Information Systems 88 (2020) 101442

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Network-wide complex event processing over geographically distributed data sources

Ioannis Flouris [a], Nikos Giatrakos [b,a,*], Antonios Deligiannakis [b,a], Minos Garofalakis [b,a]

[a] School of Electrical and Computer Engineering, Technical University of Crete, University Campus., 73100 Chania, Greece
[b] ATHENA Research and Innovation Centre, Artemidos 6 & Epidavrou, 15125 Athens, Greece

A R T I C L E I N F O

Article history:
Received 18 December 2018
Received in revised form 25 May 2019
Accepted 21 September 2019
Available online 10 October 2019
Recommended by Matthias Weidlich

A B S T R A C T

In this paper we focus on Complex Event Processing (CEP) applications where the data is generated by sites that are geographically dispersed across large regions. This geographic distribution, combined with the size of the collected data, imposes severe communication and computation challenges. To attack these challenges, we propose a novel approach for geographically distributed CEP, which combines algorithmic and systems contributions. At an algorithmic level, our work combines an in-network processing approach, which pushes parts of the processing (i.e., CEP operators) towards the sources of their input events, along with a push–pull paradigm, in order to reduce the amount of communicated events. We present optimal (but computationally expensive) solutions which seek to minimize the maximum bandwidth consumption given input latency constraints for detecting events, as well as efficient greedy and heuristic algorithmic variations for our problem. At a systems level, we explain how existing CEP engines can support, with minimal modifications, our algorithms. Our experimental evaluation, using mainly real datasets and network topologies, demonstrates that the power of our techniques lies in the combination of the in-network with the push–pull paradigm, thus allowing our algorithms to significantly outperform related centralized push–pull or conventional in-network processing approaches.

1. Introduction

Complex event processing (CEP) has become an essential tool for quickly detecting events of interest in Big Data systems and applications [1–3]. The languages of CEP systems allow the definition of complex patterns and conditions that need to be fulfilled over the input *primitive events* (PEs), so that a *complex event* (CE) of interest is detected. Primitive events may, for instance, involve a suspicious mobile phone call or credit card transaction, or an abnormal sensor temperature measurement. An application defines CEP queries composed of operators such as [2,4,5] logical disjunctions (OR), conjunctions (AND), time ordered conjunctions (SEQ) of PEs and/or CEs, or aggregations. Each such query can be represented as a directed acyclic graph, termed as the *event detection graph* (EDG), in which PEs lack incoming edges.

CEP applications include [1,6], but are not limited to, network health monitoring, mobile and sensor networks, smart cities and Internet-of-Things (IoT) applications, computer clusters, smart energy grids, detection of security attacks, such as denial-of-service attacks, intrusions or fake identities. In the business sector, accounting, logistics, warehousing and stock trading applications are also included among the ever-growing areas of application. Many CEP applications are time-critical, requiring the detection of all interesting events as soon as possible. For example, attacks need to be detected in real time and imposing constraints on the maximum allowed latency for detecting an attack is often desired.

The initial CEP systems require the collection of all events at a central location (computer or data center — cluster) for processing. A large body of works [5,7–15] optimize CEP in a non-parallel, centralized setting [6]. More scalable systems employ parallel CEP approaches [12,16–22] to optimize CEP over clusters in local data centers mostly aiming at throughput maximization and elastic resource allocation [3]. In this work we term the centralized and parallel approaches described above as centralized, in the sense that they operate on a single site (machine or data center). However, such centralized processing systems are not scalable for massive Big Data and respective applications. First of all, in applications with truly Big Data, bandwidth is still an issue and the central site (data center) becomes a communication and computation bottleneck. Prior work [23] has pointed out that the maximum stream processing rate that can be achieved in centralized settings is network bound. Therefore, what is important

* Corresponding author at: School of Electrical and Computer Engineering, Technical University of Crete, University Campus., 73100 Chania, Greece.
E-mail addresses: gflouris@softnet.tuc.gr (I. Flouris), ngiatrakos@imis.athena-innovation.gr, ngiatrakos@softnet.tuc.gr (N. Giatrakos), adeli@imis.athena-innovation.gr, adeli@softnet.tuc.gr (A. Deligiannakis), minos@imis.athena-innovation.gr, minos@softnet.tuc.gr (M. Garofalakis).

https://doi.org/10.1016/j.is.2019.101442
0306-4379/© 2019 Elsevier Ltd. All rights reserved.

in massive scale CEP applications is to reduce communication in the network and control the network latency so as to loosen or better control this bound. Second, in many monitoring applications, most events are actually rather "useless", since interesting events occur rarely. For example, when monitoring for attacks or intrusions at a system, most of the time such attacks do not occur, or when looking for malfunctioning machines in data centers or such sensors in IoT applications, most of the time things are operating correctly. Thus, centrally transmitting and processing all events overburdens the central site with little benefit. Third, the transmission of events by generating sources often comes at a cost and should be avoided if possible. This is often true in cases of data collected by battery-powered sites (i.e., sensors or IoT devices), in which data transmission is a major cause of energy drain [24].

Overview of Approach and Relationship to Prior Work. In this paper we target this important type of CEP applications, where the data is generated by sites that are geographically dispersed across large regions. For example, network elements, IoT devices, energy grid applications, etc that span one or more countries. We, thus, look into algorithms to perform CEP in a distributed (across different sites) way, while respecting application latency requirements. Table 1 shows how our work uniquely (given prior work) combines features that are vital for modern geo-distributed, CEP Big Data platforms. More details follow in Section 2. Such features include:

[A] Algorithmic: our techniques are the first to effectively blend:

[A1] In-network operator placement: assigns the evaluation of CEP operators to sites that are close to the sources of relevant input events, so as to limit data transmission and to speed up event detection; and

[A2] CEP-tailored forwarding (push–pull): this is a set of lazy evaluation strategies that further reduce communicated events by prioritizing transmission of frequent events conditional upon the occurrence of rarer input events of the same operator.

[B] Service-Oriented: compliance with Service Level Agreements (SLAs) is critical in public, hybrid cloud settings. Our techniques seek to optimize geo-distributed CEP based on a **constrained, bi-criteria optimization** problem including pay-as-you-go communication cost ([B1]) and network latency related Quality-of-Service (QoS) ([B2]). Several related techniques, do not (and cannot) support the constrained optimization of both criteria (Table 1):

[B1] Network Pricing: We provide minimization of intra-query and multi-query communication load produced by CEP analytics tasks per query client and under network latency-constraints. Using Stream Analytics [25] in Microsoft Azure [26], Apache Storm (and thus EsperOnStorm [27], IBM ProtonOnStorm [28]) in Microsoft HDInsight [29], Apache Spark and Flink (CEP) in Google Cloud [30,31], or WSO2 [32] in Amazon AWS [33] entails a pay-as-you-go network pricing model (i.e., the monetary cost depends on communicated data volumes). Minimizing the communication cost under this model interprets to fair customer charges.

[B2] Network-Aware QoS criteria: Network latency-constrained optimization is incorporated in our techniques as it allows compliance with QoS criteria defined on SLAs or on requirements of time critical applications. Network (instead of computational) latency-constrained optimization is often not supported as shown in Table 1.

[C] System oriented: Our algorithms are incorporated in a real-world streaming multi-cloud platform, which has been demonstrated in FERARI [34,35].

Both [A1], [A2] are prerequisites for CEP optimization and our approach is the first to provide algorithms that blend both of them for complex EDGs. Fig. 1 depicts the intuition of the Algorithmic feature category [A]. The top part of the figure contains the input EDG. The bottom part of Fig. 1 shows sites $S_1 - S_{16}$, while ℓ_{i-j} on network links involve communication latency values among the sites. Pentagons show PEs detected at certain sites. Placing CEP operators close to their input event sources helps improving the detection latency, reduces bandwidth consumption, and does not overload a central processing site. Because of [A1], for instance, the AND and the leftmost SEQ operators of the EDG have been placed closer to the sources of their input events so that they are evaluated early and, thus, (a) only aggregate information is communicated further in the network, (b) event detection is sped up. Each operator may function using a push–pull paradigm [A2] (not depicted in the figure). Our evaluation shows that techniques using only [A1] or [A2] yield severely suboptimal solutions. From a service viewpoint (i.e., [B]), the aforementioned sub-optimality results in overcharges in the pay-as-you-go model. To justly lower charges as much as possible, but simultaneously abide by QoS constraints, a properly modeled optimization problem is needed. We provide a bi-criteria, constrained optimization model that supports [B1], [B2] incorporating [A1], [A2]. Criterion [C] strengthens our claims for the true applicability of our approaches. The development of a real system tested in real application scenarios implied by [C] is a quite important feature. As a result of the development of a streaming multi-cloud platform that uses our algorithms [34,35], our work elaborates on real system implementation aspects and enhances its contributions along these lines. On the contrary, prior techniques such as [4,36] may, for instance, opt for [A2] but because of lacking [C] they do not comment on architectural aspects or provide implementation hints over real networks, as we do.

Contributions. Our contributions can be summarized as follows:

- We define the problem of geographically distributed CEP as an operator placement problem, where each operator can be instructed to operate in a push–pull fashion.
- We present both optimal (but computationally expensive) solutions, as well as efficient greedy and heuristic variations, which seek to minimize the maximum bandwidth consumption given input latency constraints for detecting events. Our greedy and heuristic variants combine different optimizations that can be easily enabled/disabled.
- At a systems level, we explain how (we believe many) existing CEP engines can be easily modified to support our algorithms. Our suggested modifications allow us to transform existing CEP engines to engines that support geographically distributed CEP.
- We present a detailed experimental evaluation, using a variety of setups, mainly real data and real topologies. Our analysis demonstrates that our solutions significantly outperform central push–pull [4] or conventional in-network processing approaches [37].

