole FSM instances in RIP) to different processing units. RIP outperforms this state-based approach under all workload scenarios. In addition to its inferior performance, the state-based approach also has other undesirable limitations such as being query-dependent and bounding scalability by the number of FSM states in the query. In essence, our RIP ap-

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

proach represents partitioned parallelism whereas the state-based approach represents pipelined parallelism. Thus, a significant outcome of this paper is that it clearly shows that partitioned parallelism is a better fit for multi-core CEP than pipelined parallelism.

Partitioned data parallelism for CEP itself is not a new idea [13]. However, previous approaches partition input events directly based on the "`PARTITION BY`" clause, not at the FSM instance level. Thus, our RIP approach is orthogonal/complementary to these previous approaches in the following ways: (i) Not all CEP queries have "`PARTITION BY`" clauses, in which case RIP is the only possible approach for data parallelism, (ii) Even in the presence of "`PARTITION BY`", RIP provides a way to parallelize individual data partitions further, which enables finer-grained partitioning across processing cores. Furthermore, RIP does not suffer from the data skew problem, which is an inherent problem in approaches that are solely based on "`PARTITION BY`" parallelism.

The rest of this paper is organized as follows: Section 2 describes the data and query models that underlie this work. In Section 3, we describe our basic sequential query processing algorithm and architecture, which sets a baseline for our work. We present out parallel query processing techniques in Section 4. Section 5 provides our experimental results. We summarize the related work in Section 6 and finally conclude the paper in Section 7.

2. DATA AND QUERY MODEL

In this section, we present main assumptions and definitions regarding the data and query model of RIP.

We start with the time domain. We assume that the time domain is a discrete, linearly ordered, countably infinite set of time instants. It is bounded in the past, but not necessarily in the future. Each event has a timestamp and consists of a relational tuple conforming to a schema. Furthermore, we assume that a stream is a totally ordered, countably infinite sequence of events, such that the total order is defined by the timestamps. The CEP annotates each event in the ordered stream by a unique id such that the difference of the ids of two subsequent events is 1. In Example 1 we provide a sample stream specification.

Example 1 *A stream of events that consists of stock market ticks can be specified as follows:*

```
StockTrade(tradeId Long, symbol Char(8), price Float,
          timestamp Long)={
              (1, A, 34.52, <01/19/2006 9:34:01>),
              (2, A, 34.54, <01/19/2006 9:34:01>), ...}
```

Next, we explain how pattern matching queries are specified over a given input stream. Different languages are proposed for pattern matching queries [10, 7, 9, 18]. We use a subset of the `MATCH-RECOGNIZE` clause [18] to express pattern matching queries. `MATCH-RECOGNIZE` is originally a proposal for a SQL extension that performs pattern matching on rows. Nevertheless, such queries can also be applied to streams of events which have already supported by some CEP engines [2, 6]. We will briefly discuss a subset of `MATCH-RECOGNIZE` syntax, our semantics, and then provide an example. By default, contiguous matches are searched in a given stream. However, pattern matching can also be performed in a subset of data, when optional `PARTITION BY` clause is used, such as looking for patterns for each stock symbol separately. Patterns are specified with regular expressions using pattern variables (`PATTERN` clause). Each pattern variable is accompanied by a definition in the `DEFINE` clause. This definition consists of a conjunction of boolean predicates. If an event meets this definition, it can be classified with the corresponding variable. The absence of a definition of a certain variable means that any event can be classified with that variable name. Furthermore, events can be correlated by specifying predicates over pattern variables, such that price of a $StockTrade$ event annotated as B must have a price value greater than the event just before itself ($B.price >$ `PREV`$(B.price)$). For example, a pattern specification `PATTERN`$(AB * C)$ states that we are looking for an event (classified as A) that is followed by zero ore more events that satisfy the definition of B and by exactly one event that satisfies the definition of C. We call pattern specifications which contain Kleene *,+ variable-length patterns, whereas we call patterns containing neither Kleene * nor Kleene + fixed-length patterns. Please note that search for variable-length patterns might never terminate, such as when predicate of each pattern variable is *true*. In order to avoid that, we assume that the maximum length (`MAXLENGTH`) of a match is specified in the query. `MAXLENGTH` clause is an extension to the original proposal. Next, we provide a sample pattern specification from finance domain.

Figure 1: Head and shoulders patterns in a stock trade stream [1]

In finance, recognition of chart patterns within stock prices is a common method which is used to predict the future development of stock prices. A well-known pattern is the so-called head and shoulders pattern, which belongs to the family of reversal patterns [4]. If an uptrend of the price is accompanied by the observation of the head and shoulders pattern for a stock, a drop of the price can be forecast for that stock.

Example 2 *Head and shoulders pattern consists of three peaks: (i) the left shoulder that is formed after an uptrend, (ii) the highest peak called the head and (iii) the right shoulder that shows an increase but fails to take out the previous high (head) [3]. The pattern is complete when the price falls below the neckline that is formed by connecting the two low points next to the head [3, 1]. Figure 1 depicts the pattern. A simplified head and shoulders pattern in StockTrade stream can be expressed with `MATCH-RECOGNIZE` (see Query 1).*

```
SELECT  symbol, left_shoulder_top, head_top,
        right_shoulder_top, point_of_reversal
FROM StockTrade MATCH_RECOGNIZE (
PARTITION BY symbol
MEASURES symbol  AS s_symbol,
         C.price AS left_shoulder_top,
         G.price AS head_top,
         K.price AS right_shoulder_top,
         M.price AS point_of_reversal
PATTERN  (A B* C D* E F* G H* I J* K L* M)
DEFINE
       B AS (B.price ≥ PREV(B.price)),
       C AS (C.price > PREV(C.price)),
       D AS (D.price ≤ PREV(D.price)),
       E AS (E.price < PREV(E.price)
                   AND E.price > A.price),
       F AS (F.price ≥ PREV(F.price)),
       G AS (G.price ≥ PREV(G.price)
                   AND G.price > C.price),
       H AS (H.price ≤ PREV(H.price)),
       I AS (I.price < PREV(I.price)
                   AND I.price > A.price),
       J AS (J.price ≥ PREV(J.price)),
       K AS (K.price > PREV(K.price)
                   AND K.price < G.price),
       L AS (L.price ≤ PREV(L.price)),
       M AS (M.price ≤ PREV(M.price)
                   AND M.price < E.price
                   AND M.price < I.price)
MAXLENGTH 100
)
```

Query 1: Head and shoulders pattern

After specifying a pattern matching query, next we focus on its result tuples. Output of the query is a sequence of matches, where each match contains a sequence of events. Each event in that sequence is classified by a variable such that the sequence of variables conform to the pattern of a given query and the predicates are fulfilled accordingly. Figure 2 depicts a match of Query 1.

