HYPERSONIC: A Hybrid Parallelization Approach for Scalable Complex Event Processing

ABSTRACT

The ability to promptly and efficiently detect arbitrarily complex patterns in massive real-time data streams is a crucial requirement in many modern applications. The ever-growing scale of these applications and the sophistication of the patterns involved makes it imperative to employ advanced solutions that can optimize pattern detection. One of the most prominent and well-established ways to achieve the above goal is to apply complex event processing (CEP) in a parallel manner, using a multi-core machine and/or a distributed environment. However, the inherent tightly coupled nature of CEP severely limits the scalability of the parallelization methods currently available.

In this paper, we introduce a novel parallelization mechanism for efficient complex event processing over data streams. This mechanism is based on a hybrid two-tier model combining multiple layers of parallelism. By employing a fine-grained load balancing model, this multi-layered approach leads to a substantial increase in event detection throughput, while at the same time reducing the latency and the memory consumption. An extensive experimental evaluation on multiple real-life datasets shows that our approach consistently outperforms state-of-the-art CEP parallelization methods by a factor of two to three orders of magnitude.

CCS CONCEPTS

• **Information systems** → **Stream management**; *Query optimization*; • **Computing methodologies** → Parallel algorithms; Distributed algorithms.

KEYWORDS

Complex Event Processing, Parallel Query Processing, Distributed Query Processing, Query Optimization

ACM Reference Format:

. 2022. HYPERSONIC: A Hybrid Parallelization Approach for Scalable Complex Event Processing. In *SIGMOD '22: ACM Conference on Management Of Data, June 12–17, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Complex event processing (CEP) is a leading technology for robust and high-performance real-time detection of arbitrarily complex patterns in massive data streams [26, 27, 32]. It is widely employed in many areas where the data is continuously generated in a streaming manner and needs to be promptly and efficiently analyzed on-the-fly. Online finance [29], credit card fraud detection [66], sensor networks [36], healthcare industry [15], and IoT applications [77] are among the many examples.

CEP engines treat the data items that make up the input streams as *primitive events* arriving from event sources. As new primitive events are observed, they are assembled into higher-level *complex events* that match the user-defined *patterns*. Detecting complex events is performed by collecting primitive events and incrementally combining them into *partial matches*. As more events are added to a partial match, a full pattern match is eventually formed and reported. The loose order of constructing and extending partial matches is defined by a graph-based structure, typically an automaton or a tree, composed of a set of *states*. Each state represents some intermediate stage of pattern detection. Figure 1 (and 1(b) in particular) illustrates this principle.

As discussed by multiple authors [7, 43, 52], the processing time, latency, and resource consumption of the CEP execution grows exponentially with the length of the pattern being detected. The main factor contributing to this growth is the need to explicitly examine a large number of sets of events to determine whether they comprise valid pattern matches, which leads to a crucial performance bottleneck. The situation is exacerbated by the tight real-time constraints imposed on these systems, as well as by a common requirement to simultaneously process multiple patterns and streams. Therefore, advanced optimization techniques are essential for achieving an acceptable quality of service.

Parallelizing CEP evaluation flows is one of the most prominent avenues for improving the performance of event processing applications. Various techniques for allocating the workload of a CEP system to multiple execution units and managing their parallel execution have been proposed, addressing multi-core and fully distributed scenarios. These solutions can be roughly divided into two separate categories: data-parallel and state-parallel methods.

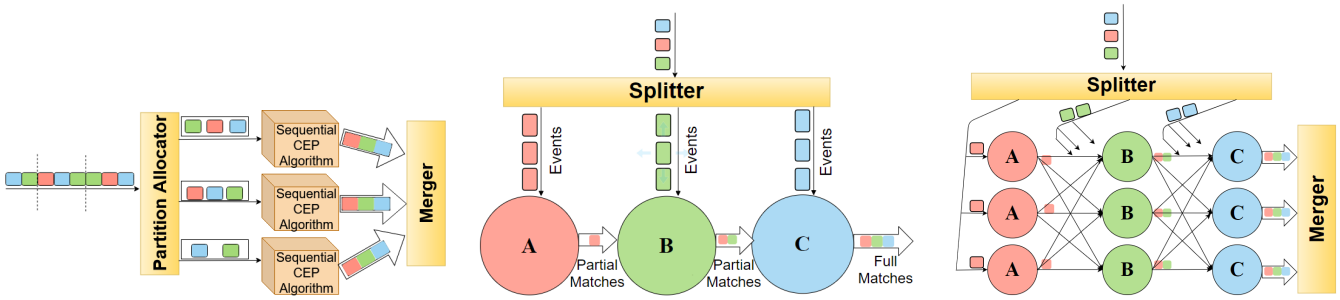
Data-parallel approaches [12, 37, 49] operate by splitting the input data stream into different partitions according to predefined criteria and routing each partition to a dedicated execution unit; this unit may be a thread, a process, or a separate machine. Each unit then applies the same pattern matching process on the data it receives, and the resulting pattern matches are merged and jointly returned to the end users. Figure 1(a) presents an example of a parallel CEP system architecture implemented according to this paradigm. While this scheme was proven highly efficient in many cases, its inherent limitation lies in the difficulty of designing a good partitioning scheme. Since any subset of data items can represent a pattern match, at least a fraction of the sub-streams must be duplicated to multiple units to avoid missing results and to guarantee detection correctness. It is also exceptionally hard to achieve fine-grained load balancing in presence of data skew.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456



(a) Data parallelism **(b)** State parallelism **(c)** Hybrid parallelism

Figure 1: Parallelism classes: (a) input is split into partitions and handled separately by each execution unit; (b) each building block is represented by a dedicated execution unit that receives a substream of events and performs a well-defined sequential task; (c) a two-layer approach combining both data parallelism and state parallelism.

The second category of CEP parallelization methods is known as *state-parallel* [12, 25, 71]. This approach assigns a dedicated execution unit to each building block of the pattern detection model, making it exclusively responsible for some functional part of the sequential pattern matching algorithm. The units are arranged according to a predefined topology and the intermediate results of the pattern match construction process are passed between them. An example is shown in Figure 1(b). This parallelization scheme avoids the data stream duplication problem that plagues data-parallel methods. However, it imposes a hard limit on the degree of parallelism as the number of execution units is bounded by the number of states.

In this paper, we propose HYPERSONIC: a HYbrid ParallElization appRoach for Scalable cOmpLex eveNt processIng appliCations. HYPERSONIC implements a new paradigm for parallelizing CEP applications, which we refer to as a *hybrid-parallel approach*. In a hybrid-parallel system, the execution units are organized in two layers, and the workload distribution proceeds in two stages. First, a state-parallel procedure allocates a set of execution units to each state according to its expected load. Second, a data-parallel routine is applied within each state to share the work performed by this state between the individual units. This process is repeated continuously during the system run. In this way, the system can dynamically adapt to the ever-changing data arrival rates, system properties, and resource availability. Figure 1(c) illustrates this scheme.

By providing two distinct layers of parallelism, our approach combines the strengths of data-parallel and state-parallel solutions while overcoming their limitations. Unlike that of a pure state-parallel system, the degree of parallelism in HYPERSONIC is unbounded. In addition, no duplicate data transmission is required: the *outer parallelization layer* mimics the evaluation flow of the state-parallel approach and only passes the data once between adjacent states, while the *inner parallelization layer* avoids the need for an explicit partitioning scheme by utilizing shared memory between execution units. This latter communication mechanism is designed in a way allowing an extension to a fully distributed share-nothing environment. The two-tier dynamic load balancing scheme of HYPERSONIC ensures efficient allocation of execution units to states and of the input data to execution units based on up-to-date data characteristics and system load.

Rather than merely constituting a load balancing scheme, HYPERSONIC is an end-to-end system for efficient and scalable parallel execution of complex event processing workloads. As we elaborate later on, our solution successfully tackles a variety of CEP-specific parallelization challenges, such as avoiding data duplication, enabling data sharing between the execution units, and supporting complex operators.

The contributions of this paper can be summarized as follows:

- A novel *hybrid-parallel* approach for efficiently distributing CEP workloads between multiple execution units.
- A two-level load balancing scheme based on the above system architecture, and a thorough analysis of its performance.
- Practical extensions to the basic hybrid-parallel method allowing us to further improve the performance and the resource utilization of pattern detection.
- An extensive experimental evaluation of our method, demonstrating its superiority over state-of-the-art CEP parallelization mechanisms as well as a superlinear speedup over the sequential baseline.

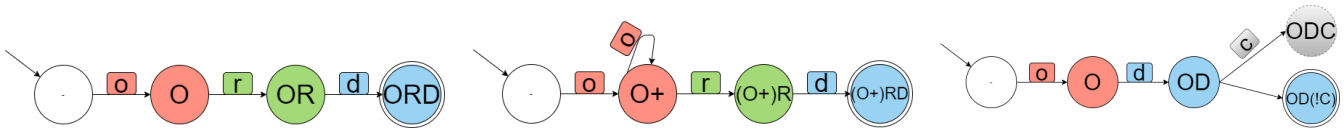
The remainder of this paper is organized as follows. Section 2 presents the necessary background and formally defines the targeted problem. The core design of HYPERSONIC is described in Section 3 with important extensions covered in Section 4. We report the results of our experimental study in Section 5. Section 6 discusses the related work and Section 7 concludes the paper.

2 BACKGROUND AND TERMINOLOGY

2.1 Event and Pattern Model

The functionality of a CEP system revolves around the notion of an *event*. A *primitive event* $e = \{T, \{a_1, a_2, ..., a_n\}, ts\}$ is an indication that a single action of interest happened at a specific point in time. It is associated with a single *event type* T, contains a set of *attributes* $\{a_i\}_{i=1}^{n}$ and the event occurrence timestamp ts. An input *event stream* $I = \{e_1, e_2, ..\}$ is a sequence of temporally ordered events where every two events e_i, e_j with timestamps ts_i, ts_j respectively satisfy $t_i < t_j$ if $i < j$ [35].

A CEP system receives a large, potentially infinite event stream arriving from one or more sources and is tasked with detecting



(a) Sequence pattern **(b)** Kleene closure pattern **(c)** Negation pattern

Figure 2: Example NFAs: (a) a sequence of an order, a removal from storage, and a delivery of an item; (b) a sequence of one or more duplicate orders, a removal, and a delivery; (c) a sequence of an order and a delivery with no subsequent order cancellation.

complex events - combinations of the primitive events conforming to the user-defined *patterns*.

A pattern is an expression describing the situation of interest that a user is willing to identify. While its precise format depends on the declarative language in use [11, 16, 23, 72], a pattern commonly consists of the *structure* defining how to combine the participating events, the Boolean *conditions* to be satisfied by a pattern match, and the *time window* W that sets the time frame within which the complex event is to be detected. The pattern structure is an expression over a set of *operators*, including sequence (SEQ), conjunction (AND), disjunction (OR), negation (NOT), and Kleene closure (KL).