The rest of the manuscript is organized as follows. In the upcoming section we discuss related work. Section 3 presents the utilized event data model, the supported operators, the event detection graph and the network of sites that are input to our optimization problem. We further present the push–pull rationale in detail. In Section 4 we outline the statistics that are necessary to our optimization algorithms for devising a preferable plan for executing the posed geo-distributed CEP analytics tasks and provide the formal Push–pull Enhanced CEP Operator Placement (PECOP) problem definition. We introduce Dynamic Programming (DP) and Exhaustive Search (ES) algorithms that provide optimal solutions to PECOP in the absence (DP) or existence (ES) of events that are shared among CEP operators, in Section 5. Besides the computationally expensive DP and ES, Section 5 presents greedy and heuristic algorithms for quickly deriving efficient solutions to PECOP in practical scenarios. Section 6 details how CEP systems operating over state-of-the-art Big Data platforms need to be minimally modified to support our algorithms for geo-distributed CEP. Experimental results are demonstrated in Section 7, while Section 8 includes concluding remarks.

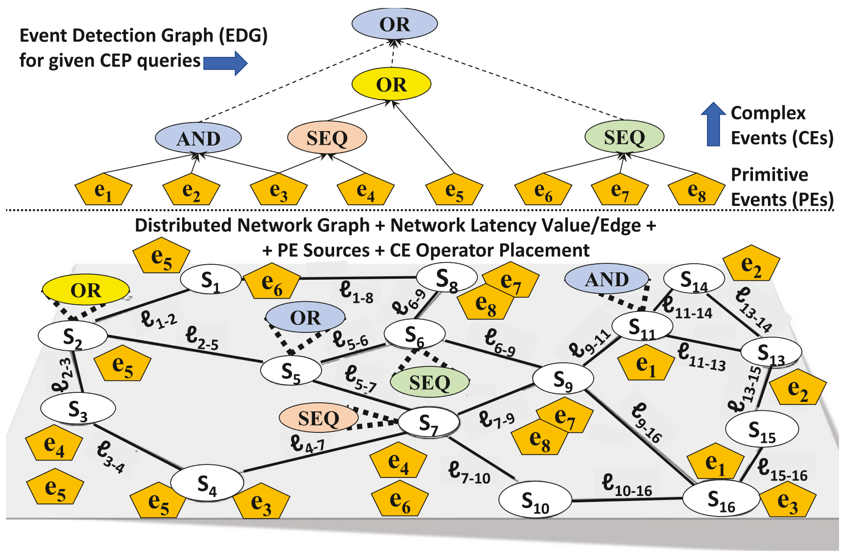

Event Detection Graph (EDG) for given CEP queries
OR
OR
AND
SEQ
SEQ
Complex Events (CEs)
Primitive Events (PEs)
Distributed Network Graph + Network Latency Value/Edge + + PE Sources + CE Operator Placement

Fig. 1. Overview of our Optimization Framework. The top part of the figure shows an exemplary event detection graph composing one or more CEP queries. Complex events are the output of AND,OR,SEQ operators, while primitive events are noted with pentagons on the figure. The bottom part of the figure illustrates a physical network of sites $\{S_1, \ldots, S_{16}\}$. Edges correspond to communication links among them, while network latency values are noted on the links among pairs of sites (S_x, S_y) as ℓ_{x-y}. The bottom part of the figure also illustrates a possible in-network placement of the CEP operators so that they are evaluated near the sources of the relevant, primitive events and, thus, only aggregate event information is further forwarded in the network, finally reaching the query source. A push–pull strategy (not shown in the figure) along with in-network placement is applied by our framework.

2. Related work

2.1. Overview and comparative analysis

Table 1 shows that our techniques are the first to compose a solution that addresses criteria [A], [B], [C], the importance of which was explained in the introduction. From a purely algorithmic viewpoint, that is [A], the task of coming up with a proper geo-distributed event query execution plan is non-trivial. First, one needs to examine every possible in-network placement of each operator on par with a number of alternative push–pull strategies that may be applicable. What is more, each pair of (push–pull strategy, in-network placement) for a given operator affects the push–pull and placement choices not only of all directly connected operators, but also of all EDG paths that pass through that given operator. Some of the efforts outlined in this section attempt to tackle specific aspects of the problem, such as [A2] [4] or restrict [A] to a single operator instead of an EDG and its aforementioned paths [36]. Hence, they do not confront the greater challenge of the generic problem setup addressed by our work, as described above.

Geographically Distributed CEP [64]: There is a pair of works that are more closely related to ours. The first is the work of Comet [36], which proposes to combine in-network operator placement with a push–pull mechanism for communication-efficient and latency-aware CE detection over mobile networks. Nonetheless, the approach is restricted to CEP queries with only one operator per query. This is only a very special case that is covered by the algorithms proposed in our work. The impact of this restriction is that Comet cannot address the service-oriented criterion [B] and its sub-criteria in the general case. Moreover, contrary to our approach, Comet is not incorporated or tested in a real world distributed system. On the other hand, our work does not account for mobile networks, which would be an interesting direction for future work.

The second technique that is closer in spirit to ours is that of [4], which applies a push–pull paradigm, but performs central event data collection at a fixed site (Table 1). [4] falls short with respect to criterion [A1] and [C]. In our evaluation we significantly outperform [4] (termed *CPP*). Geo-distributed CEP appears in Hermes [40], PADRES [45], Cordies [44] and DHCEP [41]. Hermes uses a Distributed Hash Table (DHT) to determine in-network operator placement. DHTs simply minimize the hop count and are severely suboptimal regarding network latency or bandwidth [37]. Thus, respective techniques fall short with respect to criteria [A2], [B]. PADRES and Cordies fall short with respect to [A1], [A2], [B], as they neither take into account system specific statistics nor propose specific operator placement strategies. DHCEP, uses a network usage metric during its optimization process. Network usage is the sum of products of *dataRate* × *latency* on communication links. However, such a blended metric in DHCEP (and [37, 46,47] discussed shortly) does not allow for latency-constrained optimization and network pricing separately ([B]).

Broader Distributed Stream Processing (DSP) [65]: To avoid repetition, we state that all efforts discussed here fall short with respect to [A2] (Table 1) and are represented by the *GRIN* (GReedy IN-network placement without push–pull) approach in our experiments, which is a best case scenario satisfying [B] and only lacking [A2]. The early work of [42] is DHT-based. As we already argued, such techniques cannot abide by [B]. The seminal work of SBON [37] employs a metric similar to network usage. Thus, as previously described, it cannot support constrained optimization per metric and thus [B]. The same hold for efforts [46,47] that employ a similar utility or usage metric. Moreover, although [46, 47] and [66] also try to support latency constraints, this comes after (in [46,47]) or before (in [66]) having determined operator placement. [38] neglects operator sharing falling short in [A2], [B1], despite considering the intra-query communication burden.

JetStream [49] restricts itself to obvious in-network placement on source nodes or nearest site of relevant data presence, essentially lacking [A1]. Iridium [48] examines data migration instead of operator placement to optimize query response latency. Geode [57] purely focuses on minimizing bandwidth cost and does not account for criterion [B2]. Similarly, SQPR [56] accounts for computational, instead of network ([B2]), latency.

The works in [50–55] aim at optimal component composition such that load distribution is achieved subject to various function, resource, and QoS constraints inside clusters. Nonetheless,

Table 1
Summarizing features of this work vs related techniques.

Feature category	Algorithmic		Service		System
Related work	A1	A2	B1	B2	C
Akdere et al [4]	✗	✓	✓	✓	✗
Cardellini et al [38], SODA [39]	✓	✗	✗	✓	✓
SBON [37], Hermes [40], DHCEP [41], SAND [42], SPADE [43]	✓	✗	✗	✗	✓
Cordies [44], PADRES [45]	✗	✗	✗	✗	✓
Kumar et al [46], Rizou [47]	✓	✗	✗	✗	✗
Iridium [48], JetStream [49]	✗	✗	✓	✓	✓
Borealis [50], Medusa [51], Chatzistergiou et al [52], Gu et al [53], Flux [54], Zhou et al [55]	✓	✗	✗	✓	✓
SQPR [56], Geode [57]	✓	✗	✓	✗	✓
Chatzimilioudis et al [58], Srivastava et al [59], Ying et al [60]	✓	✗	✗	✗	✗
Amini et al [61], Benzing et al [62]	✗	✗	✗	✓	✗
Repantis et al [63]	✗	✗	✗	✓	✓
Comet [36]	✓	✓	✗	✗	✗
This work	✓	✓	✓	✓	✓

apart from [A2], [50–53,55] all neglect [B1]. In [53,55] this is due to examining purely network-oriented metrics (congestion, communication performance ratio) to balance the load, while in Medusa [51], Borealis [50], Flux [54] and [52] the focus is to primarily balance the load and minimize usage of available resources while doing so. Similarly, [61] focuses on weighted throughput without any latency constraint, while [62,63] seek to maximize availability while fulfilling QoS and bandwidth constraints, respectively, lacking [A1], [B1] and [C]. Likewise, SODA [39] performs placement for load balancing while also taking into account resource matching and licensing constraints. SPADE [43] simply allows to guide placement by specifying host pools.
Sensor Networks: Works such as [58–60], aim at reducing communication and energy costs. Such techniques are mostly tailored for tree-like network organizations [58,59], tree-like query graphs [60] and lack [A2], [B1] and [B2].