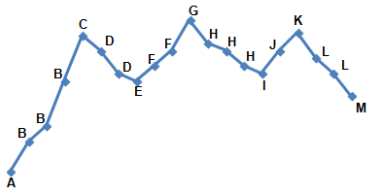

Figure 2: A sample head and shoulders match

Matches can have arbitrary relations depending on the predicates and the input stream, such as a match can include events of another match partially or fully. Some language proposals enable to specify certain match behaviors in the query such as non-overlapping only, or the longest matches [9, 18, 13]. We focus on outputting all matches that can be found, as it is more general and computationally more expensive. Please note that even in the case of non- overlapping match search, all events must be processed as overlapping can only be detected after a match is found, and matches can have arbitrary starts and sizes. Furthermore, we assume that a single event per match is reported which summarizes the events belonging to the match. `MEASURES` clause is used to specify the fields of a match event in the `MATCH-RECOGNIZE` proposal such as *head_top* is the highest price of the stock in Query 1. Next, we give a sample output event to Query 1.

```
HeadShoulders(Id Float, s_symbol Char(8), left_shoulder_top Float,
              head_top Float, right_shoulder_top Float,
              point_of_reversal Float)={
                    (1, A, 35.0, 37.5, 35.5, 32.1), ...}
```

We use FSM to model pattern matching queries. Since patterns are specified with regular expressions, FSM is a natural choice. Furthermore, It is widely used by other pattern matching engines [16, 10, 11]. Every pattern is transformed into a non-deterministic FSM. Each pattern variable is represented by states in the FSM. An FSM has always a start state (indicated by s_0). The predicates are represented by edges (indicated by p_i). Next, we provide a sample FSM.

Example 3 *FSM of Query 1 is as follows:*

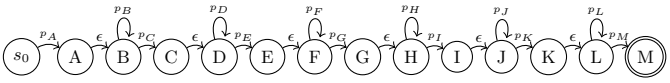

Figure 3: FSM for Query 1

We presented our main assumptions and definitions regarding pattern matching in general. A CEP engine is required to find all possible matches over a given input stream according to a given query regardless of its processing strategy. In the following sections, we discuss how pattern matching queries can be processed sequentially and in parallel.

3. QUERY PROCESSING

This section describes a basic sequential query processing algorithm and architecture for pattern matching.

Figure 4 depicts a simple model of a sequential pattern matching engine. In the figure, the client program feeds the engine with input events and receives the match results. The engine performs three main tasks: receive events, evaluate events, control matches. Next, we discuss each of these tasks.

Input handler is a simple interface that provides event streams to the engine. If `PARTITION BY` is specified in the query, the input can be partitioned at the input handler. Similarly, match output handler deals with outputting the detected matches, such that it formats the found matches with regard to the output specifications given in the `MEASURES` clause and sends them to the client. Pattern evaluator performs the main task, which is evaluating events and finding all matches.

Pattern evaluator uses the FSM of the query to evaluate patterns. Each event might contribute to a match and must be evaluated with the FSM. In order to move from one state of the FSM to another state, an event has to be consumed. Furthermore, the movement is only allowed if the event fulfills the predicates of the edge. An exception is the edges that are labeled by an epsilon. These edges do not consume an event and thus it is possible to move to another state without classifying an event. Arriving at a final state means that a match is found. However, finding no edges to move means that the search ended with non-match.

Example 4 *We illustrate execution of part of the head and shoulders query (Query 1), more specifically until the head is*

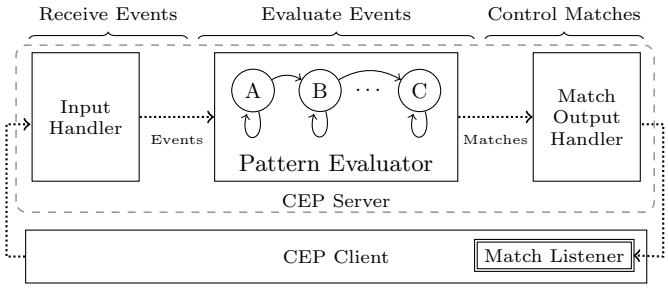

Figure 4: Basic Query Processing Architecture

detected. FSM of the query can be found in Figure 5. Please note that Figure 5 depicts the first half of the FSM shown in Figure 3, with explicit predicate definitions.

We call an execution instance of an FSM a "run". Each run holds a pointer to the current state of the FSM (denoted as *curr_state*) and a set of events which conformed to the predicates till the current state (partial match denoted as PM). A run might or might not lead to a match. For each incoming event, pattern evaluator creates a new run having the start state of the FSM as its current state and an empty partial match. After that, each run is updated regarding to the incoming event (see Algorithm 1). In Example 5, we illustrate the execution of Algorithm 1.

Algorithm 1: Sequential pattern matching

Input: e : event, R : set of runs, *fsm* : FSM of the query
Result: R': updated set of runs

```
1  begin
2      R' ← ∅;            // create an empty set of runs
3      r_i ← create a new run;
4      r_i.current-state ← fsm.start-state;
5      r_i.partial-match ← ∅;
6      R.add(r_i);                          // add r_i to R
7      foreach run r in R do
8          S ← computeReachableStates(r.current-state, e);
9          foreach state s in S do
10             if s.evalPredicates(r.partial-match, e) = true
               then
11                 add e to r.partial-match;
12                 r.current-state ← s;
13                 if s is a final-state then
14                     report the r.partial-match;
15                     if s has outgoing edges then
16                         r_j ← make a copy of r;
17                         R'.add(r_j);
18                 else
19                     r_j ← make a copy of r;
20                     R'.add(r_j);
21     R ← R';
22     return R';
```

Example 5 *A StockTrade input stream is given with the following price values for a certain stock symbol:*

```
StockTrade(tradeId Long, price Float)= {(1,33.1),(2,34.0),
                                        (3,33.2),(4,35.0), ...}
```

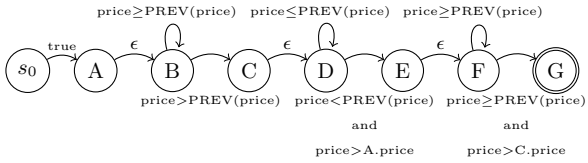

Figure 5: FSM for Example 5

event (price)	run r_{id}:<curr_state, PM> → <curr_state', PM'>	Prd
$e_1(33.1)$	$r_1 :< s_0, \emptyset >\rightarrow< A, (a_1) >$	p_A
$e_2(34.0)$	$r_2 :< s_0, \emptyset >\rightarrow< A, (a_2) >$	p_A
$e_2(34.0)$	$r_1 :< A, (e_1) >\rightarrow< B, (a_1, b_2) >$	p_B
$e_2(34.0)$	$r_1' :< A, (e_1) >\rightarrow< C, (a_1, c_2) >$	p_C
$e_3(33.2)$	$r_3 :< s_0, \emptyset >\rightarrow < A, (a_3) >$	p_A
$e_3(33.2)$	$r_2 :< A, (a_2) >\rightarrow -$	p_B
$e_3(33.2)$	$r_2 :< A, (a_2) >\rightarrow -$	p_C
$e_3(33.2)$	$r_1 :< B, (a_1, b_2) >\rightarrow -$	p_B
$e_3(33.2)$	$r_1 :< B, (a_1, b_2) >\rightarrow -$	p_C
$e_3(33.2)$	$r_1' :< C, (a_1, c_2) >\rightarrow< D, (a_1, c_2, d_3) >$	p_D
$e_3(33.2)$	$r_1'' :< C, (a_1, c_2) >\rightarrow< E, (a_1, c_2, e_3) >$	p_E
$e_4(35.0)$	$r_4 :< s_0, \emptyset >\rightarrow< A, (a_4) >$	p_A
$e_4(35.0)$	$r_3 :< A, (a_3) >\rightarrow< B, (a_3, b_4) >$	p_B
$e_4(35.0)$	$r_3' :< A, (a_3) >\rightarrow< C, (a_3, c_4) >$	p_C
$e_4(35.0)$	$r_1' :< D, (a_1, c_2, d_3) >\rightarrow -$	p_D
$e_4(35.0)$	$r_1' :< D, (a_1, c_2, d_3) >\rightarrow -$	p_E
$e_4(35.0)$	$r_1'' :< E, (a_1, c_2, e_3) >\rightarrow$ $< F, (a_1, c_2, e_3, f_4) >$	p_F
$e_4(35.0)$	$r_1''' :< E, (a_1, c_2, e_3) >\rightarrow$ $< \mathbf{G}, (a_1, c_2, e_3, g_4) >$	p_G
...	...	...