Formally, a *flat pattern* $FP = \{E, O, W, C\}$ is defined by a time window W, a set $E = \{E_1, E_2, ..E_m\}$ of event types, an operator O describing the desired relation between the event types and a set of conditions on the event types $C = \{C_1, C_2, C_k\}$. A set of primitive events $m = \{e_1, e_2, ..., e_l\}$ satisfying the operator semantics and the conditions of a pattern FP is a *match* of FP. Nested patterns could be created by utilizing multiple operators to form an arbitrarily deep operator tree (that is, the pattern structure). A *nested pattern* $NP = \{E, S, W, C\}$ is formally defined similarly to a flat pattern, with the operator O replaced with an operator structure S. Interested readers are referred to [31] for more details.

As an example, consider a warehouse serving customer orders. The RFID tag of a served item is scanned during certain actions, such as removing an item from storage, loading it onto a forklift, and registering it as ready for delivery. We are interested in tracking items ordered in the last hour and ready to be delivered.

Under the above notations, we will define each item action as a primitive event with the event types including O (ordering an item), R (removing from storage), D (delivery), and C (cancellation of an order), among others. Event attributes could include the item ID, the name of an employee performing the action, and the details of the customer that ordered the item. A CEP pattern representing a recent order ready to be shipped could then be formulated as *"detect a sequence of three events of types O, R, and D respectively within one hour such that the item ID of all events is the same"*. Applying this pattern on an event stream of log entries listing the registered actions would yield a list of triplets comprising the complex events.

An additional part of the pattern definition is a *selection strategy*, specifying how events are selected and consumed from the input stream. There are several widely used selection strategies [34], including *Strict contiguity* and *Skip till next match*, which differ by the restrictions they impose on the event selection for a match. In this paper, we exclusively assume the *skip-till-any-match* strategy which poses no restrictions on event inclusion in a match. It was shown in [74] that the runtime complexity of this policy if combined with Kleene closure is exponential in relation to the time window and as such the most challenging to support from the performance standpoint.

2.2 CEP Evaluation Mechanism

CEP systems detect complex events satisfying the predefined patterns by incrementally combining incoming primitive events into *partial matches* that eventually grow into *full pattern matches* and are returned to the end users. This process is handled by an *evaluation mechanism*, a graph-based data structure defining either strict or loose order in which events of different types are processed. Widely employed evaluation mechanisms include NFAs (nondeterministic finite automata) [6, 43, 72, 74], evaluation trees [41, 52], and EPNs (event processing networks) [31, 58].

Figure 2(a) depicts an example of a NFA detecting the sequence pattern from Section 2.1. Each state represents a particular step during the pattern matching process, with every traversed edge causing a new partial match creation. The transition from the initial state is performed upon an arrival of an event of type O, prompting the creation of a new partial match containing this event. Consequently, the outgoing transition to state OR is traversed when an event of type R arrives satisfying the mutual condition with some previously obtained event of type O. The traversal and the creation of a new partial match within state R takes place for each valid (O, R) pair. Finally, the outgoing transition to the accepting state ORD is traversed by each (O, R, D) triplet representing a full match.

In [6, 42], the authors have shown how any non-nested CEP pattern can be translated into a chain-based automaton. Figures 2(b) and 2(c) illustrate NFAs for patterns containing a Kleene closure and a negation operator, respectively. For nested patterns, extensions combining multiple sub-automata have been proposed [47].

While the ideas presented in this paper could be applied to a CEP system utilizing any evaluation mechanism, for ease of presentation we will solely focus on NFAs from now on.

2.3 Problem Definition

The computational cost of CEP is inherently exponential in the length and the complexity of the pattern. In the example depicted in Figure 2(a), if the NFA creates 100 partial matches corresponding to 100 past O events, any new event of type R will be evaluated against all of them and up to 100 new partial matches could be created. Consequently, 100 new events of type R can create up to 10,000 partial matches, each of which will have to be evaluated upon each arrival of an event of type D.

To overcome this performance bottleneck, modern CEP systems utilize a variety of advanced optimization techniques. Parallelizing

event detection over multiple cores, VMs, or servers (which we commonly refer to as *execution units*) is a popular choice.

We will formalize the targeted problem as follows.

Given a CEP evaluation mechanism M, a workload of patterns $P = \{p_1, ..., p_n\}$, a set of homogeneous execution units $U = \{u_1, ..., u_m\}$, and a set of performance metrics $Perf$, execute M over U to detect P in a way optimizing the metrics in $Perf$.

In the remainder of this paper, we use throughput and peak memory consumption as our metrics in $Perf$. As we show in Section 5, this choice also positively impacts the average detection latency.

The solution presented in the next section assumes $|P| = 1$, i.e., only a single pattern is provided. We will target an extension to HYPERSONIC supporting multiple patterns in our future work (Section 7).

3 HYBRID PARALLELIZATION APPROACH

3.1 Overview

Our method is based on a two-tier parallel architecture. Given an automaton representing the pattern to be detected, the *outer* parallelization layer allocates a fraction of the available execution units to each state. The allocations are roughly proportional to the expected state load. Within a state, the *inner* parallelization process takes place by splitting the incoming data stream between the locally available execution units in a data-parallel manner utilizing a local load balancing strategy. To implement this approach, we introduce an *agent* - a logical unit of execution responsible for all calculations related to a given state. The outer layer is thus responsible for state-parallelizing the agents, while the per-agent inner layer executes the respective units in a data-parallel manner.

Figure 3 depicts a CEP system executing a HYPERSONIC automaton that detects a sequence of four event types in an event stream. Agents, denoted from now on by $A_1, ..., A_m$, are marked with circles. Event streams of all agents originate at the *splitter*, a lightweight component partitioning the global input stream by event type[1]. We assume the global stream to emit events in-order.

As described in Section 2.2, the task performed by a NFA state is to match between events belonging to a particular type and partial matches received from the state immediately preceding it in the automaton. Hence, each agent receives two input streams: the output stream of partial matches from the preceding agent and a substream of the system input stream restricted to the required event type[2]. The figure also shows the local storage buffers utilized by the agents to keep recently arrived elements. We discuss these buffers later on. Internally, each agent matches the new events with the accepted partial matches. The successfully extended matches are emitted into the output stream.

While the inner parallelization layer is based on a shared memory model, the outer layer is free of any such assumption. Adjacent agents communicate via a producer-consumer queue shared between their respective execution units, where the output stream of a preceding agent (producer) serves as the input stream of a succeeding agent (consumer). This makes our method naturally applicable to single-server and mixed (an agent runs on a single server; different agents are distributed) environments. Supporting a fully distributed use case will be the main focus of our future research (Section 7).

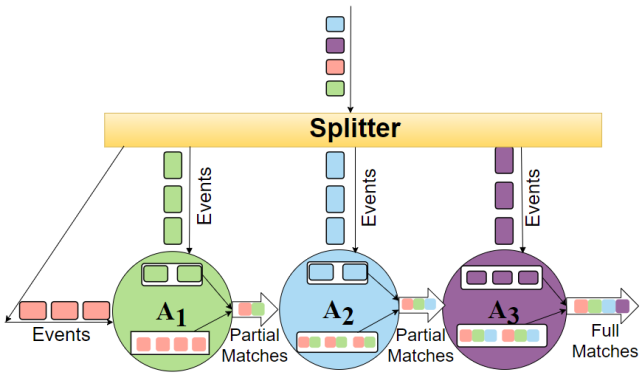


Figure 3: An instance of HYPERSONIC detecting a sequence of four event types. Vertical arrows denote the event streams, while horizontal arrows indicate the partial match streams.

In the remainder of this section, we will present the internal structure of an agent, the operations it supports, the load balancing mechanism for the outer (allocating execution units to agents) and the inner (scheduling execution units inside of an agent) layers, and the complexity analysis of our model.

3.2 Agent Internals

Unless otherwise stated, an agent receives two input streams and produces a single output stream[3]. These inputs are the *match stream (MS)* and the *event stream (ES)*. The match stream is essentially the output stream of the previous agent in the sequence containing the partial matches detected by that agent. The ES is a substream of the global input stream restricted to the type processed by this agent. It is emitted by the splitter as depicted in Figure 3.

Figure 4 illustrates the internal structure of the agent A_2 from Figure 3. The task of this agent is to match between newly arrived events of the third type in the sequence and previously formed pairs of events belonging to the first two types. These pairs are received from the preceding agent via the corresponding MS.

Due to the parallel nature of our model, the mutual order in which events and partial matches reach a given agent cannot be guaranteed. In Figure 4, when a new event arrives from the ES, not all potentially fitting partial matches could be available due to delay in processing in the preceding agent. Therefore, events have to be buffered upon arrival and later matched against new partial matches to guarantee detection correctness. Symmetrically, partial matches have to be stored and evaluated against later events.

An agent addresses this need by maintaining two local buffers, an *event buffer (EB or EB_i for agent A_i)* and a *match buffer (MB or MB_i for agent A_i)*, storing events and partial matches respectively. A newly arrived input item (either an event or a partial match) is first evaluated against the currently stored items in the opposite buffer (an incoming event is matched with all the matches located

[1]Since the splitter only inspects one event at a time to make a routing decision, it avoids the CEP scalability problem and can thus be implemented sequentially.

[2]The first agent in the sequence is an important exception. It has no preceding agent and hence it receives two input substreams corresponding to the first two event types.

[3]Extensions supporting more inputs and outputs are discussed in Section 4.

in the MB, and vice versa). The extended matches created from this evaluation are immediately transferred to the output stream of this agent. Then, the item that triggered the calculation is stored in its dedicated buffer (EB for events and MB for matches), thus allowing it to be combined with future items. This procedure closely resembles a popular strategy for parallelizing streaming joins [46, 63, 69, 76]. Since an item is only compared with items received prior to its arrival, every event-match pair is evaluated exactly once.

The above functionality is jointly performed by a number of parallel execution units, normally assigned to an agent on system startup[4]. At any given moment, each execution unit takes one of the two *roles*: (1) an *event worker* or (2) a *match worker*. Units assuming the role of event workers are responsible for receiving events from the ES, comparing them against the current content of the MB, and storing the events into the EB. Similarly, match workers receive, evaluate, and buffer partial matches.

It can be observed that the described method requires simultaneous write access of a large number of execution units to two potentially very large data structures: EB and MB. As every worker can add or remove items, these buffers could quickly become a major bottleneck due to the involved synchronization cost. We avoid this issue by distributing EB and MB among the workers. The distributed buffers can be seen in Figure 4. Each event worker or match worker maintains a fragment of the EB or the MB respectively. When, for example, an event worker receives a new event, it iteratively contacts all match workers to get the access to their respective MB fragments, then pushes the event to its own EB fragment. This design makes all inter-worker synchronization operations pairwise and improves the system throughput. In addition, it allows the system to support a fully distributed scenario where execution units could be located on different servers.

To avoid duplicate storage of events, which can potentially be of large size, we introduce *agent-global buffers (AGBs)*. An agent-global buffer contains all events that enter an agent, either via ES or via MS (as a part of a partial match), while the EB and MB only hold pointers to the content of the AGB.

As we explained in Section 2.1, pattern matches are only considered valid if they appear within a predefined time window W, that is, the maximal timestamp difference between the events in a match does not exceed W. Hence, the locally stored events and matches must be purged from their respective buffers once they expire. This is necessary to avoid the extreme growth of the buffers and to guarantee evaluation correctness that could otherwise be violated by returning expired events as a part of a pattern match.