2.2. Other related work: Push–pull paradigm and beyond

With respect to criterion [A2], the push–pull paradigm has been adopted in broader CEP contexts to optimize centralized [67] or parallel (within one site) [21] event query execution. In both cases, the paradigm retains its ability of reducing the unnecessary processing of partial pattern matches that are unlikely to produce a full match unless rare events occur.

Choosing a push–pull strategy essentially involves rewriting the CEP query by reordering the input of an operator so as to prioritize transmission of frequent events conditional upon the occurrence of rarer input events. This is not the only kind of rewriting that is admissible by a CEP query. The work of [12] is one of the first to formalize query rewriting for optimizing CEP in centralized and parallel settings. It presents an assertion-based pattern rewriting framework that aims at splitting patterns into disconnected components that can be independently processed. The acquisition of the disconnected components is achieved through the following steps: (i) the pattern is converted into conjunctive normal form, (ii) a variable dependency graph is created to recognize independent components and (iii) the pattern is split into maximal number of independent partitions which imply the finest granulation that can be performed. The work argues about the fact that the provided optimizations are related to code optimization and efficient state management, but providing disconnected components is also useful in parallelizing the processing of posed CEP queries. Such query rewriting-based optimizations are orthogonal to the techniques we present in this work. They can be applied before our techniques so as to rewrite the EDG in equivalent forms that can be input to our algorithms. Moreover, rewritings may be used within each site, for parallelization purposes, after our algorithms have assigned an operator to a site in the scope of a geo-distributed execution plan.

Beyond CEP, the push–pull mechanism has been used in view maintenance scenarios in data warehouses [68], data integration and mediation systems [69–71] as well as social networking applications [72]. In this context, a view is a virtual table defined by a query, while a materialized view pre-computes and stores the query result set in an actual table. A conventional view works in pull mode, in the sense that the content of the view is compiled from the sources, on demand, at query time. This ensures data freshness but increases the query execution time. A materialized view, on the other hand, can quickly provide a query answer at the cost of extra storage and a potential compromise of data freshness. To establish proper trade-offs between data freshness and query execution time, the push mode triggers the sources to update the materialized view on a regular basis, i.e., after a number of updates at data sources take place. Hybrid approaches are also applicable. For instance, the materialized view may get updated in push mode, but also pull updates, incrementally since the last push operation, at query time [71].

The use of the push–pull paradigm in the above settings relates to pro-actively (or not) building views so as to avoid information or time lags at query time. On the contrary, our techniques attempt to incrementally evaluate a continuously running CEP query in steps and to prune unpromising steps (partial pattern matches), if possible. Contrary to the techniques cited above, the trade-offs introduced by the push–pull paradigm in our work engage communication cost and network latency instead of data freshness and query execution time. Nonetheless, for wide-area applications [70], the push update mode for materialized views involves increased communication cost since data updates may be sent irrespectively of whether a relevant query exists. Another important observation is that, in our work, sites that transmit (push) events are not required to cache them anymore (assuming reliable network delivery) while, for materialized views, data sources store respective data tuples anyway.

3. Preliminaries

We consider two graphs. The first graph is the event detection graph which represents the posed CEP queries. The nodes of this graph are either primitive or complex events. PEs and CEs are timestamped event occurrences. PEs correspond to nodes with no incoming graph edges. The rest of the nodes of the graph are CEs which correspond to CEP operators such as logical disjunctions

(OR); conjunctions (AND); time ordered conjunctions (SEQ); aggregations etc of PEs or/and CEs. These operators receive input from/may provide input to other CEs. The second graph is the physical network where nodes are sites and graph edges refer to communication links. Each PE is produced at a subset of sites which are the sources of this PE. These elements are formalized in Section 3.1. As discussed in the upcoming Section 4, given these elements along with statistics, the output of our techniques is a mapping of the event detection graph to the physical network of sites so that, some sites of the network undertake the evaluation of, one or more, CEP operators. The evaluation of a CEP operator assigned to a site may involve the application of the push–pull rationale introduced in Section 3.2.

3.1. Optimization problem input

Network of Sites. We assume a distributed setting represented by a graph $Net = (S, H)$. The graph contains a set of vertices $S = \{S_1, \ldots, S_{|S|}\}$ of $|S|$ cardinality, for available sites (including event sources and potential relay nodes) and a set H of undirected edges (our algorithms can trivially handle directed graphs as well). $\ell_{i-j}, \forall(S_i, S_j)$ denotes the communication latency between neighboring (connected through an edge) sites S_i, S_j.

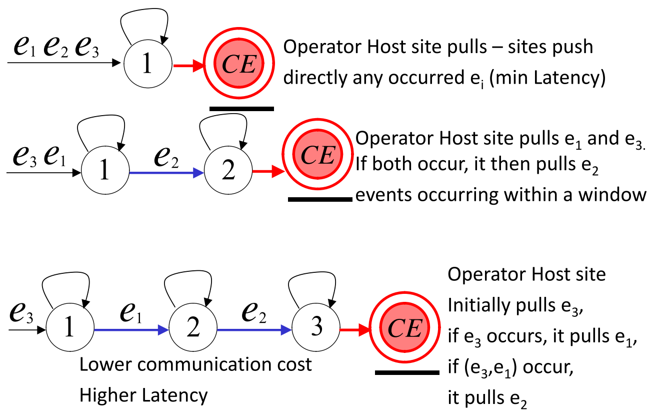
Definition 1 (*Event Data Model*)**.** A set $E = \{e_1, \ldots e_{|E|}\}$ of cardinality $|E|$ is the union of primitive event types that can be observed across the network. A tuple $\langle e_i, t\rangle$ represents an observed PE instantiating a particular event type e_i at time t. We use "event" and "event type" equivalently for simplicity.

Operators, CEP Queries and Event Detection Graph (EDG). Our algorithms support all major operator categories (and subcategories) included in [2] (Chapter 9, up to 9.3.3), excluding negation of events, i.e., requiring the input to a CEP operator be the non-occurrence of an event of a particular type. This operator list includes the following popular [2,4–6,15,21] operators:

- AND outputs a CE when all participating events occur.
- SEQ outputs a CE when all participating events occur in specified chronic sequence.
- OR outputs a CE whenever any participating event occurs.
- AGGREGATION operators (COUNT, SUM, etc.). These operators may accept a frequency parameter which specifies when they run, i.e. if an operator accepts a frequency parameter of 1, it runs every time a new input event is received. Aggregation operators can be used along with a specified threshold. Then, a CE is produced when the performed aggregation surpasses or falls below that threshold.

Operators may incorporate the definition of an event selection policy. Event selection policies define which events may be allowed in a pattern match by posing conditions about whether we are allowed to skip any events, deemed as "irrelevant", or not. Another type of policy is the event consumption policy which specifies the number of matches in which an event can participate. Operator definitions may include a selection policy among those defined in SASE [15,73] (having solved delayed event arrivals as described in Section 6), while we consider a `reuse` consumption policy. `reuse` means that an event can participate in all pattern matches where it is considered relevant. This is the default, and sometimes the only, consumption policy in popular CEP engines and prototypes [5,9,15,21,73–75].

Moreover, all operators that are admissible in our setup bear a time window W (which may differ across different operators), which expresses the maximum time period within which the



$e_1\, e_2\, e_3$
1
CE
Operator Host site pulls – sites push
directly any occurred e_i (min Latency)
$e_3\, e_1$
1
e_2
2
CE
Operator Host site pulls e_1 and e_3.
If both occur, it then pulls e_2
events occurring within a window
e_3
1
e_1
2
e_2
3
CE
Lower communication cost
Higher Latency
Operator Host site
Initially pulls e_3,
if e_3 occurs, it pulls e_1,
if (e_3,e_1) occur,
it pulls e_2

Fig. 2. Three alternative NFAs for different push–pull orderings for the AND operator of Fig. 1. Self-transiting edges are used for the occurrence of e_i's at the corresponding state. Colored edges represent events that may be pulled. Pulling events reduces bandwidth, but increases latency. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

input events of an operator should appear. The same is true even for aggregation operators.[1]

Operators are applied on PEs (i.e., e_i events) and/or receive input from other operators. Application CEP patterns are queries, composed of operators, that are submitted as input to our problem setting. Interdependencies among operators, i.e., one operator providing input to another, e.g. SEQ(AND(e_1, e_2), e_3), form a directed acyclic graph, hereafter termed as Event Detection Graph (EDG). Given an EDG operator Op_i, we refer to operators or PEs which contribute input to Op_i as its *upstream* operators, while *downstream* operators are the ones that receive input from Op_i. We term as *output operators* those operators that have no downstream operators (i.e., that do not provide their output as input to other operators). To an EDG we add a *top level OR operator* with incoming edges from all output operators.[2]

Definition 2 (*Event Detection Graph*)**.** A CEP query is represented by a directed acyclic graph, $EDG = (Op, X)$ with $|Op|$ nodes, $Op_i \in \{$SEQ, AND, OR$\}$ or Op_i is an aggregation operator, and $|X|$ edges $(Op_i \rightarrow Op_j)$ among operators that receive input from one another. For multi-query optimization purposes, output operators of the EDG are connected to a *top level OR operator.*

Fig. 1 provides an example of an EDG. At the top of Fig. 1, the top level OR with dotted incoming edges is added a posteriori unifying the underlying queries at a common sync point in the EDG. The leaves of the EDG are PEs, while intermediate nodes are distinct query operators forming a set Op, of $|Op|$ cardinality. Such operators correspond to complex events synthesizing information from PEs or other CEs. The edges of the EDG denote the use of CEs or PEs as input. Event types that are common input for two or more operators are referred to as *shared events*. The output of an operator $Op_i \in Op$ is always a CE.