Table 1: Event evaluation sequence for Example 5

Other attributes of StockTrade are avoided for brevity. The query having the FSM shown in Figure 5 is executed over StockTrade stream by a sequential pattern matching engine according to Algorithm 1. The evaluation sequence can be found in Table 1. Table 1 shows new runs created with each incoming event and changes in the states of both the existing and new runs when the corresponding predicate (prd) is executed. p_X denotes the predicate of state X. Dashes(−) mean that the run is finalized with a non-match. Please note that, reachable states from a given state are the set of states which can be reached by consuming one event. For example, in the FSM shown in Figure 5 reachable states from state A are both state B and C. If multiple edges can be followed, a run is created for each (e.g., r_1 and r_1' in Table 1). a_1 denotes that the event e_1 was classified by A. In the example, r_1''' reaches final state G and thus reports a match including events from e_1 to e_4.

We will use the architecture in Figure 4 as a basis for further parallelization approaches. Next, we discuss how the work done by Pattern Evaluator can be parallelized.

4. PARALLEL QUERY PROCESSING

In order to parallelize the basic query processing architecture of Figure 4, the first step is to divide the CEP server thread into three threads, one for each task (*i.e.*, Input Handler, Pattern Evaluator, and Match Output Handler), and to replace method calls between task boundaries with queues accordingly. Of these three threads, Pattern Evaluator has the highest computational cost and therefore is likely to be

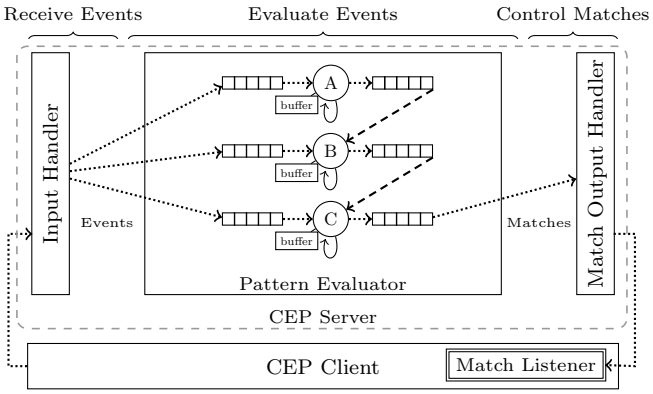

Figure 6: State-based Parallelization

the bottleneck in the face of fast input event arrival and a complex pattern to evaluate. How can the Pattern Evaluator thread be further parallelized?

Given the event evaluation sequence (*e.g.*, Table 1) generated by the sequential pattern matching algorithm (*i.e.*, Algorithm 1), there can be three possibilities:

- *Parallelize by input events:* Each processing unit is assigned a thread that is responsible for a certain partition or a sub-stream of events from the input. For example, approaches that parallelize based on the `PARTITION BY` attributes of the query implement this possibility [13].
- *Parallelize by state predicates:* Each processing unit is assigned a thread that is responsible for a certain FSM state. This possibility corresponds to the pipelined parallelization approach.
- *Parallelize by runs:* Each processing unit is assigned a thread that is responsible for a certain FSM run instance. This corresponds to the RIP approach that we are proposing in this paper.

Next, we describe two approaches: state- and run-based.

4.1 State-based Parallelization

The idea is to represent each FSM state corresponding to a pattern variable as a processing unit that can run on a dedicated core in parallel. In other words, in the event evaluation sequence, we parallelize the evaluation of each state predicate.

Figure 6 illustrates the basic processing model of our state-based parallelization approach for a pattern that consists of three variables or FSM states (A, B, C). There is exactly one processing unit for each variable in the pattern. Each processing unit has an input queue receiving events of the complete input stream. Except the processing unit of the first pattern variable, all other processing units have a second input queue that contains partial matches from the previous state. The job of a given processing unit (*e.g.*, the one representing variable C) is to evaluate each incoming partial match (*e.g.*, $a_{i-1}b_i$) from the second queue together with the succeeding input event e_{i+1} from the first queue according to all predicates of the corresponding pattern variable (*e.g.*, C). For positive evaluations, the input event is appended to the partial match and the updated partial match is forwarded to the outgoing queue. Note that a processing unit evaluates an event from its first input queue only if it succeeds events of a partial match that is forwarded from its second input queue. This way, redundant evaluations are avoided. In fact, in total there will exactly be the same number of predicate evaluations as in the sequential case illustrated in Table 1.

For variable-length patterns, a processing unit that represents a pattern variable that is optional (*i.e.*, quantified with * or ?) has to forward each incoming partial match to its output queue in addition to the task described above. Furthermore, a processing unit that represents a pattern variable that allows multiple occurrences of itself (*i.e.*, quantified with * or +) has to keep copies of the partial matches that are put into its output queue in an internal buffer. The partial matches in this buffer have to be evaluated together with the subsequent event. Thus, the processing unit has to work off both its input queues and the internal buffer.

While state-based parallelization is intuitive and can be implemented in practice, we see several potential problems with this approach:

- Communication overhead: Partial matches have to be forwarded among consecutive processing units, causing communication overhead among threads running on different cores. This problem would be even more pronounced in variable-length patterns, which are likely to maintain a higher number of partial matches.
- Replication of input stream: The complete input event stream has to be replicated to each processing unit, since each event has the potential to participate in any of the state predicates depending on the partial matches coming from the preceding processing units. Fortunately, not all of them need to be blindly evaluated at each processing unit, since the processing unit can skip the ones that are earlier than the last event of a partial match.
- Query dependence: The number of processing units, and thus the degree of parallelism is restricted by the number of pattern variables or FSM states in the query. Therefore, this approach would not scale beyond a certain number of cores.
- Load imbalance: It is likely that in a pattern matching query certain states may take longer to evaluate their predicates. This would lead to load imbalance across the processing units, such that processing the state predicates with the most loaded node will create a bottleneck. The load imbalance can be due to several factors, including: selectivity of preceding states, number of predicates for a given state, type of predicates for a given state, and the presence of expensive operators such as Kleene Star.

One solution to query dependence and load imbalance problems of the state-based approach could be state replication. Bottleneck states can be good candidates for replication. Load of a state depends on the number of events it processes and the cost of its predicate. Since the first state of the FSM has to evaluate all the incoming events and likely to filter some of them, we enabled replication of it as an optimization to state-based approach. As a result, we implemented both state-based parallelism approach and its optimized version where the first state of the FSM is replicated.