HYPERSONIC purges expired events during its main evaluation loop, based on the timestamp of the latest available partial match (t_{latest}). We define the timestamp of a partial match as the timestamp of the earliest event it contains. The latest partial match is estimated upon each traversal of an event worker over the match buffer as part of its operation. An event whose timestamp t satisfies $t + W < t_{latest}$ can be safely removed. We assume W to be large enough to exceed possible processing and communication delay between agents. This assumption typically holds in practice unless W is extremely small, in which case the computation complexity of CEP is low and no parallelization is required.

[4]We will cover dynamic execution unit allocation in Section 4.1.

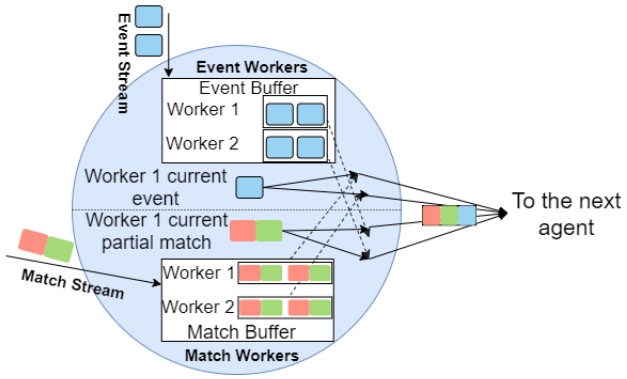


Figure 4: Internal components of the agent A_2 from Figure 3. For simplicity of presentation, the agent-global buffer is not shown.

Removing partial matches is done in a similar manner and is based on events arriving in-order. We find the timestamp of the latest event and any partial match that precedes it by at least W is removed.

The presented agent architecture requires a minor modification for patterns containing Kleene closure operators. As shown in Figure 2(b), NFA states handling Kleene closure contain an additional self-loop allowing them to repeatedly extend partial matches with more occurrences of the same event type. We implement this by routing all output matches emitted by a Kleene closure agent to its match stream in addition to the match stream of the next agent [5].

3.3 Execution Unit Allocation

We present below the two layers at which load balancing is performed in a hybrid-parallel CEP system. At the outer (inter-agent) layer, the pool of available execution units is divided between the agents. At the inner (intra-agent) layer, each execution unit assumes either a match worker or an event worker role.

3.3.1 Outer Load Balancing. The goal of the outer load balancer could be formalized as follows: *given a set* $U = \{u_1, ..., u_n\}$ *of homogeneous execution units, and a set* $A = \{A_1, ..., A_m\}$ *of agents derived from the NFA detecting the given pattern, partition* U *into* m *subsets* $\{U_1, ..., U_m\}$ *corresponding to the agents in* A *to maximize system performance according to the given set of metrics* $Perf$.

To decide on the most efficient partition, our load balancer requires additional statistical properties of the input stream, namely the *average arrival rate* of each event type and the *selectivity* (success rate) of each condition in the pattern. In this section, we assume these statistics to be constant and hence easily measurable by executing the system on a small fraction of the input stream. Accommodating shift in statistics will be addressed in Section 4.1.

More formally, given a pattern containing event types $E_1, ..., E_k$, the (average) arrival rate e_i of type E_i refers to the (average) number of new instances of E_i arriving per time unit. Given a chain-structured NFA consisting of states $S_1, ..., S_{m+1}$, we will denote as

[5]Note that this structure assumes a single event type under Kleene closure. Nested patterns, including multi-type Kleene closure operators, will be thoroughly addressed in our future research (Section 7)

Table 1: Table of notations.

E_i	The i^{th} event type in a pattern processed by HYPERSONIC.
W	The time window size of a pattern processed by HYPERSONIC.
S_i	The i^{th} state in a NFA detecting the pattern processed by HYPERSONIC.
A_i	The i^{th} agent in the processing chain, corresponding to S_i.
MB_i	Match buffer of A_i.
EB_i	Event buffer of A_i.
e_i	Average arrival rate of events of type E_i.
s_i	Average selectivity of conditions verified at the NFA state S_i.
m_i	Average number of partial matches entering agent A_i from agent A_{i-1}
$comp_i$	Computational load on A_i.
$sync_i$	Synchronization load on A_i.
$load_i$	Total load on A_i.
c_i	Average cost of a single comparison between an event and a match on A_i.
acc_i	Average number of A_i's buffer accesses per time unit.
b_i	Average cost of locking a buffer fragment in A_i.
q_i	Average cost of sending partial matches on the output queue of A_i.
m_i^{prev}	Average rate of partial matches arriving from the preceding agent of A_i (for Kleene closure agents).
v_i	Average size of an event of type E_i.
p	The system-wide size of an event pointer.
a_i	Average number of events in a partial match of MB_i.

state selectivity s_i the fraction of match-event pairs matched in S_i whose evaluation results in a new extended partial match.

Intuitively, our goal is to have the same load on each agent so there are no agents sitting idle while others are overloaded. To achieve this, we model the load on each agent and then allocate the execution units proportionally to the calculated load. Note that, due to the assumed homogeneity of the execution units, it is enough to only calculate the sizes $|U_1|, ..., |U_m|$. Let $load_i$ denote the load on the i^{th} agent. Then, the following is the number of execution units that the load balancer assigns to this agent:

$$|U_i| = \frac{load_i}{\sum_{j=1}^{m} load_j} \cdot |U|.$$

We define the total agent load as the sum of its computation load and synchronization load: $load_i = comp_i + sync_i$. To calculate $comp_i$ and $sync_i$, we will first define the *(average) partial match arrival rate* m_i as the (average) number of partial matches entering agent A_i from agent A_{i-1} per time unit. m_i can be recursively calculated using the following rule:

$$m_i = \begin{cases} e_1, & i = 2; \\ 2m_{i-1}e_{i-1}s_{i-1}W & i > 2. \end{cases}$$

The explanation to this rule is as follows. Each agent except for the first one in the sequence receives e_i incoming events and m_i incoming matches. Each event is matched against each buffered match, and, on average, $|MB_i| = m_i W$. Considering the condition selectivity, for each incoming event $m_i W \cdot s_i$ output matches are generated for the total of $e_i \cdot m_i W \cdot s_i$. Similarly, the average size of the event buffer is $|EB_i| = e_i W$, and the incoming partial matches contribute $m_i \cdot e_i W \cdot s_i$ output matches. The first agent in the sequence represents the first two states in a NFA and receives two event substreams (Section 3.1). For convenience of notation, we will start counting agents from $i = 2$ and set $|U_1| = 0$.

Following the same logic, it can be observed that the average number of computations is proportional to the sum of the number of comparisons between the events in ES and matches in MB, and between the matches in MS and events in MB, i.e., $2e_i m_i W$. Let c_i denote the average cost of a single comparison between an event and a match on agent A_i. Obviously, c_i differs between agents due to the different sizes of partial matches and can be found empirically. Then, the average number of computations per time unit is given by $comp_i = 2c_i e_i m_i W$.

As we described in Section 3.2, the average number of synchronization actions on an agent depends on the number of times a worker accesses a fragment of the buffer of the opposite role, that we will denote as acc_i. For each incoming event, this is the number of the allocated match workers, and vice versa. As these numbers are not known in advance during load calculation, we will use an approximation assuming equal allocation. That is, for calculating acc_i we assume that each agent receives the same number of n/m workers and exactly one half was assigned each role, resulting in $n/2m$ event and match workers.

Defining b_i as the cost of locking a buffer fragment, we obtain

$$acc_i = e_i \cdot \frac{n}{2m} + m_i \cdot \frac{n}{2m} = \frac{(e_i + m_i)n}{2m}.$$

In addition, the synchronization load includes the cost of sending partial matches using the concurrent queue at a rate of m_{i+1} at a load cost of q_i. Thus, the total synchronization load of an agent is

$$sync_i = acc_i b_i + q_i m_{i+1} = \frac{(e_i + m_i)nb_i}{2m} + q_i m_{i+1}.$$

For agents implementing a Kleene closure operator, m_i is calculated differently due to the presence of a self-loop. Due to lack of space, we omit the intermediate steps and only present here the

final expression:

$$m_i = m_i^{prev}\left(1 + \sum_{j=1}^{e_i W}\left((e_i)^j (s_i)^j W^j\right)\right),$$

where m_i^{prev} is the rate of partial matches arriving from the preceding agent.

3.3.2 Inner Load Balancing. The goal of the inner load balancer is to assign the roles of event and match workers to these execution units such that the throughput and the peak memory consumption of A_i are optimized.

Initially, worker roles are assigned to execution units using a simple heuristic that chooses a half of the units at random to be event workers, and the rest to be match workers. However, our empirical evaluation results show that this strategy often leads to workers becoming idle for long periods of time due to discrepancies between the values of e_i and m_i, which ultimately leads to suboptimal system performance.

Furthermore, even if a smarter strategy for assigning intra-agent roles was implemented, it would not solve the problem in its entirety. On-the-fly fluctuations in the input statistics could lead to significant deviations of e_i and m_i from their initial values [40], severely degrading the event detection process. As an example, a sudden increase in e_i could lead to a faster growth of EB_i, increasing the number of events evaluated against each new partial match. Consequently, the processing rate of partial matches would slow down, leading to slower growth of MB_i. This would allow the event workers to handle new events even faster due to fewer evaluations needed, further increasing EB_i and exacerbating the problem. The imbalance between the match workers and the event workers would keep growing, with the former being overloaded and the latter staying idle most of the time.

To improve the stability, flexibility, and adaptivity of our system, we utilize the *role-dynamic* model. On startup, each execution unit allocated to an agent receives a *primary role* and assumes the other role to be its *secondary*. Primary roles are assigned by splitting the units into two equal sets at random. At runtime, an execution unit first tries to follow its primary role. If no work is currently available (that is, the corresponding stream is empty), it temporarily proceeds to the secondary role and checks the second input stream for input. As we show in Section 5, this strategy allows to efficiently spread the load on an agent and overcome runtime situations where one of the input streams has a significantly higher rate than the other.

As the execution units alternate between their roles, they have to simultaneously manage fragments of both the EB and the MB and satisfy access requests for both from other units. While this leads to more synchronization operations as compared to the basic model, the benefits greatly outweigh this negative impact.

3.4 Complexity Analysis

The total number of calculations performed by the system per time unit is given by $\sum_{i=1}^{m} comp_i$, where $comp_i$ is given by the formula provided in Section 3.3.1. As expected, a larger time window, faster arrival rate of events, and longer patterns all generate more calculations and increase this value.

Memory complexity is given by the sum of the sizes of all event buffers and match buffers in the system. As discussed in Section 3.2, all buffers only contain pointers of a constant size p, while the actual events are located in agent-global buffers. As each buffer stores both events received from its agent stream as well as potentially all events from the previous agents (received via MS), the size of all agent-global buffers containing the actual events (Section 3.2) is given by $\sum_{i=1}^{n}(e_i v_i W + \sum_{j=1}^{i-1} e_j v_j W)$, where v_i is the average size of an event that is handled by agent A_i[6]. The size of an event buffer EB_i is given by $|EB_i|p$, where p is the size of an event pointer, which we assume to be a system-wide constant. Each partial match in MB_i contains pointers to events of types covered by the previous agents in the sequence. Let a_i be the average number of events in a partial match of MB_i. Then, the size of a match buffer MB_i is given by $|MB_i|a_i p$. For a non-Kleene agent, $a_i = a_{i-1} + 1$. For an agent i containing a Kleene closure operator, the self-loop traversal is taken into account, resulting in the following expression:

$$a_i = a_{i-1} + \sum_{j=1}^{e_i W} \frac{m_i^{KC_j} \cdot j}{\sum_{k=1}^{e_i W} m_i^{KC_k} + m_i^{prev}} + 1,$$

where $m_i^{KC_j}$ is the rate of partial matches arriving from the self-loop that has exactly j events of the type E_i. Calculation of this value is omitted due to lack of space.