[1] To understand why time windows may be useful in the case of aggregation operators, one reason is that there are several *pattern* policies that are associated with each operator, beyond selection and consumption policies, which help disambiguate the semantics of the output event and of the pattern matching process for the operator. One such pattern policy is the *evaluation* policy, which specifies when output events are produced by the operator. They can either be outputted incrementally (*Immediate evaluation policy*) or at the end of the time window (*Deferred evaluation policy*). For details on pattern policies, please refer to [2]. The above discussion also holds for the OR operator.

[2] Actually the addition of this top level OR operator is only needed in the case of EDGs with multiple output operators, in order to make the EDG rooted. However, its presence helps simplify our discussion hereafter, especially at the problem definition of Section 4.2.

3.2. The push–pull rationale

Instead of having every S_i transmit input for Op_i as soon as relevant events occur, i.e., setting all of them in *push mode*, some input events can initially be set to *pull mode* [4,21,67], being cached at the location where they are produced and transmitted only upon request. The pull mode thus increases the storage/buffering requirements at some sites and also increases the latency but, as explained shortly, it can avoid unnecessary communication. Reversely, the push mode may increase communication, but reduces latency since it instantly forwards events as soon as they occur.

The push–pull strategy is not feasible for any operator. It is feasible for operators requiring the participation of all of their input events, such as a SEQ and AND operator. A complex event cannot occur in these operators unless all relevant events have been observed somewhere in the network (and within a window W). Among the relevant input, some events may be less frequent than others. Then, until rare events occur, communicating frequent events (each of which expires after W time units) will only consume bandwidth with little potential to output a CE. On the other hand, if the rarest input event to the operator occurs, relevant frequent events within the window W should then be transmitted to allow for its evaluation. Other types of operators (i.e., OR or aggregation operators) cannot operate using a push–pull strategy, but their in-network placement can be optimized using our algorithms.

For this push–pull process, it holds that (a) it is performed in a number of steps, no more than the number of the input events, and that (b) the input events provided in each step admit different orderings. Each available option can be represented using a graph of states, whose number is equal to the number of steps, and edges marking the inputs for each step. We use Non-deterministic Finite Automata (NFAs) as the most popular structure [6] for that purpose. The number of available options for an operator with M inputs is equal to the Bell number $T(M)$ [76].

Example 1. Fig. 2 depicts three (but not all) alternative push–pull strategies for the AND operator of Fig. 1. These are also applicable for the SEQ operators, i.e., with PEs (e_6, e_7, e_8). Possible strategies include: (i) setting all e_1, e_2, e_3 in push mode: all such input events are immediately transmitted to our operator (top of the figure), (ii) choosing a two step push–pull strategy where e_3, e_1 are set to push mode, while e_2 is set to pull mode: e_3, e_1 are directly communicated, while e_2 is cached at the sites that generate it. If a pair e_3, e_1 occurs, then e_2 events are pulled, so that cached and new e_2 instances are communicated for a time window (middle of the figure), (iii) a three step push–pull strategy that can set e_3 in push mode, and e_1, e_2 to pull mode, but in two successive steps. As Fig. 2 states, the 1-state NFA (top) provides the minimum latency, while moving towards a 3-state NFA (bottom), increases latency while decreasing communication as transmission is conditional upon the occurrence of prerequisite PEs.

All edges in the NFA may have additional constraints. For example, Fig. 3 depicts a three-step NFA for the $\mathtt{SEQ}(\mathtt{e}_6, \mathtt{e}_7, \mathtt{e}_8)$ of Fig. 1. In order to ensure correctness, please note that conditions involving timestamps have been added to NFA edges, ensuring that $t_{e_6} < t_{e_7} < t_{e_8}$. □

4. Problem definition

In a nutshell, each solution provided by our algorithms is a mapping of *EDG* to *Net* (Section 3.1) combined with a particular push–pull strategy that optimizes the communication cost, given constraints on the maximum latency for detecting events.

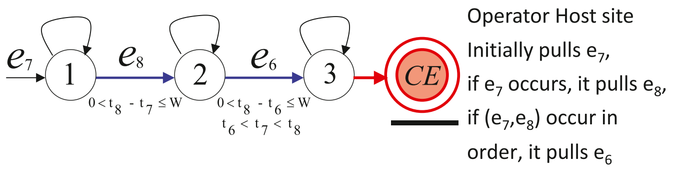


Fig. 3. A three-step NFA for the $\mathtt{SEQ}(\mathtt{e}_6, \mathtt{e}_7, \mathtt{e}_8)$ operator of Fig. 1. W denotes the time window of the operator, while t_i denotes the timestamp of event e_i. Constraints on NFA edges are present in order to ensure correctness.

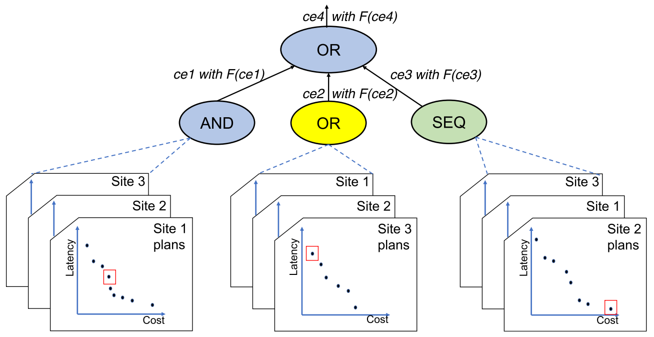


Fig. 4. Computing solutions for the top level OR operator of the EDG of Fig. 1 based on Pareto optimal solutions computed on input operators. For each CEP operator we examine its placement at different sites (along with a push–pull strategy where applicable). The folded cards below each operator AND, OR, SEQ illustrate the Pareto fronts that are formed for each such possible in-network placement of the corresponding operator at a site. To compute solutions for the top level OR operator, our algorithms consider combinations (triplets) of solutions from its input complex events and a particular combination is marked using squares around dots inside each visible folded card. Given this combination, the communication cost and network latency of the top level OR operator will be computed using the respective formulas included in Appendix.

4.1. Statistics and candidate solutions

Besides *EDG*, *Net* our algorithms further require the following simple statistics to be periodically collected over the network:

- **Local and Global Event Rates — Frequencies**: $f(e, S_i)$ stands for the number of e occurrences at S_i per time unit (equivalent to a Poisson parameter). The global frequency of a particular $e \in E$ is $F(e) = \sum_{S_i \in S} f(e, S_i)$.
- **Minimum Latency Paths**: Note that *Net* incorporates latency values ℓ_{i-j} for each *symmetric* communication link between sites S_i and S_j. Further exploiting ℓ_{i-j}s, we can easily compute the minimum latency paths $m\ell path_{i-j}$s (i.e., using an all pairs shortest path algorithm) for every S_i, S_j in the network and their associated latency $m\ell_{i,j} = \sum_{\forall (S_\phi, S_\psi) \in m\ell path_{i-j}} \ell_{\phi-\psi}$. The number of nodes in a respective minimum latency path is denoted by $hops(S_j, S_i)$.

Based on these simple statistics, one can easily compute the following statistics, used by our algorithms:

- **Hoped Frequency** $hf(e, S_i)$ is the cumulative event rate (frequency) for collecting all e data at S_i from all over the network: $hf(e, S_i) = \sum_{S_j \in S} f(e, S_j) \cdot hops(S_j, S_i)$.
- **Maximum Placement Latency** $mp\ell_{i,j}$ is the maximum latency required to gather e_i from all over the network (for every site S_b where e_i occurs) to site S_j: $mp\ell_{i,j} = \max_{\forall S_b \in S : e_i \mapsto S_b} \{m\ell_{j-b}\}$.

Based on these input parameters, our algorithms compute for each operator Op_i a set of candidate solutions. Each candidate solution corresponds to a set of decisions for (a) the placement of all operators in the subgraph of Op_i in the EDG, and (b) the push–pull strategy (NFA type) for each operator. For each candidate solution, our algorithms compute the communication cost and the latency for this solution (i.e., for all operators in the entire subgraph of

Op_i). Our algorithms differ in the number of placement decisions that they examine, on the way that they compute these solutions and on the number of solutions that they maintain per operator. However, for each operator our algorithms will compute a set of Pareto optimal solutions — one set for each potential placement of the operator (excluding our most Greedy variants, which maintain one solution per operator). Pareto optimal solutions are not dominated in all dimensions (i.e., cost, latency) from any other candidate solution in the same set.

The formulas for the estimation of the frequency of CEs and the network latency calculation can be found in Appendix. This is due to readability purposes and because our algorithms are equally applicable using as their basis either our own estimation formulas for the output rate of an operator, or using the analysis of other (i.e., [4]) works that use a push–pull approach.