4.2 Run-based Parallelization

As discussed above, the degree of parallelism in state-based approach is dependent on the number of variables in

a query. Loads assigned to each processing unit might be unbalanced; therefore, available resources can not be fully exploited. Moreover, transfer of partial matches between states and thus between processing units seem to have a big overhead. These drawbacks lead to the idea to arrange the evaluations of (event, partial match, state)-triples such that all evaluations of a given run are performed on the same processing unit. We call this approach run-based parallelism. In other words, a partial match remains on a single processing unit until it constitutes a match or it is removed because it cannot end in a match anymore.

In run-based parallelism, each processing unit has an identical task: performing pattern matching on a given sequence of input events. Therefore, the degree of parallelism is independent of the query, unlike state-based approach. In addition to that, since events belonging to the same match stay in a single processing unit, the cost of carrying partial matches around is avoided. However, an important question with this approach is the scheduling of the runs, in other words: "which processing unit will start a run for a given event?". A straight-forward approach could be assigning the processing units in a round-robin manner to the incoming events. For instance, if there are X processing units available, a processing unit i evaluates all runs that start with an event having an id which is congruent to i in modulo X. With the round-robin approach, the input events must be replicated to each processing unit, as pattern matching is sequential by nature such that each processing unit will also need other events. Instead of each incoming event starting a run on another processing unit in a round-robin manner, we took a different approach: we could let n subsequent events start a run on the same processing unit. In other words batches of n events are created and a particular processing unit tries to find all matches that start with events from that batch. Figure 7 depicts the architecture of a pattern matching engine that supports run-based intra-query parallelism (RIP). We can observe that each processing unit has the same task. Besides, each has an input queue for events and an output queue for matches. The event receiver is responsible for forwarding events to the processing units and a match controller collects the matches from the processing units.

With batching, we forward a certain batch only to one processing unit. This way we avoid the replication of events. But how about matches that overlap batches? Beside the batch size n we define another parameter s (size of the shared part of a batch). A processing unit which performs pattern matching on a batch having events with ids $[j; j+n]$ searches for all matches starting between j^{th} and $(j+n-s)^{th}$ events. s events are replicated (i.e., they form the first events of the subsequent batch), so that matches having tail in $(j+n-s; j+n]$ can also be reported. The next question is how to choose a value for the shared part of a batch such that the correctness of the result of a pattern matching query is not violated. For fixed-length patterns with k variables and for variable-length patterns with an upper bound for the match length (`MAXLENGTH` $= k$) we simply define: $s = k - 1$. For variable-length patterns where matches can have any length it is more difficult to find an adequate value. The length of the shared part s has to be chosen such that the probability of an occurrence of a match with length $> s+1$ is almost 0. Furthermore, it is required that $n > s$ and in order to avoid that events belong to more than two batches it should hold

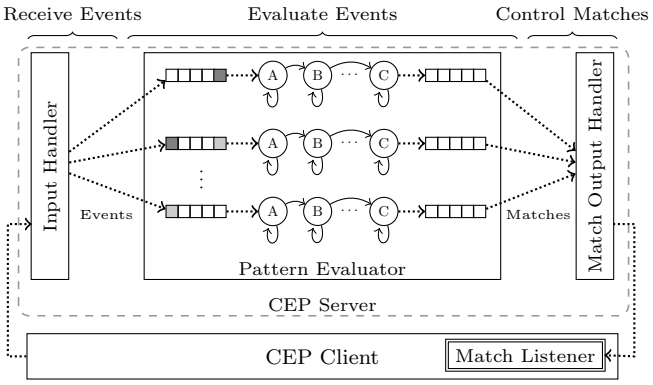

Figure 7: Run-based Parallelization

that $\frac{n}{2} \geq s$. As a result, s events are replicated twice and every event starts exactly one run on exactly one processing unit, but it can contribute to the runs that have started in the previous batch. Shared parts of the batches are shown as shaded boxes in Figure 7.

Due to the reduction of event replication and query independence run-based parallelism approach looks promising. Furthermore, RB+ enables fair scheduling policy. Although different processing units will require different amounts of times, idle processing unit can be assigned to the next batch, and all processing units will be busy at a given time. Thus, different processing times do not create load imbalance across the processing units.

4.3 Discussion

We described two techniques for parallelization of pattern matching: state-based and run-based. In this section, we will analytically compare the performance of these two techniques, focusing on processing time required for event evaluation in each technique.

We start with the processing time of each pattern variable, because in state-based approach each processing unit performs event evaluation for a dedicated pattern variable. The processing time of pattern variable A_i is the number of its incoming events (M_i) multiplied with the time required to evaluate A_i's predicate (p_i) over an event.

$$C_i = M_i * p_i \tag{1}$$

The number of incoming events for pattern variable A_i depends on the selectivity of the pattern variables before A_i. For our purpose, we assume that the number of incoming events (M_i) for each pattern variable is known. Then, processing time of pattern matching in a sequential pattern matching engine is the sum of processing time of its pattern variables. Assume that there are N pattern variables, total processing time of sequential pattern matching is as follows:

$$C = \sum_{i=1}^{N} C_i \tag{2}$$

Under these assumptions, the processing time in the case of state-based parallelism is the processing time of the processing unit with the maximum cost. Remember that state-based parallelism performs pattern matching for each state in parallel.

$$C_{SB} = \max_i C_i \quad (3)$$

Assuming there are X processing units available, in run-based approach the overall processing time of each processing unit is the total cost of sequential processing divided by X. The reason is that in run-based parallelism input is divided into X batches and pattern matching is performed in parallel for each batch.

$$C_{RB} = C/X \quad (4)$$

If we compare the two approaches with an equal number of processing units (N=X), we can conclude that $C_{RB} \leq C_{SB}$. The state-based approach can only achieve the same performance as run-based approach, if each processing unit has the same load (C_1=...=C_i=...=C_N). In all other cases, run-based approach outperforms state-based approach in terms of event evaluation cost.

5. EXPERIMENTS

In this section, we present an experimental study over our parallel pattern matching techniques discussed in this paper. The goal of the experiments is three fold: *(i)* to demonstrate the performance characteristics of each individual parallelization technique in comparison to a sequential execution strategy *(ii)* to study the effects of certain parameters on the performance of corresponding parallelization technique; and finally *(iii)* to quantitatively compare both of the parallelization techniques side-by-side and present the performance superiority of our novel run-based parallelization technique.

5.1 Experimental Setup

We have implemented a prototype pattern matching engine from scratch in Java using Java SE SDK 1.6 and ran all the experiments using the OpenJDK runtime environment. Our prototype contains implementations of the parallel pattern matching techniques, namely state-based and run-based, along with the serial pattern matching implementation as a reference point. Our implementation supports a subset of the `MATCH-RECOGNIZE` query specification as described in Section 2.

In order to replicate a real-world setting, we have conducted all the experiments using a client/server architecture. Basically, the client program is run on a separate machine and sends a continuous stream of events over a Gigabit Ethernet to the server machine which runs our multi-core aware pattern matching engine in isolation. Our implementation uses the Java NIO sockets for efficient communication between the client and the engine where events are sent as byte-streams.