By summing the sizes of all buffers, including the global event buffer, we obtain the following expression for the memory complexity of our mechanism:

$$\sum_{i=1}^{n}\left(e_i v_i W + \sum_{j=1}^{i-1} e_j v_j W + (e_i W + m_i a_i W)p\right).$$

Similarly to the throughput, the expected memory usage scales with the time window, the arrival rates, and the pattern length.

The number of synchronization actions per time unit was also calculated in the previous subsection and it equals $\frac{(e_i+m+i)nb_i}{2m} + q_i m_{i+1}$. This expression is most affected by the time window, because a larger window leads to more evaluations and therefore more synchronization actions. Also, as expected, more workers in the system require more synchronization.

4 OPTIMIZATIONS AND EXTENSIONS

The hybrid-parallel approach presented above combines the strengths of state-parallel and data-parallel methods and overcomes their weaknesses. It provides an unbounded degree of parallelism since any number of execution units can be allocated to an agent. In addition, no complex synchronization protocol between the agents is required as the different agents communicate via the match streams.

However, the hybrid parallelization scheme also inherits some of the distinctive drawbacks of the other two approaches. Like in any data-parallel method, it employs a complex execution unit allocation scheme which is highly sensitive to fluctuations in the statistical properties of the input stream. Similarly to state-parallel

[6] As events of different types can contain different sets of attributes, their sizes could also differ.

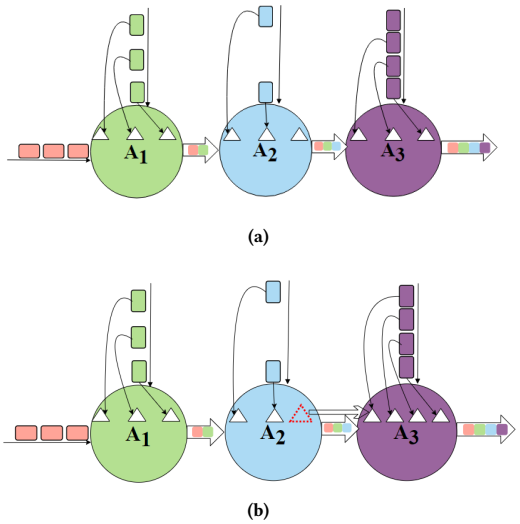


Figure 5: An agent-dynamic execution unit allocation example. Triangles denote execution units. (a) Initial allocation, agent A_3 is overloaded due to the shift in event arrival rates; (b) An idle worker of A_2 joins A_3 after finding available items in EB or MB of A_3.

mechanisms, the pipelined structure of its outer level imposes a high lower bound on the event detection latency.

In this section, we present two system optimizations that aid in minimizing the impact of runtime changes on the performance and in reducing the detection latency. We experimentally evaluate these optimizations in Section 5. In addition, we discuss how our model could be extended to operate in more diverse scenarios.

4.1 Agent-Dynamic Execution Unit Allocation

As we discussed in Section 3.3.2, on-the-fly fluctuations in the measured stream statistics could render the initial allocation inefficient. Here we address the outcome of this problem at the outer layer of the hybrid model: some agents become idle for long periods of time, while others collapse under heavy load.

To address this problem, we incorporate a solution that we call the *agent-dynamic* model. In addition to the worker roles, now each execution unit has a *primary agent*, the agent to which this unit was assigned during the initial allocation (Section 3.3.1). The input selection strategy is summarized in Algorithm 1. First, a worker tries to get an item from its primary agent in line 2. It can get either an event or a partial match as per the role-dynamic model explained in Section 3.3. When no work is available on the primary agent for both the primary and the secondary role of some execution unit, it picks an agent at random and attempts to evaluate items from its input streams, as shown in the lines 4-5. If the new agent is also idle (that is, it has no available events or partial matches), the random search continues. Once a non-idle agent is found, it becomes the *current agent* of this execution unit, with the latter effectively changing its allocation from the primary agent to the current one. Finally, the function returns both the input and the agent that was chosen to process it. That agent would be the primary agent in the subsequent call to the function.

```
1 get_input_item(primary_agent):
2     input ← get_item_from_stream(primary_agent)
3     while (input == null):
4         primary_agent ← get_random_agent()
5         input ← get_item_from_stream(primary_agent)
6     return input, primary_agent
```

Algorithm 1: Agent-dynamic input selection algorithm.

Figure 5 illustrates the idea. In a system containing three agents A_1, A_2, A_3 and nine execution units, initially three units are assigned to each agent based on the known statistics. At some moment during the run, the input distribution changes and A_3 becomes overloaded, while at the same time the load on A_2 decreases (Figure 5(a)). As a result, an idle worker on A_2 picks a new agent at random. As the load on A_1 still matches the number of allocated workers, A_3 ends up being selected and hosts the worker from now on (Figure 5(b)).

Moving from one agent to another requires an execution unit to initialize and maintain a new buffer fragment, while the previous one expires only when all of its contained items expire. To avoid situations in which an execution unit switches agents frequently and holds a large number of buffer fragments, we limit the rate of such "hops" to at most one per W. We also prevent the last event worker and the last match worker of an agent from migrating.

An alternative solution for the presented problem would be to implement a more intelligent algorithm, periodically estimating the up-to-date load $load_i$ of each agent A_i and routing idle execution units to agents with the highest load. At the first glance, this strategy could make the system immediately locate the bottlenecks and converge faster. However, this advantage is not as significant as it may seem. To identify the most loaded agent, an idle unit has to inspect all, or at least a large fraction of load indicators. To that end, it either has to contact multiple agents, or utilize some kind of centralized memory, violating the fully decentralized design. In addition, a load indicator could be updated at any time and introduce potential race conditions, making the decisions of the idle units unreliable.

4.2 Agent Fusion

While HYPERSONIC does not define an upper bound on the parallelism degree, it requires a minimal number of execution units to operate. For each agent, at least one event worker and at least one match worker are needed to guarantee that no results are lost. For a pattern with m types, this results in a lower bound of $2m-2$ workers (Section 3.3.1). In addition, the detection latency is bounded from below by the time needed to perform $m-2$ inter-agent transfers of a partial match.

Depending on the user requirements, this situation could be acceptable when resource utilization of all agents is high. However, it might cause "lightweight" agents to be overprovisioned. Given an agent that matches between very rare event types or performing a computationally light computation, the load calculation described in Section 3.3.1 will typically assign this agent a tiny fraction of execution units. However, the actual allocation will be different due

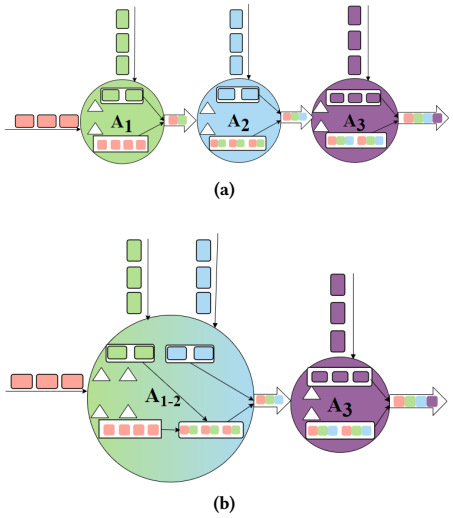


Figure 6: An example of agent fusion. Triangles denote execution units. (a) Before fusion: agent A_1 is overprovisioned despite only having two assigned execution units; (b) After fusion: the new agent A_{1-2} now has four execution units of which two will possibly be reallocated due to another iteration of the unit allocation procedure.

```
1  cost_allocation_with_fusion(agents, input_parameters):
2      allocations ← allocate_workers(agents,
         input_parameters)
3      for i = 0; i < agents.size ; i++:
4          if allocations[i] < 2 :
5              if allocations[i − 1] < allocations[i + 1] :
6                  Â← fuse_agents(A_{i−1}, A_i)
7                  agents.remove(A_{i−1}, A_i)
8              else:
9                  Â← fuse_agents(A_i, A_{i+1})
10                 agents.remove(A_i, A_{i+1})
11             agents.add(Â)
12             allocations ← allocate_workers(agents,
                 input_parameters)
13     return allocations
14 fuse_agents(A_i, A_{i+1}) :
15     Â ← new_agent()
16     Â.ES ← merge(A_i.ES,A_{i+1}.ES)
17     Â.MS ← A_i.MS
18     Â.output ← A_{i+1}.output
19     return Â
```

Algorithm 2: Worker allocation with fusion.

to the lower limit of two units. While the load balancing problem could be solved with agent-dynamic allocation, the latency bound would still persist despite the undeniable redundancy of this agent.

As an example, consider an instance of HYPERSONIC with three agents A_1, A_2, A_3. Further, assume that no condition is defined between the event types processed by A_1, that is, its only job is to forward pairs of events to A_2. Even if we only allocate one match worker and one event worker to A_1, this might lead to underutilization of resources. This scenario is presented in Figure 6(a).

As a solution, we introduce *agent fusion*. Fusion [38] refers to merging two or more processing units into one as an optimization step. In our case, fusion is performed by uniting two consecutive agents, at least one of which is overprovisioned as described above, into a single structure preserving their joint functionality. The fusion procedure is shown in detail in Algorithm 2.

More formally, given a pair of agents A_i and A_{i+1} with input streams and buffers as defined in Section 3.2, agent fusion is done as follows. The fused agent $\hat{A}_i$ retains the input match stream MS_i and has two input event streams ES_i and ES_{i+1} (lines 16-17 in Algorithm 2). The output stream of $\hat{A}_i$ is identical to that of the original A_{i+1} (line 18 in Algorithm 2). In addition, $\hat{A}_i$ contains two event buffers and two match buffers corresponding to those of A_i and A_{i+1}, and all of their execution units, whose roles are reassigned following the fusion operation. Each match worker now contains a fragment of both MB_i and MB_{i+1}, and each event worker contains a fragment of EB_i and EB_{i+1}.

The matching procedure in a fused agent emulates the joint functionality of the two underlying agents. A new event e_i arriving from ES_i is stored in EB_i and evaluated against MB_i, with new matches resulting from this evaluation written into MB_{i+1} rather than transferred to the next agent. Upon arrival of an event e_{i+1} from ES_{i+1}, it is stored in EB_{i+1}, matched with the contents of MB_{i+1} and the results are written to the output stream of $\hat{A}_i$. Matches arriving from MS_i are stored in MB_i, then compared to the events in EB_i, with each result written to MB_{i+1} triggering a comparison against EB_{i+1}. As each worker in the fused agent maintains two buffer fragments, having only two workers, rather than a minimum of four, is sufficient for the agent to operate.

Figure 6(b) illustrates the idea. By merging the "lightweight" agent A_1 with the adjacent A_2, we eliminate the need to allocate two execution units to A_1. We also reduce the number of inter-agent communication channels, thus lowering the latency.