Example 2. Fig. 4 depicts the process our algorithms will use to compute solutions. In this example, we consider the case of the top level OR operator of the EDG of Fig. 1, which receives as input the CEs $ce1$, $ce2$ and $ce3$ produced by the AND, OR and SEQ operators, correspondingly. For each input event we examine its placement at different sites. For the input event $ce1$, $ce2$, $ce3$ we have maintained a list of Pareto optimal solutions for each such placement. The event $ce2$ is output by the lower level OR of Fig. 1, for which only a 1-state NFA is admissible (Section 3.2), but since in Fig. 1 its input events include a SEQ operator (which admits push–pull - Example 1), a Pareto front is formed as well. To compute solutions for the top level OR operator, for a particular placement of it, our algorithms consider combinations (triplets) of solutions from its input events (a particular combination is depicted by squared dots), along with their frequencies ($F(ce1)$, $F(ce2)$, and $F(ce3)$) and the communication latencies between the sites.

Iterating over (not necessarily all) combinations of operator placements and NFA states, our algorithms will compute a set of candidate solutions for each operator per placement decision and then maintain the Pareto optimal subset of these candidate solutions per placement. If one (or more) of the input events was primitive (i.e., instead of $ce3$ we had as input the PE e_5), the process would only need to compute the cost of collecting e_5 to the location of the top level OR operator, with respect to e_5's position in the NFA chain. □

In our discussion hereafter, we will use the notation $CS^{\mu}_{i,j}$ to denote the set of candidate solutions for placing operator Op_i at site S_j and evaluating it using the μ-th possible NFA. $CS^{\mu}_{i,j}$ is a set because it contains candidate solutions created by different combinations of their input events (i.e., from different triplets of solutions in Fig. 4). We use $cs^{\mu}_{i,j} \in CS^{\mu}_{i,j}$ to denote a particular candidate solution in this set, and denote the cost and latency of this candidate solution as $cs^{\mu}_{i,j}.cost$ and $cs^{\mu}_{i,j}.lat$, respectively.

4.2. Formal PECOP problem definition

Definition 3 (*PECOP Problem Definition*). A Push–pull Enhanced CEP Operator Placement (PECOP) plan p is a placement of the operators Op of an EDG to sites in S with a specific NFA-based push–pull strategy per operator. Assume that Op_0 is the top level OR operator of the EDG and S_0 is the desired location of this operator. Given that, for the top level (and any) OR operator only a $\mu = M = 1$-state NFA is admissible (i.e., T(M) = 1), for a latency constraint $L > 0$:

$$\begin{array}{ll} \text{minimize} & cs^1_{0,0}.cost \\ \text{subject to} & cs^1_{0,0}.lat \leq L \\ \text{where} & cs^1_{0,0} \in CS^1_{0,0} \end{array}$$

Typically all detected events are collected at a site/database — this determines the desired location of top level OR. The PECOP problem can trivially be shown to be NP-Hard by a reduction from the Optimal CET-Graph Partitioning problem [76].

Problem Variations. Our problem supports different latency constraints per output event. Simply put, all solutions, computed at the operator that produces this output event, that violate the latency constraint of the operator can trivially be pruned. A second variation includes cases when we only care about the communication cost at specific links of our network (i.e., do not care about high bandwidth links). This can also be trivially handled by setting the hop count across these links to 0 (this zeros the transmission cost across those links in our formulas). Monetary costs (i.e., for pay-as-you-go network pricing scenarios where the cost per link scales proportionally to size of the communicated data [30] within it) can be supported by using the monetary cost in the formulas per communication link, instead of the currently used communication cost.

5. Plan generation algorithms

In this section we propose a dynamic programming algorithm, which is optimal when the number of upstream operators is 1 for each event (i.e., a PE/CE event is not input to more than one operator). We explain why the DP algorithm (while applicable) is no longer optimal when there is event sharing, in which case we propose a Brute-Force, Exhaustive Search approach to reach optimal solutions. We also propose heuristic variants, with different optimizations that can be enabled in them. All algorithms share a preprocessing time cost for network, EDG and basic statistics (Section 4.1) related data structures maintenance; which is dominated by $O\left(|S|^3\right)$, i.e., the complexity of the all-pairs-shortest-path (*APSP*) algorithm, used to compute the inter-site communication latency and number of hops values.

5.1. The dynamic programming algorithm

Algorithm 1 presents our dynamic programming (DP) algorithm. DP first sorts the operators in the EDG using a topological sort. The *sortedList* in Line 3 includes the result of the topological sorting. Then, it computes the Pareto optimal plans, calling the *BuildPlans* procedure, for each operator *op* (Line 5) using this sort order, which ensures that an operator is processed after all of its input operators. The *BuildPlans* procedure (Lines 6–15) takes as input an operator *op* to process, along with its index *opIndex* in the *sortedList*, and considers all potential placements (Line 7) and all possible NFA chain configurations (Line 8) for this operator. For each such combination of operator location S_j and NFA chain configuration μ, the procedure then iterates through all possible combinations of solutions computed at the input operators of *op* (Lines 10–11) and computes their cost and latency (Line 12), using the ideas presented in Section 4.1 and based on the formulas of Appendix. Each computed candidate solution p is checked to see if it satisfies the input constraints (i.e., latency constraints) and for Pareto optimality (Line 13) within the corresponding $CS^{\mu}_{opIndex,j}$ set. If so, p is inserted into the set (Line 15), removing candidate solutions that were dominated by p (Line 14).

A Note on the Principle of Optimality. Regarding the optimality of the problem, we need to explain why (1) it suffices to maintain a set of Pareto optimal solutions per potential placement (site) for each operator, and (2) why it does not suffice to keep a single set of Pareto optimal solutions per operator (and we need one set per site). Regarding (1), consider two candidate solutions $cs1$ and $cs2$, corresponding to a particular placement of operator op_j, such that $cs1$ dominates $cs2$. Let us consider a candidate solution cs' computed at op_i, where op_i receives as input the output event

Algorithm 1: Dynamic Programming Algorithm

```
1  Initialize all $CS^{\mu}_{i,j}$ sets to empty
2  Procedure CreatePlans(EDG)
3      List sortedList = topologicalSort(EDG)
4      foreach $op \in sortedList$ do
5          BuildPlans ($op$, $op$.index)
6  Procedure BuildPlans($op$, $opIndex$)
7      foreach $S_j \in S$ do
8          foreach $nfa.eventSet\ \mu \in powerSet(op.inputEvents)$ do
9              Plan p = new Plan(opIndex, j, $\mu$)
10             inputPoList = All combinations of solutions from $op$'s
               input operators
11             foreach $subPlanSet \in inputPoList$ do
12                 p.computeCostAndLatency(subPlanSet)
13                 if $p.satisfiesInputConstraints() \wedge$
                   $p.isParetoOptimalIn(CS^{\mu}_{opIndex,j})$ then
14                     Remove from $CS^{\mu}_{opIndex,j}$ solutions dominated by
                       p
15                     $CS^{\mu}_{opIndex,j}$.add(p)
```

of op_j. It is trivial to see that cs' will always produce a more preferable solution (in terms of achieved cost and latency) when considering $cs1$, than when considering $cs2$, in combination with *any* other candidate solutions from other input operators to op_i.

Regarding (2), the key is that if $cs1$ and $cs2$ corresponded to different placements, then keeping just $cs1$ does not suffice, even if $cs1$ dominated $cs2$. To see this, consider that the communication cost for the output of op_j to reach op_i depends on the placement of these operators. Candidate solutions that correspond to different placements of op_j require different communication costs and different latencies. For example, $cs2$ may initially seem less preferable than $cs1$, but if op_i is placed closer (or at) the location of $cs2$, then $cs2$ may yield a lower cost/latency for op_i than $cs1$.

Running Time and Space Complexities. Let c denote the maximum number of CEs that are input to an operator and let τ denote the maximum number of Pareto optimal plans that are kept per placement and operator. Our DP algorithm makes $O(|S| \cdot T(c))$ iterations per operator op_i. Each iteration considers all combinations with candidate solutions at input operators of op_i, which are $O((|S| \cdot \tau)^c)$. This yields a total of $O(|Op| \cdot T(c) \cdot |S|^{c+1} \cdot \tau^c)$ for all operators, placements and NFA chain combinations. The space complexity in this case will be $O(|Op| \cdot |S| \cdot \tau)$.

5.2. Event sharing case: Exhaustive search

The DP algorithm, is no longer optimal if the EDG contains primitive or complex events that are shared among (i.e., are input to two or more) different operators of the EDG. Consider the CE e_3, where e_3 is shared by the operators SEQ(e_3, e_4) and AND(e_1, e_2, e_3) in Fig. 1. Each Pareto optimal solution that is maintained for each operator corresponds to the total communication cost and maximum latency for the *entire subtree* of this operator (thus, it is a cumulative cost). The non optimality upon an event sharing arises because of two reasons: (i) if e_3 is a CE, then the communication cost of all Pareto optimal solutions computed at e_3 is added to the solutions of all of its downstream operators (and is, thus, added more than one time), and (ii) the algorithm cannot correctly compute the communication cost for transmitting the output events of e_3 to its downstream operators, since if two or more downstream operators of e_3 are placed at the same location, then the cost of e_3 in this case should be counted just once. Therefore, here we describe an Exhaustive Search (ES) algorithm which is optimal even when event sharing exists. In a nutshell, the ES algorithm described in Algorithm 2 considers all possible placements for each operator and (for each operator placement) all possible push–pull strategies for the operator. For each such combination, ES can compute the correct cost of each plan and return the optimal one that abides by input (network latency) constraints. Contrary to the DP Algorithm 1 which maintains a set of Pareto optimal solutions per potential placement (site) for each upstream operator of an operator op, in ES there are no such precomputed plans. In other words, the plans for the upstream operators of op, are not restricted to those that are Pareto optimal up to that point of the topologically sorted EDG. This is essentially what allows ES to take into consideration all possible cases, including the sharing of events.