We have used a recent high-end multi-core machine as our experiment platform. The machine comes with 4 processor sockets, each of which is an AMD Opteron 6174 CPU with 12 cores and a clock speed of 2.3 Ghz. The machine uses a Non-Uniform Memory Architecture (NUMA) for memory accesses with a total size of 128 GB main memory. Each core in the socket has private 64 KB L1 and 512 KB L2 caches where the 6 MB L3 cache is shared by all the cores in the socket. We relied on local memory allocations, and due to the first-touch memory allocation policy of Linux, all the allocated memory was local to threads. The machine runs a Debian Linux with kernel version 3.2.16-7.

5.1.1 Datasets

As the workload for our experimental evaluation, we have used two sets of data. The first set of data is generated synthetically to evaluate the performance by changing certain parameters such as selectivity of predicates. The first workload simulates an event stream from a stock market, where each event is a stock trade event as follows:

```
StockTrade(tradeId Long, symbol Char(8), price Float,
          timestamp Long)
```

The size of each event is 24-bytes. Without loss of generality, the queries we have used only use the price attribute for predicate evaluation which is synthetically generated uniformly in the range between 50.0 and 150.0. Using this fixed range serves the purpose of controlling predicate selectivity in queries. In all the experiments, the client generates a fixed number of events ($\approx$1.5 Billion) and pushes them to the server with highest possible rate that the pattern matching engine variant running in the server can handle. In the fastest case, the experiment runs for $\approx$30 minutes and gives sufficient data to observe the performance. In addition, to get reliable results, we start the performance measurements after first $\approx$10 % of the events are sent and stop it before the final 1 % of the events are sent.

As the second workload, we have used real-world stock market data to assess the performance under realistic conditions. As a result, experiments with this workload validate the applicability and performance of our techniques. Data we have used comes from a snapshot of real stock exchange trade and quote (NYSE TAQ [5]) collected from several stock exchanges in the U.S.A. over 3 days between January 3, 2006 and January 5, 2006. In order to use a higher volume event stream, we have explicitly used the quote data which consists of bid/ask prices given by customers as shown below:

```
StockQuote(symbol Char(8), timestamp Int, bid Float,
          bidSize Int, offer Float, offerSize Int)
```

5.1.2 Queries

In our experimental evaluation, we have used a broad number of queries with diverse properties. Here, we briefly summarize the properties of different class of queries that we considered. The first dimension that queries differ is the pattern variables. If a query contains just singleton variables, then it is called a **fixed-length** query (*i.e.*, Q5). Otherwise, it is called a **variable-length** query (*i.e.*, Q1). The second dimension that queries differ is the predicates. If a query contains predicate evaluation against only static values, such as `A.price > 100`, then it is called **static-predicate** query (*i.e.*, Q11). Otherwise, it contains predicates with correlation among values of two different events and/or previous values of those values (*i.e.*, `A.price > B.price` or `A.price > PREV(A.price)`). This type of queries are called **dynamic-predicate** queries. For the interested reader, we provide all the queries used in this paper in Appendix A.

5.1.3 Implementation Variants

We have experimented with the following variants of our pattern matching engine:

- **SEQ:** Sequential (*i.e.* single-threaded) pattern matching engine.
- **SB:** State-based parallel pattern matching engine that uses pipelining.

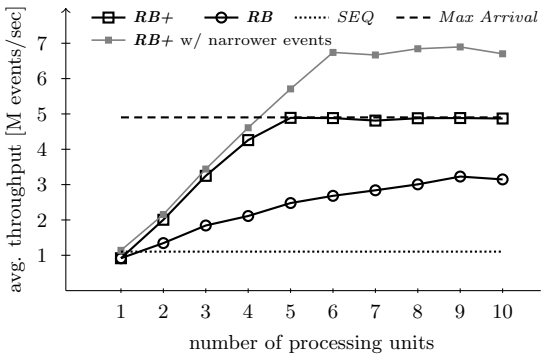

Figure 8: Comparison of run-based techniques

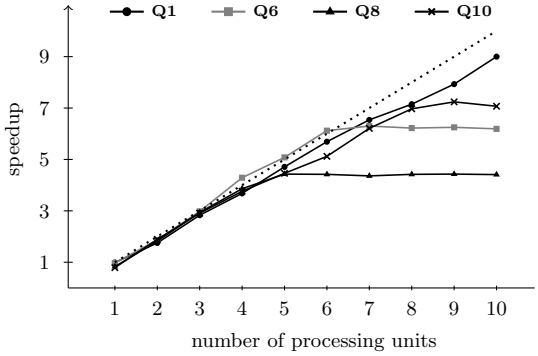

Figure 9: Speedup of RB+ to SEQ for different queries

- **SB+:** Optimized version of SB where the first state is duplicated to overcome its certain limitations.
- **RB:** Run-based parallel pattern matching engine where events are distributed in round-robin to all processing units.
- **RB+:** Optimized version of RB. Batches of events of size $n = 5000$ are distributed to processing units. Two subsequent batches, accordingly processing units, have a shared part of size s. A processing unit starts a run for the first $(n - s)$ events in a batch.

In all our experiments, each processing unit is a separate thread.

5.2 Performance of Run-Based Techniques

In this section, we compare the performance of different run-based techniques, **RB** and **RB+** using the synthetic workload and query 8 (**Q8**). The throughput achieved by these techniques with varying number of processing units are shown in Figure 8. To a certain degree, increasing number of processing units (*i.e.* threads) results in a higher throughput. As shown in Figure 8, our optimized version of the run-based, **RB+** scales much faster than **RB** up to a certain point. Surprisingly, the performance of **RB+** stays limited from there on. We have investigated the reason for this result and found out that the execution quickly becomes network-bound in our setup. Using events of size 24-bytes, the Gigabit Ethernet allows only a maximum of $\approx$5 Million events per second connection bandwidth between our client and server. As a result, input path becomes the bottleneck of our evaluation and we are not able to show scalability of our system beyond 10 processing units for this query. In order to prove this situation, we have run the same experiment with narrower events (*i.e.*, 16-byte events by dropping attributes not used in predicates). As can be seen in Figure 8, the performance with narrower events scales better but at some point network becomes the bottleneck again.

On the other hand, although **RB** seems to scale, it does not reach this upper bound even with 10 processing units. The main problem in this technique is the high cost of checking/starting of runs for each event. Based on this experiment, the batching idea clearly shows its benefit over plain **RB**. In the rest of the experiments, we will use **RB+** as it is superior to **RB**.

5.3 Evaluation of Optimized Run-Based Parallelism

In this section, we present the results achieved for our optimized run-based parallelization technique **RB+** with the synthetic workload. Figure 9 shows the speedups attained for four different queries. The four different queries shown in Figure 9 are a representative set of the ones from the entire set of queries we have experimented with, hence only they are shown. We also show the results up to 10 processing units due to the network bandwidth limitation issue as mentioned in Section 5.2. Besides that, our results show a general trend of linear scaling behavior for most of the queries. However, for queries 6 and 8, the maximum speedup is achieved at a low number of processing units and the performance does not further improve. This is mainly due to the reason that per event computation cost of these queries are rather cheaper in comparison to the others. Hence, the input path of the processing, namely the network, becomes a bottleneck earlier in this case. Overall, these experiments show that **RB+** scales well as long as the network is not a limitation. Additionally, it is highly effective independent of the query (*i.e.* run-based parallelism is **query-independent**).