Algorithm 2 depicts the application of the fusion mechanism during the initial allocation phase. When the execution unit number $|U_i|$ for some agent A_i is smaller than 2, the fusing process is initiated (line 4). As the condition in line 5 states, A_i is fused with either the preceding or the succeeding agent (the one with the smaller load, lines 6-10). Subsequently, the unit allocation procedure is restarted in line 12. In our future work, we will focus our research efforts on a scheme for efficient *dynamic fusion* applied on-the-fly.

5 EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental study. We evaluated HYPERSONIC against state-of-the-art parallel solutions and assessed the impact of the extensions presented in Section 4.

5.1 Experimental Setup

All our experiments were conducted on two real-world datasets. The first dataset was taken from the NASDAQ stock market historical records [2]. Each record represented a single update to the price of a stock. The data we used spanned a one month period covering over 2100 stock identifiers with prices updated periodically. Our input stream contained 6,239,997 primitive events where each event is a stock price update consisting of a stock identifier, a timestamp, and a current price. A separate event type was defined for each identifier. The second dataset included 13,956,534 measurements from smart home sensors utilized for recognizing human activities [22]. An event is a sensor reading containing a timestamp, the activity recorded by the sensor (that we defined as the event type), and 33 additional attributes providing raw data such as the person acceleration and the distance from predefined locations.

For each of the datasets, we created over 10 patterns varying in structure and ranging between 3 and 7 participating event types. Each pattern represented a sequence of primitive events, with about 20% of the patterns containing either a negation or a Kleene closure operator on some event type. For the Kleene closure patterns, all queries used were of the same length, because the Kleene closure operator was the most demanding by a wide margin, and thus changing the pattern's length would not produce significantly different results. Each pattern included a number of dataset-specific predicates, roughly equal to the pattern length, motivated by real-world use cases related to the respective domain of each dataset as we specify below.

For the stock dataset, the conditions were motivated by the problem of detecting closely correlated stock tickers. To that end, we augmented each event with an additional attribute named *history* and containing an array of 20 last recorded stock prices. A condition between stock ticker identifiers A and B was then formulated as $Corr(A.history, B.history) > T$, where $Corr$ stands for Pearson's correlation coefficient [56] and T is a predefined threshold. Table 2 lists the queries for used for the stocks dataset.

For the sensor dataset, the conditions were designed to detect transitions between zones. A typical condition was of the form $A.distanceX < B.distanceY$, where $distanceX, distanceY$ denote the attributes specifying the distance of the person from locations such as a bathroom or a bedroom.

We selected throughput, latency, and memory consumption as our performance metrics for this study. Throughput was defined as the number of primitive events processed per second. To estimate the memory consumption, we measured the peak memory required by the system during evaluation. Finally, the latency of detecting a pattern match was calculated as the difference between the detection time of the match and the arrival time of the latest event comprising it. All metrics were acquired separately for each of the generated patterns, and the presented results were then calculated by taking the average.

We evaluated HYPERSONIC against a sequential baseline, a state-parallel approach, and two state-of-the-art data-parallel methods, RIP and LLSF. In a state-based system [12], each NFA state is assigned a single execution unit. RIP [12] divides the input stream into batches according to the event sequence number in a round-robin manner. LLSF (least-loaded-server-first) [73] utilizes a greedy heuristic, allocating each event to the execution unit with the lowest measured load. For more information on these and other CEP parallelization methods, we refer the reader to Section 6.

Table 2: Query templates used during the experiments.

	Stocks Queries - A
Q_1^A	**SEQ**$(S_1, S_2, \ldots, S_n)$ $n \in \{3,7\}$ **WHERE** $\forall i \in \{2,n\}$: Corr$(S_i - 1, S_i) >$ T
Q_2^A	**SEQ**$(S_1, \ldots,$ **KLEENE**$(S_j), \ldots, S_n)$ $n \in \{6\}$ **WHERE** $\forall i \in \{2,n\}$: Corr$(S_i - 1, S_i) >$ T
Q_3^A	**SEQ**$(S_1, \ldots,$**NEG**$(S_j), \ldots, S_n)$ $n \in \{3,7\}$ **WHERE** $\forall i \in \{2,n\} \setminus \{j\}$: Corr$(S_i - 1, S_i) >$ T
	Sensor Queries - B
Q_1^B	**SEQ**$(S_1, S_2, \ldots, S_n)$ $n \in \{3,7\}$ **WHERE** $\forall i \in \{2,n\}$: S_i.distance $>$ S_{i-1}.distance
Q_2^B	**SEQ**$(S_1, \ldots,$ **KLEENE**$(S_j), \ldots, S_n)$ $n \in \{6\}$ **WHERE** $\forall i \in \{2,n\}$: S_i.distance $>$ S_{i-1}.distance
Q_3^B	**SEQ**$(S_1, \ldots,$**NEG**$(S_j), \ldots, S_n)$ $n \in \{3,7\}$ **WHERE** $\forall i \in \{2,n\} \setminus \{j\}$: S_i.distance $>$ S_{i-1}.distance

In our implementation of all evaluated methods, we have thoroughly ensured full detection correctness, that is, that each method returns all matches in the dataset, and only those matches. In all our experiments, we validated that the matches emitted by all compared algorithms are identical and include all existing matches, and only them.

The initial event arrival rates and condition selectivities required for allocating the execution units (Section 3.3) were measured as a preprocessing step using the techniques described in [40]. As we discussed in Section 4.1, these values may frequently and significantly fluctuate during runtime. We demonstrate the impact of these oscillations in a dedicated experiment (Figure 11).

All experiments were run on a single server with 24 physical cores, 2.20 GHz CPU, and 16.0 GB RAM. The parallel systems were given all 24 cores unless stated otherwise. All models and algorithms were implemented in Java.

5.2 Experimental Results

5.2.1 State-of-the-art comparison. Figures 7-9 show the performance metrics obtained during comparative evaluation of HYPERSONIC against the baseline sequential CEP evaluation mechanism, the state-based approach, and the state-of-the-art RIP and LLSF algorithms as presented in Section 5.1. The results are presented using logarithmic scale. To assess the scalability of the examined methods in various scenarios, we experimented with different degrees of parallelism (i.e., number of available cores) and pattern time windows.

Throughput comparison is displayed in Figure 7. For clarity of presentation, we report the results in terms of the relative throughput gain of the parallel methods over the sequential baseline. In addition to the time window and the number of cores, we also repeated the experiments for patterns of different length.

HYPERSONIC consistently achieves a considerable throughput gain over the non-parallelized system, reaching an improvement

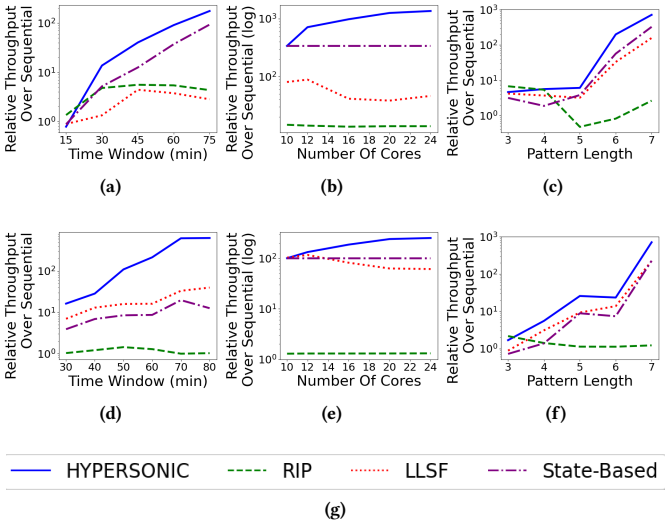


Figure 7: Relative throughput gain of HYPERSONIC, RIP, LLSF, and the state-based method over a sequential baseline (logarithmic scale, higher is better), applied on the stock dataset ((a)-(c)) and on the sensor dataset ((d)-(f)), as a function of: (a),(d) time window; (b),(e) number of cores; (c),(f) pattern length. HYPERSONIC outperforms RIP by 2 to 3 orders of magnitude, and achieves 2 to 50 times higher throughput than LLSF.

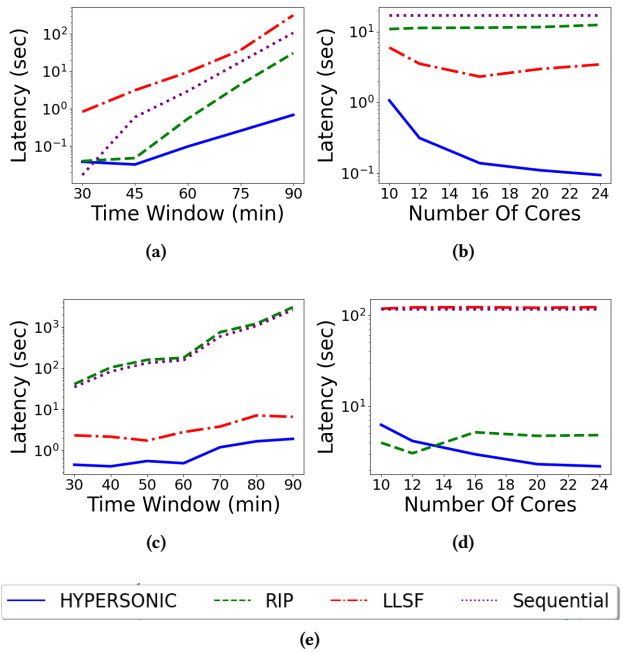


Figure 8: Pattern detection latency of HYPERSONIC and state-of-the-art methods, RIP and LLSF (logarithmic scale, lower is better), applied on the stock dataset ((a),(b)) and on the sensor dataset ((c),(d)), as a function of: (a),(c) time window; (b),(d) number of cores. For large windows and parallelism degrees, the latency of HYPERSONIC is 2 to 60 times smaller than that of the runner-up.

of up to three orders of magnitude. A key factor in obtaining high *superlinear speedup* is faster access of memory in HYPERSONIC, a result of efficient parallelization of memory distribution and usage. Indeed, memory is a major bottleneck of CEP systems, caused by the high number of partial matches that need to be stored and frequently accessed. In turn, large and frequently accessed memory usage causes a high rate of misses in the memory hierarchy. HYPERSONIC memory balancing scheme avoids duplicate storage of partial matches, which leads to close-to-optimal distribution of the corresponding memory accesses (see Figure 9). In turn, per-core memory and access reduction enable much better cache utilization, leading to less cache misses and improving the average memory access considerably (for more details, see also [60]). This leads to the following "super-scalability" phenomenon: the higher the number of cores assigned to HYPERSONIC the better it uses each of them and the better are its chances for achieving superlinear speedup.

Figure 7 also shows that the obtained speedup is proportional to the increase in pattern complexity, both in terms of longer time windows and larger pattern lengths. This outcome was expected as longer patterns (in terms of number of event types) as well as longer time windows are known to lead to an exponential rise in per-event computations, a well-established result in CEP research [39, 41, 43, 52, 57, 74]. Consequently, the computation to synchronization action ratio grows with the pattern length and the window size, which benefits parallel systems. Also, longer patterns have more agents and hence less workers per agent, leading to lower inter-agent synchronization degree.