The Exhaustive Search (*ES*) algorithm starts by creating a topologically sorted list of all operators of the EDG (Line 2) and then proceeds by iterating through all possible operator placement combinations (Line 3). Then, it iterates for all possible NFA (push–pull) combinations (Line 4) among all operators. This is because each such combination may introduce different cost based on the shared event's place in the NFA (i.e., push–pull order). Based on the already found snapshot, where the combination of operator placement and NFA configuration (push–pull strategy) has been decided, a new, overall plan is initiated (Line 5) and the actual cost of each operator can be found by iterating over the topologically sorted list of operators, additionally taking into account all sharing dependencies (Lines 6–11). Intuitively, when two operators placed at the same site share an event and one operator has set the shared event in the first state of the NFA (in push mode), then in the second operator we can be sure that this event will be timely delivered if detected in a site without the need of a pull request. Thus, the cost for event delivery will have already been computed by the first operator. In any case, the cost for a given set of plans that share an event can be computed by the formulas presented in Appendix. Upon the iteration reaches the top level OR operator, if the computed network latency satisfies constraints and the cost of the currently computed plan is lower than the previously optimal one stored in $CS^1_{0,0}$ (Line 10), the algorithm keeps the current plan as the new optimal (Line 11) and accordingly resets the rest of $CS^{\mu}_{i,j}$ with the current placement and NFA configuration for each operator (Line 13). Notice that for each operator we just keep one candidate solution, which is the one that gives the minimum cost at the top level OR. Therefore, $CS^{\mu}_{i,j} \equiv cs^{\mu}_{i,j}$ (Line 1).

Running Time and Space Complexities. Again, let $|Op|$ denote the number of operators, $|S|$ the number of sites and c denote the maximum number of input events among the operators. The algorithm considers $O(|S|)$ placements for each operator, and for each placement it considers $O(T(c))$ NFA configurations. This leads to an $O((|S| \cdot T(c))^{|Op|})$ number of combinations, each requiring $O(|Op|)$ time to compute its cost, for a total of $O(|Op| \cdot (|S| \cdot T(c))^{|Op|})$ running time. ES does not need to keep sets of Pareto optimal solutions in memory, and thus (although being much slower) requires less memory than DP.

5.3. PaNORAMA: Push–pull in-network plan placement algorithm

Candidate Selection Process Overview. We now present an algorithm with several greedy and heuristic variants. The basic intuition is to limit the number of sites that we consider for the placement of each CEP operator. We term this procedure as *Candidate Selection*. We compute two types of locations per PE and per operator using the statistics of Section 4.1: the Candidate Centers (CCs) and the Candidate Locations (CLs). For each PE, we define its *Candidate Centers* (CCs) as the union of two sets. The first set contains those sites that minimize the *Hoped Frequency* of the event. The second set contains those sites that minimize

Algorithm 2: Exhaustive Search Algorithm

```
1  Initialize all $cs^{\mu}_{i,j} \equiv CS^{\mu}_{i,j}$ sets to empty
2  List topSortList = topologicalSort(EDG)
3  foreach placement ∈ setOfPossiblePlacementCombinations do
4      foreach nfaCombination ∈ setOfPossibleNFACombinations do
5          Plan p = new Plan()
6          foreach op ∈ topSortList do
7              p.add(opIndex, $S_j$ =placement.siteOf(opIndex), $\mu$ =nfaCombination.nfaFor(opIndex))
8              copse = findCoplacedOperatorsSharingEvents(opIndex, topSortList, placement)
9              p.computeCostAndLatency(p.inputCostAndLantecyFor(opIndex), copse)
10             if opIndex = 0 ∧ p.satisfiesInputConstraints() ∧ p.isOptimalIn($CS^{1}_{0,0}$) then
11                 newOptimalPlan p
12         if newOptimalPlan then
13             reset all $CS^{\mu}_{i,j}$ according to p
```

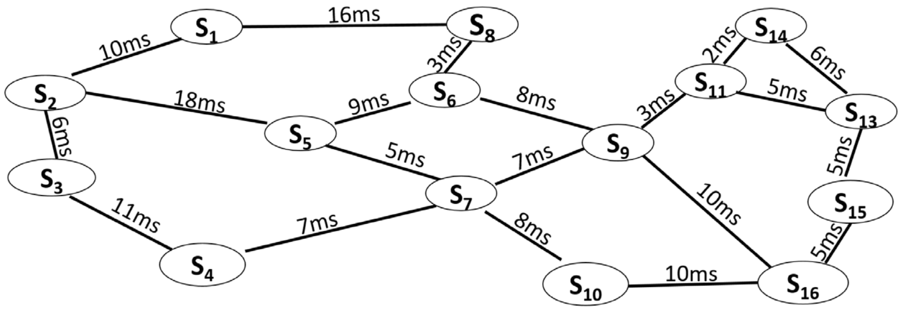


Fig. 5. Example network composed of sites $\{S_1, \ldots, S_{16}\}$. Edges correspond to communication links among them, while network latency values (in milliseconds) are noted on the links.

the maximum transmission latency from the locations where the PE is detected. The basic idea for these two sets is quite intuitive. The first set finds the sites where we can collect all events of a PE with minimum cost. The second set finds the corresponding sites with the minimum collection latency. The locations in these two sets constitute (along with their neighboring sites) a good starting location of where an operator having as input this PE should be placed. For PEs, the notion of CCs and CLs coincides.

For each operator op_i that has inputs just PEs, we set as its CCs the union of the CLs of the input events of op_i. We then create a queue of candidate locations (*candidateQueue*) that initially contains these CCs. For each candidate location in *candidateQueue*, the algorithm will compute the cost of candidate solutions for op_i, starting from a naive plan (i.e., all input events in push mode) for each solution. Each candidate solution that is Pareto optimal is examined further, expanding our search in two ways. First, the NFA chain of the Pareto optimal candidate solution is expanded to look for solutions that potentially have lower communication cost (yet still satisfying the latency constraint). Second, since it seems as an intuitive idea that op_i should be placed somewhere "in-between" its CCs, we consider a site to be in the *vicinity* of the CCs, if the maximum latency from the site to the CCs does not surpass the maximum pairwise latency of the CCs. Given this, we expand the area of search with the potential insertion of such neighboring sites of the candidate location to the *candidateQueue*. The process ends when NFA expansions cannot provide any more Pareto optimal solutions (that satisfy the latency constraint) for op_i and when all candidate locations in the *candidateQueue* list have been processed. At that point, the locations corresponding to the remaining Pareto optimal solutions constitute the Candidate Locations for op_i. The algorithm for operators that also have as inputs CEs is the same, since we just mentioned how we compute the CL list of a complex event.

Example 3. Before formally presenting our algorithms, we outline the function of the Greedy variant of PaNORAMA's Plan Generation process for the operator $\mathtt{SEQ}(\mathtt{e}_3, \mathtt{e}_4)$ from the EDG of Fig. 1 on the network example illustrated in Fig. 5. We assume that the Candidate Selection process assigned S_9 site as Candidate Center for event e_3 and Candidate Center S_4 for event e_4. Then, the algorithm will create the naive 1-state NFA plan with both events (e_3, e_4) on push mode. This plan will be placed to the first site (S_9) of the *candidateQueue* and will be inserted in the operator's single Pareto optimal list. Then, the algorithm will iterate for other NFA configurations, such as e_3 in the first state and e_4 in the second state. Let us further assume that the latter NFA plan provides less communication cost but more detection latency than the already inserted naive plan. Thus, the Pareto optimality criterion upholds and the plan is inserted in the Pareto optimal list. Since a plan entered the Pareto optimal list for S_9, all neighboring sites are examined for admission in the operator's *candidateQueue*. Please note that the maximum minimum latency among the CCs (S_4 and S_9) is 14 ms. Among all neighbors of S_9, only S_7 enters the *candidateQueue*, because its maximum minimum latency from the CCs is 7 ms < 14 ms. All other neighbors of S_9 are not considered to be in the vicinity of S_4 and S_9. For example, site S_{11} is not admitted since its latency distance (17 ms) for one of the CCs (namely S_4) is greater than our bound. After the algorithm has exhausted all NFA plan configurations, or has reached a point that new plans provide worst cost gain than the last ones, for site S_9, the algorithm moves to the next candidate in *candidateQueue* which is S_4 and the process is continued until no other site is left in *candidateQueue*. In this example, at most 4 sites (the only site in the vicinity of S_4 and S_9 that may be added if S_7 helps generate Pareto optimal solutions is S_5) will be considered. □

Greedy and Heuristic Variations. The pseudocode of our algorithms is presented in Algorithm 3. The *Greedy* algorithm begins by creating a topological sorted list from the operators of the