5.4 Effect of Shared Events between Batches

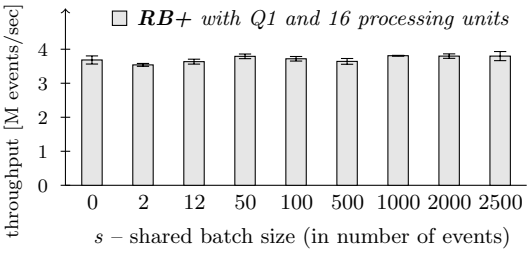

Figure 10: Effect of shared batch size in RB+

The optimized run-based technique **RB+** has a configuration parameter s which is the number of shared events between consecutive batches of different processing units. In this experiment, we investigated whether s has an impact on achievable throughput. Figure 10 shows the results of the experiment where we varied s for evaluation of query 1 (**Q1**) using 16 processing units. Our results mainly show that the size of shared part between batches does not effect

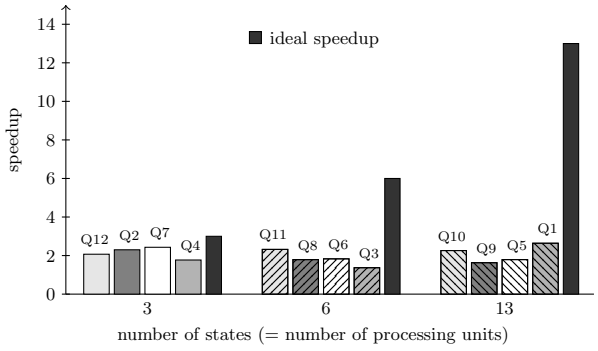

Figure 11: Performance of SB

	Throughput (M Events/sec)	Speedup	# Processing Units
SB	3.34 M	2.32	6
SB+	4.31 M	2.99	7
SEQ	1.44 M	1	1

Table 2: Optimized state-based parallelism (SB+)

the achievable throughput (similar results are observed for different queries).

5.5 Performance of State-Based Parallelism

In this section, we present the results achieved for our state-based parallelization technique **SB** with the synthetic workload. Figure 11 presents the throughput speedups achieved for a representative set of queries. The number of finite automata states in each query corresponds to the parallelism level used for that query (*i.e.,* the number of processing units). In an ideal case, we would normally expect the speedup achieved for a query to be in line with the number of states. Unfortunately, the results in Figure 11 do not correspond to the expectations. For queries with three state variables, the speedup achieved is close to what we would expect. However, as number of states in a query increase, the speedup achieved does not increase further. The reason for this observation is as follows. State-based parallelism mainly builds up on pipelined parallelism. In the pipelined parallelism model, the slowest stage in the pipeline determines the maximum throughput that can be achieved. In a pattern matching query, usually different states in a query have different per event costs due to the probability of matches and selectivity of predicates. As a result, unless a query is artificially constructed to have balanced set of states in terms of execution time, it is not possible to achieve a parallelism speedup proportional to the number of states.

In general, state-based parallelism helps to improve the performance to a certain degree but does not scale with the number of states. As a result, state-based approach is rather **query dependent** where the number of states and the distribution of the workload among the states determine the achievable speedup.

5.6 Evaluation of Optimized State-Based Parallelism

In state-based technique **SB**, the first state of the automata might constitute the bottleneck when it has a high predicate selectivity with expensive predicates. To reduce its impact on the overall performance, we have implemented an optimized version of the state-based technique called **SB+**. Basically, **SB+** duplicates the first state. For queries where the first state is a bottleneck, this optimization in **SB+** becomes marginally effective. Table 2 shows the results of an experiment for such a query (**Q11**). However, the generality of this optimization is also query dependent and rather limited. Due to this reason, it is not possible to apply this optimization in all the queries of our experiments. Thus, we use **SB** in the rest of the experiments.

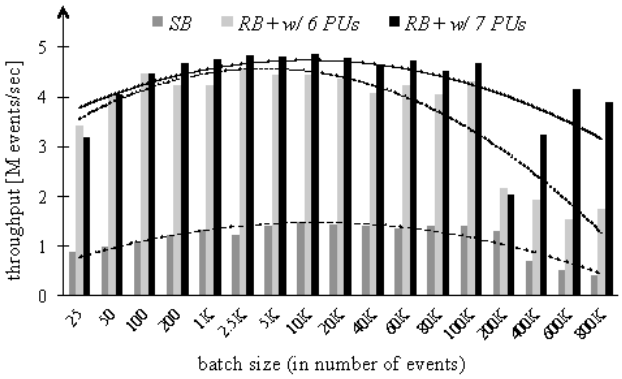

Figure 12: Impact of event distribution batch size

5.7 Impact of Batching on Event Distribution

In this section, we investigate the impact of batching on event distribution to processing units. The discussion applies to both of the parallelization techniques and should not be confused with the batching in run-based parallelism.

In all the implementations, the events and partial matches are collected into batches before they are transferred among processing units. The other extreme alternative is event-by-event distribution, which we found out to be a severe performance limitation. The batch-based distribution technique avoids the communication overheads among processing units. However, the batch-size of events for distribution becomes a configuration parameter. In the experiment shown in Figure 12, we have varied the event distribution batch size to observe its impact. First of all, we have observed a similar behavior for different queries (*i.e.* **query independent**) and we present the results for query 7 (**Q7**). Secondly, in general there is a slight trade-off between a small batch size and a large batch size. Usually, there is a fixed cost for creating a batch and putting it into a queue. With small batch sizes, the frequency of accesses to the queue becomes higher, which means increased synchronization among processing units. On the other hand, a larger batch size means that an event will reside in the memory for a longer time and in case of Java, would stress the garbage collector more. All in all, regardless of the query, a batch size between 5,000 and 10,000 achieves the best performance in the existence of these trade-offs.

5.8 Comparison of Run-Based and State-Based Techniques

In this section, we quantitatively compare our best run-based parallel pattern matching engine **RB+** and the state-based parallel pattern matching engine **SB**. State-based parallelism works best when the execution time of different

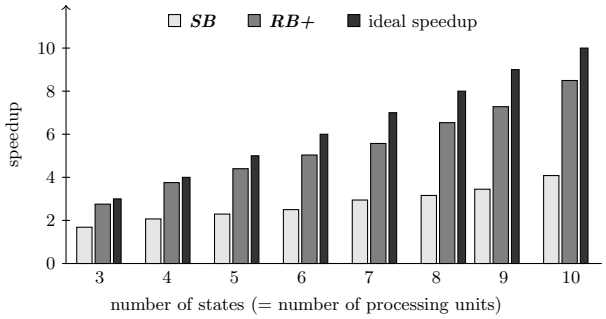

Figure 13: Comparison of parallelization techniques with a balanced query

states are balanced. Although such queries are not common in real-world applications, we also include measurements with such a query in our results. The results of the comparison are shown in Figure 13. Essentially, the results show that even for balanced queries, run-based techniques outperform state-based techniques.