Similar to the results mentioned above, Figures 7(b) and 7(e) show that the hybrid-parallel approach also scales well when adding more execution units. In comparison, LLSF only displayed a steady and consistent increase in the throughput gain as a function of the pattern length, while RIP did not scale at all with the growth in any of the considered parameters. The state-based approach scales better than RIP and LLSF with the pattern complexity but fails to scale with the number of cores as the additional cores are not utilized by that approach. All in all, HYPERSONIC outperformed the RIP algorithm by a factor of two to three orders of magnitude, and achieved 2 to 50 times higher throughput than LLSF while improving over the state-based method by up to 70 times.

Parallelized CEP mechanisms often suffer from increased latency due to out-of-order pattern detection and the overhead of communication between the execution units. However, an efficient load balancing scheme could greatly reduce the latency by decreasing the average time spent by a data item in an input queue while waiting to be processed. The superiority of the two-tier load balancing strategy of HYPERSONIC can be observed in Figure 8 that depicts the latency of the compared methods obtained during our experiments as a function of the time window size and the number of cores. Except for a few relatively simple scenarios (small time window size for the stock dataset and low parallelism degree for the sensor dataset) our method operates with the lowest latency.

Remarkably, there is no consistent runner-up, as both RIP and LLSF struggled to maintain their advantage.

Figure 9 demonstrates the peak memory consumption of each of the compared methods. The memory consumption of a parallel system was approximated by independently measuring the highest memory usage in each execution unit and summing the results.

The results fully match our analysis in Section 3.4, as the performance deteriorates nearly linearly with an increase in the time window. Additionally, while we expected all parallel runs to consume more memory as compared to the sequential baseline due to the simultaneous processing of events and partial matches, HYPERSONIC actually demonstrated a reduction in memory usage in most experiments. This can be explained similarly to the latency improvement: an efficient load balancing strategy minimizes the waiting time of a data item and thus considerably reduces the average queue length. For large patterns, the partial matches are the most dominant contributor to the memory consumption as they represent sets of primitive events. By reducing the average time a pattern match waits to be processed, HYPERSONIC decreases the average number of simultaneously buffered partial matches. In addition, since the procedure of purging expired partial matches is performed in parallel by all participating execution units, it takes less time on average for an expired match to be deleted.

Figure 10 shows the impact of our load balancing scheme on the system performance. As described in 3.3, HYPERSONIC allocates workers to agents based on their expected load. To demonstrate the effectiveness of this allocation, we evaluated the throughput of a restricted version of HYPERSONIC, where the load balancing was disabled and all agents were given an equal number of workers. This version was compared to a "complete" version of HYPERSONIC. As can be observed in Figure 10, the cost model allocation performs 1.8 to 3 times better than the trivial allocation.

5.2.2 Impact of extensions on the system performance. We tested the effect of the extensions proposed in Section 4 on the performance of HYPERSONIC. This study was conducted solely on the stock dataset, and we experimented with varied window lengths and number of cores.

Figure 11 depicts the impact of applying agent-dynamic execution unit allocation as opposed to only using role-dynamic allocation (Section 4.1). Applying this optimization results in the consistent boost in throughput for every tested configuration. However, it can be observed that HYPERSONIC benefits from this extension the most when the parallelism degree is low. There are two reasons for this observation. First, an idle execution unit degrades the system performance by a larger fraction when the overall number of units is lower. Because the agent-dynamic allocation utilizes the idle units, systems where the relative importance of a single execution unit is higher gain more from utilizing this strategy. The second reason is the synchronization overhead introduced by the agent-dynamic allocation. This overhead grows with the total number of the available execution units, resulting in smaller improvement in the system throughput when this number is larger.

In Figure 12, we present the results of applying fusion on the initial set of agents. The experiments were performed on a fixed set of three patterns of length 6. For each pattern, we fixed a pair of adjacent agents in advance and fused them on system initialization.

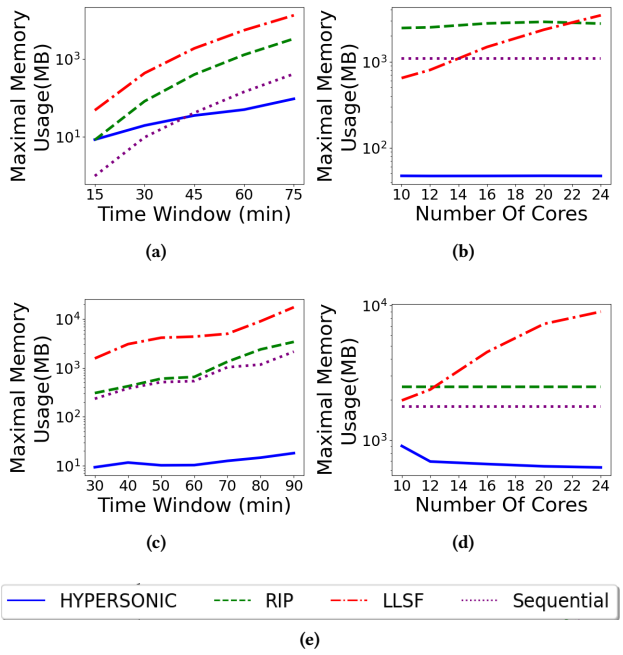


Figure 9: Peak memory consumption of HYPERSONIC and state-of-the-art methods, RIP and LLSF (logarithmic scale, lower is better), applied on the stock dataset ((a),(b)) and on the sensor dataset ((c),(d)), as a function of: (a),(c) time window; (b),(d) number of cores. For large windows and parallelism degrees, HYPERSONIC consumes at least 5 times less memory than the runner-up.

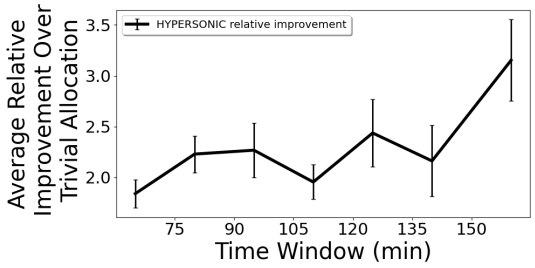


Figure 10: Average relative throughput (higher is better) of HYPERSONIC as compared to a restricted version of HYPERSONIC where the load balancing scheme is replaced with a trivial allocation. The graph is displayed as a function of the time window. Our load balancing model allows to improve the throughput of "raw" HYPERSONIC by a factor of up to 3. The error bars indicate the minimal and the maximal values obtained over all patterns.

In all but one experiment, the fused evaluation mechanism achieved lower average latency, returning pattern matches up to two times faster than the version without the extension.

In addition, while the focus of the fusion optimization is on decreasing the latency, we also witnessed a considerable increase in the throughput. The performance boost is achieved thanks to

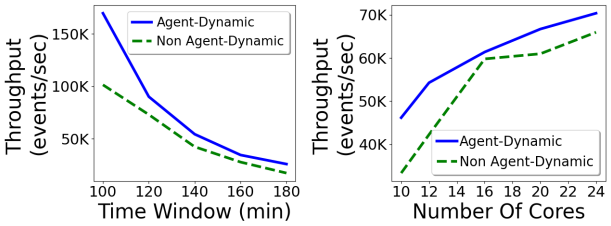


Figure 11: Throughput (higher is better) achieved by applying the agent-dynamic allocation extension (Section 4.1) as compared to the basic version, displayed as function of (a) the time window and (b) the number of cores.

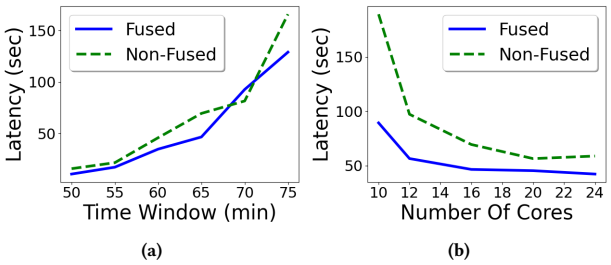


Figure 12: Latency (lower is better) achieved by applying the fusion extension (Section 4.2) as compared to the basic version, displayed as function of (a) the time window and (b) the number of cores.

restarting the unit allocation process after applying the fusion. In the revised allocation, the fused agent is allocated less workers than the sum of the two agents that were fused (otherwise, it would not have been fused). Those orphaned workers are now allocated to other agents that were not overprovisioned and thus the extra workers will contribute to faster processing at those agents.

6 RELATED WORK

Complex event processing has been an increasingly active research field in recent years [26, 27, 32]. Following the success of earlier data stream management systems [4, 10, 19, 20], many CEP frameworks were developed [5, 13, 24, 52, 72]. CEP functionality is widely available in many commercial stream processing products [9, 18, 64, 75], as well as via dedicated open-source software libraries [1, 3, 29, 67]. All of these solutions utilize a graph-based evaluation structure such as a NFA or a tree, making it possible to utilize HYPERSONIC to further boost their performance and improve the scalability.

Numerous techniques have been proposed for optimizing the performance of CEP and data stream processing in general [38]. Notable examples include rewriting the pattern into an equivalent, yet more efficient representation [7, 43, 52, 66], sharing common subexpressions to optimize simultaneous processing of multiple similar patterns [7, 29, 48, 59, 75], and utilizing sophisticated data structures to efficiently support Kleene closure [57, 74].

Parallelization approaches for CEP systems are being actively researched [62]. RIP [12] and LLSF [73] evaluated in Section 5 are two prominent state-of-the-art solutions. RIP [12] applies a data-parallel approach by partitioning the workload into multiple intersecting windows. Each execution unit is allocated a fixed-sized chunk of incoming events in a round-robin manner. To avoid losing matches that overlap two adjacent chunks, some of the events are replicated to two neighboring units. The authors also describe a simple state-parallel algorithm and experimentally demonstrate that it performs considerably worse than the data-parallel one. Xiao et. al. [73] propose three data-parallel strategies for distributed complex event processing: RR (round-robin), JSQ (join-the-shortest-queue), and LLSF (least-loaded-server-first). They empirically show that the latter strategy is superior to the other two.

Many other works explored the area of CEP parallelization. Hirzel [37] proposed a data-parallel algorithm dividing the input stream on the predefined event attributes. In contrast to the majority of CEP parallelization methods, this approach only works on patterns requiring all events in a match to agree on the value of a certain attribute, denoted as 'partition key' in the paper. This severe restriction makes it impossible to directly evaluate [37] against HYPERSONIC. Mayer et. al. [49–51], split the workload into windows that can be defined by an arbitrary predicate. [17] is a distributed data-parallel extension of Cayuga [16]. Wang et. al. [70] presented an event stream partitioning mechanism based on fixed-size sub-windows. Other authors explored strategies based on state- and task-level parallelism [25, 65, 78], as well as on embedded architectures such as FPGAs [71] and GPUs [25]. None of the above works consider a combination of data and state parallelism.

General stream processing (SP) is a broad term covering the execution of arbitrary computations over data streams. Various methods have been proposed for distributing the computations performed by SP systems [33, 53, 54, 61, 68]. In particular, multiple authors addressed streaming joins [63, 69, 76]. Despite the similarities between SP and CEP, the challenges associated with parallelizing these two system types are fundamentally different. In contrast to CEP, SP systems typically execute highly computationally intensive tasks, put less emphasis on stateful operators, and lack the dependencies caused by combining multiple items.