Algorithm 3: PaNORAMA: Greedy, Heuristic and Plus(+) Variations

```
1  Algorithm PaNORAMA(EDG, isHeuristic, isPlus)
2      List<Op> topSortList = topologicalSort(EDG)
3      if isHeuristic then
4          planLists = new HashMap<op, HashMap<site, List>>
5      else
6          planLists = new HashMap<op, List>
7      foreach op ∈ topSortList do
8          if !isHeuristic then
9              list = new List()
10         candidateQueue = findCandidateCenters(op)
11         while ((site = candidateQueue.pop()) != null) do
12             Plan p = new Plan(op, site, 1)  // 1-step Naive Plan -
               Inputs in push mode
13             if isHeuristic then
14                 list = new List()
15             boolean pWasAdded = false
16             // start with the combination of the 1st PO lists from
               each input
17             whichPOListPerInput[|op.inputEvents|] = {1, . . . , 1}
18             do
19                 if op.inputEvents contain CEs then
20                     Set subPlanSet to the combination of the most
                       efficient candidate solutions from each
                       whichPOListPerInput
21                     p.computeCostAndLatency(subPlanSet)
22                 do
23                     p.setToNaivePlan()
24                     Plan pToExpand = p
25                     boolean naiveWasExpanded = false
26                     do
27                         if p.satisfiesInputConstraints() ∧
                           p.isParetoOptimalIn(list) then
28                             list.add(p)
29                             pWasAdded = true;
30                             naiveWasExpanded = true
31                             pToExpand = p
32                         p = pToExpand.getNextNFAConfiguration()
33                     while p != null ∧ naiveWasExpanded = true
34                     p = 1-step plan computed using the next
                       combination of Pareto optimal solutions in
                       op's input whichPOListPerInput
35                 while isPlus ∧ p != null
36                 Set whichPOListPerInput to the next valid
                   combination of input Pareto optimal lists to
                   process
37             while isHeuristic ∧ whichPOListPerInput.isValid()
38             if pWasAdded then
39                 for candidate ∈ site.getNeighbours() do
40                     if isInTheVicinity(candidate, op.candidateCenters)
                       then
41                         candidateQueue.add(candidate)
42                 if !isHeuristic then
43                     planLists.add(op, list)
44         if isHeuristic then
45             planLists.get(op).add(site, list)
46     return planLists
```

EDG (*topSortList* in Line 2). A single plan list is created for each operator *op* (*planLists* in Line 6), since for the Greedy algorithm *isHeuristic* = *false* (Line 3). Then, the algorithm iterates for every operator in the topologically sorted list (Line 7–45). A set of candidate centers are found for the operator *op* (Line 10) and placed in a queue (*candidateQueue*). The pseudocode does not have the details of this operation, which was however described at the beginning of Section 5.3. The algorithm then proceeds by iterating through all the sites in the *candidateQueue* (Lines 11–43). A new naive plan *p* (i.e., a 1-state plan with all inputs in push mode) is then created and placed on the candidate site in question (Line 12). The cost and latency of this plan are computed as described in Section 4.1, using for each input operator the most efficient, in terms of communication cost, Pareto optimal solution computed at that input operator (variable *whichPOListPerInput* initialized in Line 17 and used in Lines 19–21). Thus, only one combination of solutions from input operators is considered in Greedy.

For Greedy, the do-while loops in Lines 18–37 and 22–35 are only accessed once per candidate site, since the *isHeuristic* and *isPlus* flags are set to false. At their first execution (within each do-while loop in Lines 22–35), Lines 27–31 consider if the current 1-step plan satisfies the input constraints and provides a new Pareto optimal solution. If not, Line 33 terminates the do-while loop due to the *naiveWasExpanded* variable being false and the candidate solution is discarded. If yes, we continuously try in Lines 26–33 to expand this plan by considering NFA chains that are longer by 1 step (as we did in Example 3). For each plan that we consider to expand, there are multiple possible NFA expansions (and, thus, the loop). If an NFA expansion provides a Pareto optimal solution, then we start trying to further expand the plan with this new NFA (*pToExpand* = *p* in Line 31 and *pToExpand*.*getNextNFAConfiguration*() in Line 32). If a naive plan helped generate a Pareto optimal solution, then all of the candidate site's neighbors are inserted in the *candidateQueue* (Line 41), given that they reside in the vicinity of the Candidate Centers (*isInTheVicinity* in Line 40).

For Algorithm 3, based on the value of its two Boolean parameters (*isHeuristic*, *isPlus*), we first name and then explain the versions of our algorithmic variants (besides Greedy which was outlined above):

- Greedy: PaNORAMA(EDG, F, F)
- Greedy+: PaNORAMA(EDG, F, T)
- Heuristic: PaNORAMA(EDG, T, F)
- Heuristic+: PaNORAMA(EDG, T, T)

The number of combinations of candidate solutions considered for the cost and latency of an operator is different among our algorithmic variants:

- The Greedy+ variation keeps one Pareto optimal set per operator, but checks all combinations of solutions from input operators.
- The Heuristic variation maintains one Pareto optimal set per placement of an operator. When computing candidate solutions for *op*, Heuristic considers all combinations of placements of its input operators (*whichPOListPerInput*), but processes just one candidate solution (as the Greedy variation does) per such combination.
- The Heuristic+ variation keeps one Pareto optimal set per operator placement. To compute candidate solutions for *op*, Heuristic+ considers all combinations of placements of its input operators (*whichPOListPerInput*) and of candidate solutions within each Pareto optimal set.

The previously explained differences between Greedy, Greedy+, Heuristic and Heuristic+ correspond here to whether the do-while loops in Lines 18–36 and 22–34 will be executed just once, or multiple times. For Heuristic and Heuristic+, Line 36 essentially moves the processing to the next combination of input Pareto optimal solutions (i.e., combinations of site locations) computed at input operators. Moreover, Lines 44–45 store a Pareto optimal list per operator *op* and site location (thus, keeping multiple lists per operator). Similarly, for the Plus(+)

variations, Line 34 considers the next combination of candidate solutions from the Pareto optimal solutions computed at the input operators of *op*.

Running Time Complexity. Let c, τ as defined in Section 5.1. In the worst case, if the candidate centers are on the opposite edges of the network, the *Greedy* algorithm may have to visit most sites in the network, i.e., $O(|S| \cdot |Op| \cdot T(c))$ complexity. The corresponding complexity for Heuristic is $O\left(|S|^{c+1} \cdot |Op| \cdot T(c)\right)$. The complexity bound increases to $O(\tau^c \cdot |S| \cdot |Op| \cdot T(c))$ for Greedy+ and $O((|S| \cdot \tau)^c \cdot |S| \cdot |Op| \cdot T(c))$ for Heuristic+.

6. System implementation

In this section we first discuss the necessary modifications to a (distributed or centralized) CEP system so that it can support the push–pull approach. A distributed CEP system can run on more than one machines inside each site (e.g. combined with Apache Storm [27,28,75]) and may execute each operator assigned to a site in parallel across different cluster machines. We then describe the architecture that needs to encompass each CEP system (installed at each site), its operation, and the required rewritings that the optimizer performs. Our architecture has been designed so that it is not dependent on the underlying CEP engine, since it only requires logic and code to be added to places where an event is produced inside a CEP system. Since all CEP systems are built to detect events, such a place exists in all of them. The code to be added inside a CEP engine includes: (a) Registering this list of detected events that are initially at pull mode when receiving the plan from the optimizer, and (b) Transmitting detected events that may be pulled from remote sites outside the CEP engine for caching.

The algorithms of Section 5 have been implemented within the FERARI system [34,35], as part of a query optimizer running on top of the IBM ProtonOnStorm [28] distributed CEP system. ProtonOnStorm employs the concepts discussed in [2,12] as it organizes the processing in an Event Processing Network (EPN) composed of Event Processing Agents (EPAs) each dividing its operation in filtering, matching and derivation steps (please refer to [28] for further details). With respect to our previous discussion, when a CEP engine follows these concepts, the EDG corresponds to the constructed EPN for the posed CEP query, while CEP operators correspond to EPAs. Other CEP engines follow different conceptualizations but can still be incorporated in FERARI. For instance, for validation purposes, we have recently successfully incorporated EsperOnStorm [27] in FERARI and applied the same modifications ((a), (b) mentioned above) on top of it. EsperOnStorm employs a different conceptualization composed of a mixture of trees, automata and logic programming-related concepts.

Each of the modules described below and shown in Fig. 6, in [34,35] are encapsulated in bolts in Apache Storm topologies run at individual sites. Thus, any CEP engine encapsulated in a bolt can be directly supported by our architecture and this is not even limited to Apache Storm. The exact same design directly applies to popular streaming platforms including Apache Heron [77], Apache Flink [78], Apache Ignite [79] among others. The optimizer runs a plan generation algorithm and decides which operators are assigned at each site. It then creates the corresponding CEP logic for each site, after performing rewritings mentioned later on in this section. The optimizer transmits the CEP logic to each site, which is used as input in its CEP system.

Modifications to a CEP System. Fig. 6 depicts the architecture for each site. In distributed (clustered) architectures, the CEP engine used could be a distributed one (i.e., EsperOnStorm [27], ProtonOnStorm [28] etc.). The CEP engine (at each site) executes the CEP logic transmitted to it from the optimizer. Existing CEP engines support communication with remote sites. We thus focus on the required modifications for supporting the push–pull approach.

Pull messages need to occur only upon a state transition in a NFA automaton. Upon each state transition, an event is emitted inside the CEP engine. Upon seeing this event, a pull request is also emitted from inside the CEP engine towards the Communication Module (this particular step involves adding code within the CEP engine). This pull request is just a message/event that the CEP system needs to emit. The pull request contains the name of the pulled events (these are inputs to the next NFA state of the specific operator and are simple to find from the CEP logic/file that the optimizer transmitted) and specifies the time window for which these events are pulled. These time windows depend on the operator (`AND`, `SEQ`) and are explained promptly.