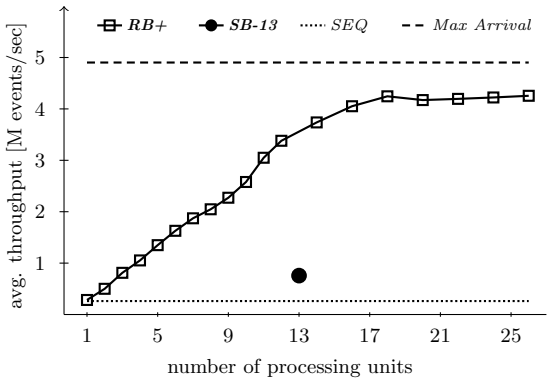

Figure 14: Comparison of parallelization techniques

In order to compare the techniques with a more realistic scenario, we have repeated the comparison experiment using the classic "head and shoulders" pattern. The query (**Q1**) is a rather costly query with 13 pattern variables. Figure 14 shows the achievable throughput for each technique. Note that state-based technique **SB** must use 13 fixed processing units for this query. Despite having sufficient number of states, the state-based parallel pattern matching engine can only achieve a speedup of 2.64. Using the same number of processing units, **RB+** achieves a much better performance. Furthermore, as shown previously, our run-based technique **RB+** has a scaling behavior (*i.e.* ≈15X) up to the point where the network becomes a bottleneck.

5.9 Performance with a Real-World Workload

In this section, we present the performance of our run-based parallelism technique under a real-world workload scenario. The data we used comes from NYSE TAQ [5] as described in Section 5.1.1. We have used the query 1 (**Q1**) which is a classic "head and shoulders" pattern query, and has been discussed throughout the paper. The results of this experiment are shown in Figure 15. The results clearly indicate the performance and the scalability of our run-based

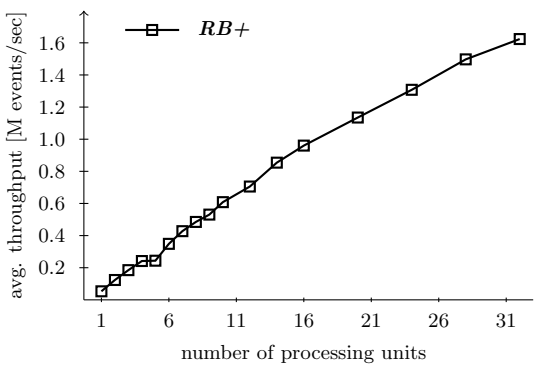

Figure 15: Performance of RB+ with a real-world workload

parallelization technique **RB+**. The performance of our technique is better pronounced under this setting for the following reasons: *(i) cost of query:* even the serial evaluation of this query on the real-world data is expensive, which makes it computation-bound rather than network-bound. As an indication, our serial implementation can handle only up to 68K events per second. *(ii) nature of real-world data:* real-world market data contains a lot of repetitions in values (*i.e.*, in price) due to clustered market activity and once something matches to our pattern, it is likely that many things will match at once (which is not the case with synthetic data). Overall, these reasons make our parallelization technique shine under the real-world dataset and it shows almost a perfect linear scalability.

5.10 Summary of Experiments

In the experiments, we showed the performance characteristics of our parallel pattern matching techniques. The results showed that state-based parallelism is usually **query dependent** and ideal speedups from parallelism could not be achieved easily. On the other hand, we showed that run-based parallelism is more robust to different queries and is **query independent**. Also our quantitative comparison of the techniques concluded that run-based parallelization achieves a superior performance to that of state-based parallelism.

6. RELATED WORK

Recently, there have been many systems proposed for sequential pattern matching on data streams [2, 7, 10, 11, 14, 12]. Each of these systems follows a different approach to optimize high cost of sequential pattern matching. Since our focus is parallelization of pattern matching processing, our work is orthogonal to them. Except ZStream [14], all other systems use some variant of a non-deterministic FSM to detect patterns. ZStream uses tree-based plans to model pattern matching queries. For a given pattern, there exist multiple plans with different costs [14]. ZStream presents a cost-based adaptive optimization where it is possible to adjust the order of evaluations on-the-fly [14]. ZStream can benefit from run-based parallelism, such that multiple query plans can run in parallel. On the other hand, SASE+[7] proposed a shared match buffer to share partial matches across different runs efficiently. Similarly, SASE+ can also

use run-based parallelism and use its shared match buffer to maintain partial matches on a single processing unit.

There has been also a few recent works focusing on parallelization of pattern matching processing. Pattern matching can be seen as a stateful operator in a general-purpose streaming system [13]. Hirzel et al. [13] exploit the partitioning constructs provided by the queries (*i.e.*, `PARTITION BY`). Despite being an effective solution, this approach becomes insufficient at times when queries do not contain such constructs. Wu et al. [17] propose a parallelization framework for stateful stream processing operators. Their model splits events in round-robin to different replicas of an operator that are assumed to have an access to a shared state. Their assumption is not feasible for pattern matching, since in this case shared state would consist of set of pointers to an FSM. Additionally, events must be evaluated in a determined order for each active state. Therefore, our approach considers creating parallel tasks from an operator as much independent as possible.

Schneider et al. [15] also consider intra-operator parallelism through data-partitioning. They introduce a compiler and a run time system that automatically extract data parallelism from queries. Additionally, they introduce the concept of safety conditions for automatic parallelization. Given an operator, if one can assign a key to each partition out of the attributes of events, then state of an operator can be partitioned. In this manner, the approach becomes effective without requiring any shared state among replicas.

In a similar effort, Brenna et al. [8] take a different approach and distribute an event processing system across a cluster of machines. They implemented a distributed event pattern matching system based on Cayuga. As a first step, they also apply data parallelism. In contrast to other related work, their focus is also running multiple queries in parallel. In their FSM-based evaluation approach, FSM is decomposed into separate states running on different machines. In this regard, pipelined parallelism of states is achieved. Our state-based parallelization approach also works similarly, where the pipelining is achieved within a machine.

In this paper, our focus is parallelization within a single partition of an event stream. Our approach is complimentary to key-based (*i.e.*, `PARTITION-BY`) parallelization techniques and can be further applied for fine-granular parallelization, especially on multi-core CPUs. We consider two main techniques, run-based and state-based. Lastly, we present an experimental evaluation over both approaches and conclude with the superiority of our run-based parallelization technique.

7. CONCLUSIONS

In this paper, we investigated parallel pattern matching techniques for scaling a CEP query on multi-core architectures. We proposed RIP - a run-based intra-query parallelization approach that achieves linear scale-up, while being query-independent and skew-tolerant. RIP complements previous partitioned parallelism approaches and outperforms its pipelined parallelism alternative. Our focus in this work was throughput, we would like consider other performance criteria such as response time in the future. Further future work directions include analyzing and handling network-bound use cases, extending RIP to be fault-tolerant and to apply in cluster settings, and investigating inter-query parallelism for CEP.

8. REFERENCES

[1] Day trading technical analysis. `http://www.daytradingcoach.com/daytrading-technicalanalysis-course.htm`. Accessed: 03/11/2012.

[2] Esper. `http://www.espertech.com`. Accessed: 02/12/2012.

[3] Head and shoulders. `http://www.chartpatterns.com/headandshoulders.htm`. Accessed: 03/02/2013.

[4] Head and shoulders (chart pattern). `http://en.wikipedia.org/wiki/Head_and_shoulders_(chart_pattern)`. Accessed: 10/11/2012.

[5] NYSE Data Solutions. `http://www.nyxdata.com/nysedata/`.

[6] Oracle CEP. `http://www.oracle.com/technetwork/middleware/complex-event-processing/index.html`.

[7] J. Agrawal et al. Efficient Pattern Matching over Event Streams. In *ACM SIGMOD Conference*, Vancouver, Canada, 2008.