Parallel and distributed processing of large databases has been a major research trend during the past decades [8, 44, 55]. The most widely employed model providing such capabilities is MapReduce [28, 30, 45]. Many other models have been proposed [14, 21].

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented HYPERSONIC, a novel method for parallelizing CEP applications. To the best of our knowledge, HYPERSONIC is the first to combine the strengths of the state-parallel and the data-parallel approaches while overcoming their major disadvantages. Our experiments demonstrated a significant throughput improvement over the state-of-the-art methods, while achieving lower latency and consuming less memory.

Our future research efforts will cover a number of essential extensions to HYPERSONIC, including fully distributed execution, supporting heterogeneous execution units, multi-pattern CEP, nested patterns, as well as additional evaluation mechanisms such as trees and EPNs.

REFERENCES

[1] [n.d.]. http://flink.apache.org.
[2] [n.d.]. http://www.eoddata.com.
[3] [n.d.]. http://www.espertech.com.
[4] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. 2005. The design of the Borealis stream processing engine. In *CIDR*. 277–289.
[5] A. Adi and O. Etzion. 2004. Amit - the Situation Manager. *The VLDB Journal* 13, 2 (2004), 177–203. https://doi.org/10.1007/s00778-003-0108-y
[6] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. 2008. Efficient Pattern Matching over Event Streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. ACM, New York, NY, USA, 147–160. https://doi.org/10.1145/1376616.1376634
[7] M. Akdere, U. Çetintemel, and N. Tatbul. 2008. Plan-based Complex Event Detection Across Distributed Sources. *Proc. VLDB Endow.* 1, 1 (2008), 66–77.
[8] Alaa Aljanaby, Emad Abuelrub, and Mohammed Odeh. 2005. A Survey of Distributed Query Optimization. *Int. Arab J. Inf. Technol.* 2, 1 (2005), 48–57.
[9] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. 2006. SPC: A Distributed, Scalable Platform for Data Mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms* (Philadelphia, Pennsylvania). ACM, New York, NY, USA, 27–37. https://doi.org/10.1145/1289612.1289615
[10] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. 2016. *STREAM: The Stanford Data Stream Management System*. Springer Berlin Heidelberg, Berlin, Heidelberg, 317–336. https://doi.org/10.1007/978-3-540-28608-0_16
[11] A. Arasu, S. Babu, and J. Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z
[12] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. 2013. RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems* (Arlington, Texas, USA) *(DEBS '13)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2488222.2488257
[13] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing.. In *CIDR*. 363–374.
[14] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication steps for parallel query processing. *Journal of the ACM (JACM)* 64, 6 (2017), 1–58.
[15] M. Blount, M. Ebling, J. Eklund, A. James, C. Mcgregor, N. Percival, K. Smith, and D. Sow. 2010. Real-Time Analysis for Intensive Care: Development and Deployment of the Artemis Analytic System. 29 (05 2010), 110–8.
[16] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. [n.d.]. Cayuga: A High-performance Event Processing Engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China). ACM, 1100–1102. https://doi.org/10.1145/1247480.1247620
[17] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. 2009. Distributed event stream processing with non-deterministic finite automata. In *DEBS*, A. S. Gokhale and D. C. Schmidt (Eds.). ACM.
[18] P. Brown. 2013. *Architecting Complex-Event Processing Solutions with TIBCO* (1st ed.). Addison-Wesley Professional.
[19] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World.. In *CIDR*.
[20] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD Rec.* 29, 2 (2000), 379–390. https://doi.org/10.1145/335191.335432
[21] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 63–78.
[22] Diane J Cook. 2010. Learning setting-generalized activity models for smart spaces. *IEEE intelligent systems* 2010, 99 (2010), 1.
[23] G. Cugola and A. Margara. 2010. TESLA: a formally defined event specification language. In *DEBS*. ACM, 50–61.
[24] G. Cugola and A. Margara. 2012. Complex Event Processing with T-REX. *J. Syst. Softw.* 85, 8 (2012), 1709–1728. https://doi.org/10.1016/j.jss.2012.03.056
[25] G. Cugola and A. Margara. 2012. Low latency complex event processing on parallel hardware. *J. Parallel and Distrib. Comput.* 72, 2 (2012), 205–218.
[26] G. Cugola and A. Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (2012), 62 pages. https://doi.org/10.1145/2187671.2187677
[27] M. Dayarathna and S. Perera. 2018. Recent Advancements in Event Processing. *ACM Comput. Surv.* 51, 2, Article 33 (Feb. 2018), 36 pages. https://doi.org/10.1145/3170432
[28] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).
[29] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. [n.d.]. Towards Expressive Publish/Subscribe Systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*. Springer-Verlag, 627–644. https://doi.org/10.1007/11687238_38
[30] Christos Doulkeridis and Kjetil Nørvåg. 2014. A survey of large-scale analytical query processing in MapReduce. *The VLDB journal* 23, 3 (2014), 355–380.
[31] O. Etzion and P. Niblett. 2010. *Event Processing in Action*. Manning Publications Co.
[32] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, and M. Mock. 2017. Issues in complex event processing: Status and prospects in the Big Data era. *Journal of Systems and Software* 127 (2017), 217 – 236. https://doi.org/10.1016/j.jss.2016.06.011
[33] Bugra Gedik, Habibe G Özsema, and Özcan Öztürk. 2016. Pipelined fission for stream programs with dynamic selectivity and partitioned state. *J. Parallel and Distrib. Comput.* 96 (2016), 106–120.
[34] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. 2008. On Supporting Kleene Closure over Event Streams.. In *ICDE*. IEEE, 1391–1393.
[35] Yeye He, Siddharth Barman, Di Wang, and Jeffrey F Naughton. 2011. On the complexity of privacy-preserving complex event processing. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 165–174.
[36] M. Hill, M. Campbell, Y. C. Chang, and V. Iyengar. 2008. Event detection in sensor networks for modern oil fields. In *DEBS (ACM International Conference Proceeding Series)*, Vol. 332. ACM, 95–102.
[37] M. Hirzel. 2012. Partition and Compose: Parallel Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems* (Berlin, Germany) *(DEBS '12)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/2335484.2335506
[38] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412
[39] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 46.
[40] I. Kolchinsky and A. Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *PVLDB* 11, 11 (2018), 1346–1359.
[41] I. Kolchinsky and A. Schuster. 2018. Join Query Optimization Techniques for Complex Event Processing Applications. *PVLDB* 11, 11 (2018), 1332–1345.
[42] I. Kolchinsky, A. Schuster, and D. Keren. 2016. Efficient Detection of Complex Event Patterns Using Lazy Chain Automata. *CoRR* abs/1612.05110 (2016). arXiv:1612.05110 http://arxiv.org/abs/1612.05110
[43] I. Kolchinsky, I. Sharfman, and A. Schuster. 2015. Lazy Evaluation Methods for Detecting Complex Events. In *DEBS* (Oslo, Norway). ACM, 34–45. https://doi.org/10.1145/2675743.2771832
[44] Donald Kossmann. 2000. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)* 32, 4 (2000), 422–469.
[45] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: a survey. *AcM sIGMoD Record* 40, 4 (2012), 11–20.
[46] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 811–825.
[47] Mo Liu, Elke Rundensteiner, Dan Dougherty, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 2011. High-performance nested CEP query processing over event streams. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 123–134.
[48] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. 2011. E-Cube: Multi-dimensional Event Sequence Analysis Using Hierarchical Pattern Query Sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. ACM, New York, NY, USA, 889–900. https://doi.org/10.1145/1989323.1989416
[49] R. Mayer, B. Koldehofe, and K. Rothermel. 2015. Predictable Low-Latency Event Detection With Parallel Complex Event Processing. *IEEE Internet of Things Journal* 2, 4 (Aug 2015), 274–286. https://doi.org/10.1109/JIOT.2015.2397316
[50] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 161–173.
[51] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 54–65.
[52] Y. Mei and S. Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference*. ACM, 193–206.
[53] Gabriele Mencagli, Massimo Torquati, Marco Danelutto, and Tiziano De Matteis. 2017. Parallel continuous preference queries over out-of-order and bursty data streams. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2608–2624.
[54] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. 2016. When two choices are not enough: Balancing at

scale in distributed stream processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 589–600.

[55] M Tamer Özsu and Patrick Valduriez. 1996. Distributed and parallel database systems. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 125–128.

[56] Karl Pearson. 1895. VII. Note on regression and inheritance in the case of two parents. *proceedings of the royal society of London* 58, 347-352 (1895), 240–242.

[57] O. Poppe, C. Lei, S. Ahmed, and E. Rundensteiner. 2017. Complete Event Trend Detection in High-Rate Event Streams. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 109–124. https://doi.org/10.1145/3035918.3035947

[58] E. Rabinovich, O. Etzion, and A. Gal. 2011. Pattern Rewriting Framework for Event Processing Optimization. In *Proceedings of the 5th ACM International Conference on Distributed Event-based Systems* (New York, New York, USA). ACM, 101–112. https://doi.org/10.1145/2002259.2002277

[59] M. Ray, C. Lei, and E. A. Rundensteiner. 2016. Scalable Pattern Sharing on Event Streams. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 495–510. https://doi.org/10.1145/2882903.2882947

[60] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. 2016. Superlinear speedup in HPC systems: Why and when?. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 889–898.

[61] Nicoló Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, and Bruno Sericola. 2016. Online scheduling for shuffle grouping in distributed stream processing systems. In *Proceedings of the 17th International Middleware Conference*. 1–12.

[62] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)* 52, 2 (2019), 1–37.

[63] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.

[64] Ravindra S. and Dayarathna M. 2015. Distributed Scaling of WSO2 Complex Event Processor. (2015). https://wso2.com/library/articles/2015/12/article-distributed-scaling-of-wso2-complex-event-processor/.

[65] Omran Saleh, Heiko Betz, and Kai-Uwe Sattler. 2015. Partitioning for scalable complex event processing on data streams. In *New Trends in Database and Information Systems II*. Springer, 185–197.

[66] N. P. Schultz-Møller, M. M., and P. R. Pietzuch. 2009. Distributed complex event processing with query rewriting. In *DEBS*. ACM.

[67] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. 2011. Siddhi: A Second Look at Complex Event Processing Architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments* (Seattle, Washington, USA) *(GCE '11)*. ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/2110486.2110493

[68] Yuzhe Tang and Bugra Gedik. 2012. Autopipelining for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (2012), 2344–2354.

[69] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 625–636.

[70] YH Wang, Kening Cao, and XM Zhang. 2013. Complex event processing over distributed probabilistic event streams. *Computers & Mathematics with Applications* 66, 10 (2013), 1808–1821.

[71] L. Woods, J. Teubner, and G. Alonso. 2010. Complex event detection at wire speed with FPGAs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 660–669.

[72] E. Wu, Y. Diao, and S. Rizvi. 2006. High-performance complex event processing over streams. In *SIGMOD Conference*. ACM, 407–418.

[73] Fuyuan Xiao, Cheng Zhan, Hong Lai, Li Tao, and Zhiguo Qu. 2017. New parallel processing strategies in complex event processing systems with data streams. *International Journal of Distributed Sensor Networks* 13, 8 (2017), 1550147717728626. https://doi.org/10.1177/1550147717728626 arXiv:https://doi.org/10.1177/1550147717728626

[74] H. Zhang, Y. Diao, and N. Immerman. 2014. On Complexity and Optimization of Expensive Queries in Complex Event Processing. In *SIGMOD*. 217–228.

[75] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 1213–1224. https://doi.org/10.1109/ICDE.2017.166

[76] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2020. Hardware-conscious stream processing: A survey. *ACM SIGMOD Record* 48, 4 (2020), 18–29.

[77] Q. Zhou, Y. Simmhan, and V. K. Prasanna. 2012. Incorporating Semantic Knowledge into Dynamic Data Processing for Smart Power Grids. In *International Semantic Web Conference (2) (Lecture Notes in Computer Science)*, Vol. 7650. Springer, 257–273.

[78] Nikolaos Zygouras, Nikos Zacheilas, Vana Kalogeraki, Dermot Kinane, and Dimitrios Gunopulos. 2015. Insights on a scalable and dynamic traffic management system.. In *EDBT*. 653–664.

APPENDIX A - FORMAL STATEMENTS AND PROOFS

Here we formally state and prove the claims presented in Sections 3.3 and 3.4.

Theorem 1 (Execution unit allocation is relative to agents' load). *Let $|U|$ be the total number of execution units and let $load_i$ denote the load on the i^{th} agent. Then, the number of execution units allocated to the i^{th} agent $|U_i|$ is as follows:*

$$|U_i| = \frac{load_i}{\sum_{j=1}^{m} load_j} \cdot |U|,$$

where m is the total number of agents.

Proof. As we consider homogeneous execution units, each unit can perform the same number of computational and synchronization actions. Thus, the relative load on an agent, defined as the absolute load divided by the total load, is the only factor that should be taken into consideration when allocating execution units to agents. Substituting the total load with the sum of the loads of all agents load on all agents, we infer

$$|U_i| = \frac{load_i}{\sum_{j=1}^{m} load_j} \cdot |U|.$$

□

Theorem 2 (Calculation of partial match arrival rate from preceding agent). *Let $A_2, A_3 ..., A_m$ an ordering of agents in HYPERSONIC (i.e., such A_{i-1} passes its output to A_i for all $i \leq m$) where m is the total number of agents. Let m_i be the average number of partial matches that arrive at agent A_i from agent A_{i-1} per time unit. Then, m_i can be calculated using e_i, the average rate of events arriving at agent A_i, s_i, the state selectivity, and W, the time window of the query, as follows:*

$$m_i = \begin{cases} e_1, & i = 2; \\ 2m_{i-1}e_{i-1}s_{i-1}W & i > 2. \end{cases}$$

Proof. The first agent in the agent sequence receives events of the second event type as its input stream, has no preceding agent and receives partial matches of size one directly from the input stream. We denote it as A_2 to represent that agent A_i indeed receives events of type E_i as its event input. A_2 match stream solely composed of the events of type E_1, thus its associated data arrival rate equals to e_1. Any agent A_i such that $i > 2$ receives a stream of partial matches from its preceding agent A_{i-1} which receives partial matches at a rate of m_{i-1} and events at a rate of e_{i-1} (both by definition). For each partial match received, A_{i-1} compares it with every event in EB_{i-1} and outputs a fraction of the event-match combinations proportional to the state selectivity s_{i-1}. Thus, the average rate of partial matches created by combining incoming partial matches with buffered events is $m_{i-1}|EB_{i-1}|s_{i-1}$. Similarly, incoming events are compared with every stored partial match in MB_{i-1} and thus partial matches are created at a rate of $e_{i-1}|MB_{i-1}|s_{i-1}$ due to incoming events. The size of an event buffer or match buffer is determined by the rate of incoming items and their time of removal from the buffer. Items exist in the buffers for roughly W time units until they are removed, hence the size of an event buffer is $|EB_{i-1}| = e_{i-1}W$, similarly $|MB_{i-1}| = m_{i-1}W$. By replacing these values with the rate of created partial matches, we receive:

$$m_{i-1}|EB_{i-1}|s_{i-1} + e_{i-1}|MB_{i-1}|s_{i-1} =$$

$$m_{i-1}e_{i-1}s_{i-1}W + e_{i-1}m_{i-1}s_{i-1}W =$$

$$2m_{i-1}e_{i-1}s_{i-1}W$$

□

Theorem 3 (Calculation of the load imposed by synchronization actions). *Let $sync_i$ denote the total load of synchronization actions performed during a time unit, it is calculated as follows:*

$$sync_i = acc_i b_i + q_i m_{i+1} = \frac{(e_i + m_i)nb_i}{2m} + q_i m_{i+1},$$

where acc_i denotes the total number of worker accesses of the buffers of the opposite role. With the necessary assumption that each agent receives n/m workers and that exactly half are assigned as event workers and the other half are assigned as match workers, acc_i is calculated as follows:

$$acc_i = e_i \cdot \frac{n}{2m} + m_i \cdot \frac{n}{2m} = \frac{(e_i + m_i)n}{2m}.$$

Proof. First, we will calculate acc_i. Each incoming event triggers an access to every match buffer fragment in that agent. Under the assumption that each agent receives the same n/m workers and splits them equally between event workers and match workers, we get that there are $n/2m$ match workers and thus each incoming event has to perform $n/2m$ buffers accesses. Similarly, each incoming partial match has to perform $n/2m$ buffers accesses to event buffer fragments. Thus, the total number of buffer accesses for an agent A_i is:

$$acc_i = e_i \cdot \frac{n}{2m} + m_i \cdot \frac{n}{2m} = \frac{(e_i + m_i)n}{2m}.$$

The total load of synchronization actions for A_i is composed of the buffer access load, which is $acc_i b_i$, and the cost of writing to the output queue. As m_{i+1} is number of output queue write operations actions, the output queue cost is $q_i m_{i+1}$. Thus, the total synchronization cost is:

$$sync_i = acc_i b_i + q_i m_{i+1} = \frac{(e_i + m_i)nb_i}{2m} + q_i m_{i+1}.$$

□

Theorem 4 (Calculation of the rate of partial matches outputted by a Kleene closure agent). *Let A_i be an agent implementing the Kleene closure operator and let m_i^{prev} be the rate of partial matches arriving from the preceding agent. Then, m_i, the rate of partial matches outputted by A_i, is calculated as follows:*

$$m_i = m_i^{prev}\left(1 + \sum_{j=1}^{e_i W}\left((e_i)^j (s_i)^j W^j\right)\right)$$

Proof. A_i receives partial matches via a self-loop as well as from the previous agents. m_i is comprised of partial matches created due to input arriving from the previous agent(m_i^{prev}) and from the self-loop. Thus,

$$m_i = m_i^{prev} + \sum_{j=1}^{e_i W} m_i^{KC_j}$$

where the rate of incoming partial matches that arrive from the self-loop is denoted as $m_i^{KC_j}$, where j is the number of events of type E_i, the type of the incoming events in A_i. In other words, j is the number of times this partial match has "passed" through A_i. As stated in theorem 2, an arriving partial match triggers creation of $e_i s_i W$ new partial matches. Consequently, m_i^{prev} partial matches arriving from the previous agent result in

$$m_i^{KC_1} = m_i^{prev} e_i s_i W$$

new partial matches. These partial matches create additional partial matches at a rate of $m_i^{KC_1} e_i s_i W$, which in turn create new partial matches at a rate of $m_i^{KC_2} e_i s_i W$ and thus we infer that

$$m_i^{KC_j} = m_i^{KC_{j-1}} e_i s_i W$$

. Eliminating the recursion, we get

$$m_i^{KC_j} = m_i^{prev} (e_i)^j (s_i)^j W^j$$

. By summation over all values of j, we obtain the following expression:

$$m_i = m_i^{prev} + \sum_{j=1}^{e_i W} m_i^{KC_j} = m_i^{prev} \left(1 + \sum_{j=1}^{e_i W} \left((e_i)^j (s_i)^j W^j \right) \right).$$

□

Theorem 5 (Calculation of the average number of events in a partial match). *Let A_i be an agent implementing the Kleene closure operator and let $m_i^{KC_j}$ denote the rate of partial matches arriving from the self-loop that has exactly j events of the type E_j. Additionally, let a_i be the average number of events in a partial match inside the match buffer MB_i. Then, a_i is calculated recursively as*

$$a_i = a_{i-1} + \sum_{j=1}^{e_i W} \frac{m_i^{KC_j} \cdot j}{\sum_{k=1}^{e_i W} m_i^{KC_k} + m_i^{prev}} + 1.$$

Proof. First we state that a Kleene closure agent adds to a partial match at least one event and up to $|EB_i|$ events, so that the total number of events is $a_i = a_{i-1} + 1 + X$ where X is the additional number of events due to traversals on the self-loop. To calculate this value, we need to first calculate P_j which is the probability a partial match stored at a Kleene closure agent has j events of the type processed by this agent (which is only possible after j traversals of the self-loop). The total number of partial matches stored at a time unit is:

$$S_{total} = \sum_{k=1}^{e_i W} m_i^{KC_k} + m_i^{prev},$$

that is, the sum of all the incoming partial matches from the previous agent as well as from the self-loop. As $m_i^{KC_j}$ indicates the rate of partial matches created with exactly j events of the type processed by A_i, we infer $P_j = \frac{m_i^{KC_j}}{S_{total}}$. All that remains is to add the P_j to a_i weighted according to the number of additional events j, resluting in the following expression:

$$a_i = a_{i-1} + \sum_{j=1}^{e_i W} P_j \cdot j + 1 = a_{i-1} + \sum_{j=1}^{e_i W} \frac{m_i^{KC_j} \cdot j}{\sum_{k=1}^{e_i W} m_i^{KC_k} + m_i^{prev}} + 1.$$

□

Theorem 6 (Calculation of the memory consumption of HYPERSONIC). *Let v_i denote the average size of an event of the type handled by agent A_i and let p be a system-wide constant denoting the size of an event pointer. Then, the memory consumption of HYPERSONIC is calculated as follows:*

$$\sum_{i=1}^{n} \left(e_i v_i W + \sum_{j=1}^{i-1} e_j v_j W + (e_i W + m_i a_i W) p \right).$$

Proof. Each HYPERSONIC agent A_i utilizes three data structures, namely EB_i, MB_i and AGB_i. The total memory consumption is thus the sum of the sizes of these buffers over all agents. AGB_i stores all events entering an agent while EB_i and MB_i only store pointers. The events located in AGB_i arrive with a rate of e_i. Each event is of size v_i and is stored for a period of W. AGB_i also stores all the events that arrive on the match stream as parts of partial matches and thus has to store up to $e_j W$ events for every agent A_j that precedes A_i, for a total size of $\sum_{j=1}^{i-1} e_j v_j W$.

The size of the event and match buffers is given by $(|EB_i + |MB_i|a_i)p$, that is, the total number of pointers in EB_i and MB_i multiplied by the constant system-wide pointer size. A partial match has a_i events on average and hence the number of pointers is also multiplied by this number. Summing over all three buffers and over all agents, we get:

$$\sum_{i=1}^{n} (|AGB_i| + (|EB_i| + |MB_i|a_i)p) = .$$

$$\sum_{i=1}^{n} \left(e_i v_i W + \sum_{j=1}^{i-1} e_j v_j W + (e_i W + m_i a_i W) p \right).$$

□