[8] L. Brenna et al. Distributed Event Stream Processing with Non-deterministic Finite Automata. In *ACM DEBS Conference*, Nashville, Tennessee, July 2009.

[9] G. Cugola et al. Tesla: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[10] A. Demers et al. Cayuga: A General Purpose Event Monitoring System. In *CIDR Conference*, Asilomar, CA, 2007.

[11] N. Dindar et al. DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events (Demo). In *ACM SIGMOD Conference*, Providence, RI, 2009.

[12] N. Dindar et al. Efficiently Correlating Complex Events over Live and Archived Data Streams. In *ACM International Conference on Distributed Event-Based Systems (DEBS'11)*, New York, NY, USA, July 2011.

[13] M. Hirzel. Partition and Compose: Parallel Complex Event Processing. In *ACM DEBS Conference*, Berlin, Germany, July 2012.

[14] Y. Mei et al. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *ACM SIGMOD Conference*, Providence, RI, June 2009.

[15] S. Schneider et al. Auto-Parallelizing Stateful Distributed Streaming Applications. In *ACM PACT Conference*, Minneapolis, MN, September 2012.

[16] E. Wu et al. High-Performance Complex Event Processing over Streams. In *ACM SIGMOD Conference*, Chicago, IL, June 2006.

[17] S. Wu et al. Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should you and How much? In *ACM DEBS Conference*, Berlin, Germany, July 2012.

[18] F. Zemke et al. Pattern Matching in Sequences of Rows. Technical Report ANSI Standard Proposal, 2007.

APPENDIX

A. QUERIES

This appendix shows the pattern declarations of all the queries used in our paper.

Query 1 (Q1)

```
PATTERN
(A B* C D* E F* G H* I J* K L* M
    )
DEFINE
 B AS (B.price ≥ PREV(B.price)),
 C AS (C.price > PREV(C.price)),
 D AS (D.price ≤ PREV(D.price)),
 E AS (E.price < PREV(E.price)
       AND E.price > A.price),
 F AS (F.price ≥ PREV(F.price)),
 G AS (G.price ≥ PREV(G.price))
       AND G.price > C.price),
 H AS (H.price ≤ PREV(H.price)),
 I AS (I.price < PREV(I.price)
       AND I.price > A.price),
 J AS (J.price ≥ PREV(J.price)),
 K AS (K.price > PREV(K.price)
       AND K.price < G.price),
 L AS (L.price ≤ PREV(L.price),
 M AS (M.price ≤ PREV(M.price)
       AND M.price < E.price
       AND M.price < I.price)
```

Query 2 (Q2)

```
PATTERN
    (A B+ C)
 DEFINE
    A AS (A.price < 70),
    B AS (B.price > 80
          AND B.price < 120),
    C AS (C.price > 130)
```

Query 3 (Q3)

```
PATTERN
   (A B+ C+ D+ E+ F)
 DEFINE
   B AS (B.price > PREV(B.price)
         AND B.price > A.price),
   C AS (C.price < PREV(C.price)
         AND C.price > A.price),
   D AS (D.price > PREV(D.price)
         AND D.price > A.price),
   E AS (E.price < PREV(E.price)
         AND E.price > A.price),
   F AS (F.price < A.price)
```

Query 4 (Q4)

```
PATTERN
   (A B* C)
 DEFINE
   A AS (A.price < 70),
   B AS (B.price > PREV(B.price)
         AND B.price > A.price),
   C AS (C.price > PREV(C.price)
         AND C.price > 130)
```

Query 5 (Q5)

```
PATTERN
  (A B C D E F G H I J K L M)
DEFINE
 B AS (B.price > PREV(B.price)),
 C AS (C.price > PREV(C.price)),
 D AS (D.price < PREV(D.price)),
 E AS (E.price < PREV(E.price)
       AND E.price > A.price),
 F AS (F.price > PREV(F.price)),
 G AS (G.price > PREV(G.price)
       AND G.price > C.price),
 H AS (H.price < PREV(H.price)),
 I AS (I.price < PREV(I.price)
       AND I.price > A.price),
 J AS (J.price > PREV(J.price),
 K AS (K.price > PREV(K.price)
       AND K.price < G.price),
 L AS (L.price < PREV(L.price)),
 M AS (M.price < PREV(M.price)
       AND M.price < E.price
       AND M.price < I.price)
```

Query 6 (Q6)

```
PATTERN
   (A B C D E F)
 DEFINE
   B AS (B.price > PREV(B.price)
         AND B.price > A.price),
   C AS (C.price < PREV(C.price)
         AND (C.price > A.price),
   D AS (D.price > PREV(D.price)
         AND D.price > A.price),
   E AS (E.price < PREV(E.price)
         AND E.price > A.price),
   F AS (F.price < A.price)
```

Query 7 (Q7)

```
PATTERN
    (A B C)
 DEFINE
    A AS (A.price < 70),
    B AS (B.price > A.price),
    C AS (C.price < B.price
          AND C.price < A.price)
```

Query 8 (Q8)

```
PATTERN
   (A B* C+ D+ E+ F)
 DEFINE
   A AS (A.price < 80),
   B AS (B.price > 80
         AND B.price < 120),
   C AS (C.price < 80),
   D AS (D.price > 80
         AND D.price < 120),
   E AS (E.price > 120),
   F AS (F.price > 130)
```

Query 9 (Q9)

```
PATTERN
(A B+ C D+ E F+ G H+ I J+ K L+ M)
DEFINE
   A AS (A.price < 70),
   B AS (B.price > 70
         AND B.price < 130),
   C AS (C.price > 130,
   D AS (B.price > 70
         AND D.price < 130),
   E AS (E.price < 70),
   F AS (F.price > 70
         AND F.price < 130),
   G AS (G.price > 130),
   H AS (H.price > 70
         AND H.price < 130),
   I AS (I.price < 70),
   J AS (J.price > 70
         AND J.price < 130),
   K AS (K.price > 130),
   L AS (L.price > 70
         AND L.price < 130),
  M AS (M.price < 70)
```

Query 10 (Q10)

```
PATTERN
    (A B C D E F G H I J K L M)
 DEFINE
   D AS (D.price < 100),
   E AS (E.price > 100),
   F AS (F.price < 100
         AND F.price > 60),
   G AS (G.price > 100
         AND G.price < 140),
   H AS (H.price < 100
         AND H.price > 70),
   I AS (I.price > 100
         AND I.price < 130),
   J AS (J.price < 100
         AND J.price > 80),
   K AS (K.price > 100
         AND K.price < 120),
   L AS (L.price < 120
         AND L.price > 80),
   M AS (M.price > 80
         AND M.price < 120)
```

Query 11 (Q11)

```
PATTERN
    (A B C D E F)
 DEFINE
    A AS (A.price < 80),
    B AS (B.price < 90
          AND B.price > 60),
    C AS (C.price < 100
          AND C.price > 70),
    D AS (D.price < 110
          AND D.price > 80),
    E AS (E.price < 120
          AND E.price > 90),
    F AS (F.price < 130
          AND F.price > 100)
```

Query 12 (Q12)

```
PATTERN
    (A B C)
 DEFINE
    A AS (A.price < 90),
    B AS (B.price > 110),
    C AS (C.price < 90)
```