The remaining logic for the push–pull has been pushed to the other components of our architecture. The Communication Module is responsible for sending/receiving events or pull requests from remote sites. The Communication module stores the pull requests (the pulled intervals per event type) at the Time Buffer Module, to ensure that two pull requests for the same event will not overlap (thus, ensuring that the same event will not be pulled twice). At the site pulling an event, multiple pull requests for an event that is actively being pulled by a previous request are aggregated and a single pull request for them (with the maximum desired pull duration) is transmitted shortly before the pull window expires. Received pulled data are forwarded to the CEP engine. The Time Buffer Module also stores all derived complex events that are input to other sites, but are by default cached (unless they are pulled). This is essentially a buffer of events and pull requests.

The optimizer, besides deciding which operators are placed at which site, also performs important rewritings, explained below. The events generated in a CEP system can have arbitrary attributes, and these (along with how to compute their values) are included in the CEP logic fed into the CEP system. The optimizer augments each event with an additional timestamp. This *realOccurrence* timestamp for primitive events is set to the time that they were produced. For complex events, its value is the maximum among the realOccurence timestamps of all input events to the operator that generated this event. This timestamp represents the time that the complex event would have been detected, if there was zero processing and communication latency. All checks regarding the time of each event are performed on this timestamp.

Operator-Specific Rewriting - `AND` operator. For an NFA state, let t_{min}/t_{max} denote the minimum/maximum realOccurence timestamp of all detected events in the previous NFA states of the operator. Then, the pull request when the NFA state is activated includes all detected events with OccurenceTime within the window $t_{max} - W \leq t_{pull} \leq t_{min} + W$.

Fig. 7 presents an example of the execution of the complex event $\mathtt{AND}(\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3)$ within $W = 2.5$ s and a 3-state plan ($e_3 \rightarrow e_1 \rightarrow e_2$). All depicted timestamps are realOccurence timestamps. The site hosting the aforementioned plan/operator waits for events of type e_3. Upon detection of an event of type e_3, a pull request is issued upon the transition to the 2nd state of the NFA that includes events of type e_1. The pull request searches for events of type e_1 from event sources in our network with realOccurence time: $t_{e_3} - W \leq t_{e_1} \leq t_{e_3} + W$. Upon detection of an event of type e_1 a pull request is issued for events of type e_2 with realOccurence time: $t_{e_1} - W \leq t_{e_2} \leq t_{e_3} + W$. Upon the arrival of an e_2 event within the requested time range, a complex event is generated.

Operator-Specific Rewriting - `SEQ` operator. The sequence operator is similar to the AND operator but additionally requires

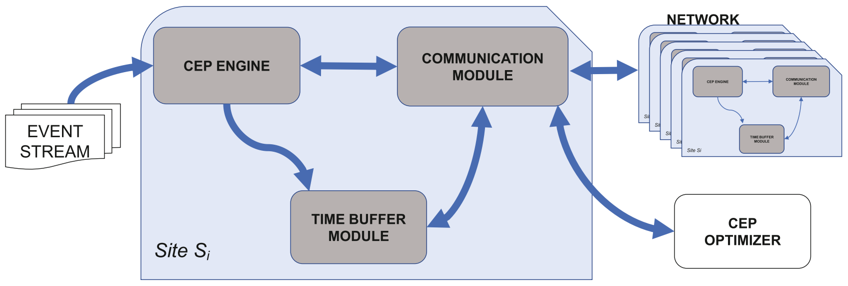


Fig. 6. Site architecture. A CEP optimizer at a central site collects statistics from and prescribes the execution plan for the running CEP queries using our algorithms. Each site is composed of a CEP Engine which is the heart of the intra-site architecture. It executes the CEP logic transmitted to it from the optimizer. A Time Buffer Module is a cache of events and pull requests. A Communication Module is responsible for implementing the required communication according to the prescribed push–pull strategy.

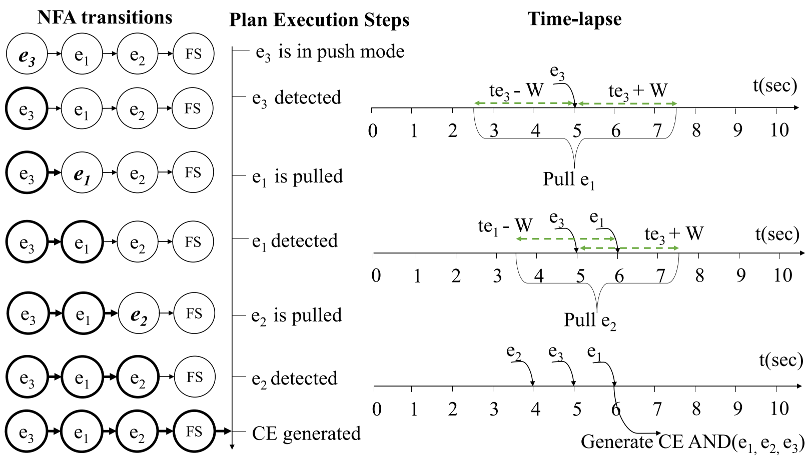


Fig. 7. AND(e_1, e_2, e_3) with $W = 2.5$ s execution example.

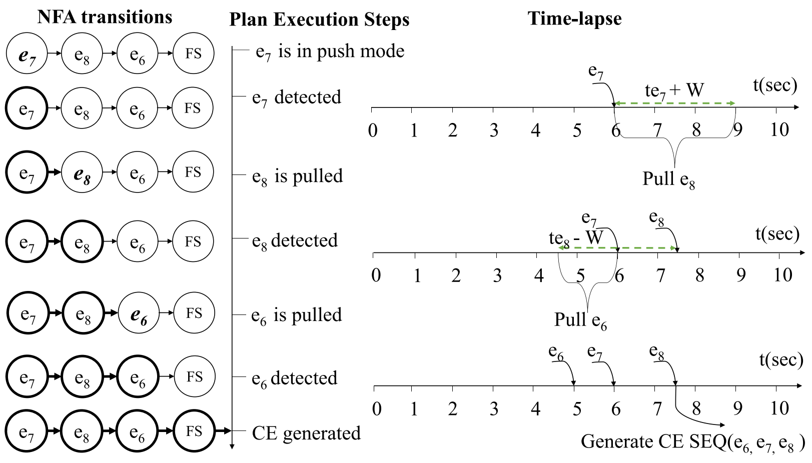


Fig. 8. SEQ(e_6, e_7, e_8) with $W = 3$ s execution example.

that the time ordering of the events will also uphold $t_{1^{st}event} \leq \cdots \leq t_{i^{th}event}$. As such the SEQ operator is transformed into a series of AND operators (1 per state of the NFA) and the transition from one state to the next marks the pull request of the events included in the next state. The pull request must simultaneously conform with the time ordering of the events and with the window constraints.

Fig. 8 presents an example for the execution of the complex event $SEQ(\ e_6,\ e_7,\ e_8)$ within $W = 3$ sec and a 3-state plan ($e_7 \rightarrow e_8 \rightarrow e_6$). In this example, upon detection of an event of type e_7 a pull request is issued upon the transition to the 2nd state of the NFA that includes events of type e_8. The pull request involves events of type e_8 with realOccurrence time: $t_{e_7}+W \leq t_{e_8}$ that may occur in the future. Upon detection of an event of type e_8 a pull request is issued for events of type e_6 with realOccurrence time: $t_{e_8} - W \leq t_{e_6} \leq t_{e_7}$.

Handling Latency in Event Detection. Up to this point we have described which events are pulled in NFA state transitions. The second parameter is how long will the operator wait for these pulled events to arrive to determine if a CE has occurred. In order to avoid missing complex events the optimizer enlarges the plan's window by adding the plan's latency (up to the operator) to the operator's window size W. By doing so the system waits a sufficient amount of time for delayed pulled input events. Please note though that this window enlargement does not increase the amount of pulled events — it just ensures that these will arrive before the operator determines if a complex event (or a state transition) has occurred.

Filter Conditions and Local CEs. In our event data model (Section 3.1), we provide a definition that matters for our algorithmic analysis. Apart from event occurrence and timestamp, real CEP engines allow for defining attributes of events (PEs or CEs). Contrary to CEP operators, attributes do not get evaluated, but are calculated or get sampled from the environment. Hence, attributes do not get explicitly involved in our algorithms, but handling selections on attributes, i.e., based on filter conditions, should be accounted for in practice. As an example, in a mobile fraud detection scenario [34,35], an outgoing, long call to a premium location during night hours may produce a mobile fraud event. The temporal window W interprets to "night hours". The CEP operator is a thresholded aggregation operator summing up evolving call durations and comparing the aggregated duration with the specified threshold for characterizing a call as "long". The "call duration" is the attribute that is calculated here, but does not trigger an event per se. Should this attribute surpass the posed threshold, the call is characterized as long and this is the produced CE. Finally, there exist two selection criteria for the input events. The first selection (filter) comes from the fact that the attribute, say "call type", should equal "outgoing". The second selection says that the attribute "call destination" should receive a value from the set of locations that are considered as premium.

Each CEP operator may incorporate selections on attributes of its input events of the form $e_i.attr_j \lesseqgtr value$. The optimizer strips these conditions from the operator itself and pushes them (as simple filtering operators) to the sites that generate e_i. Filters to input CEs are pushed to one site, while filters to PEs are pushed to all sites that observe them.

7. Experimental evaluation

Data Sets and Network. We use two real data sets from the stock exchange and the telecommunications fields:

• Stock Trade Traces: We use a real data set (also used in [11, 13]) of stock trade traces [80] with 120000 event occurrences along with their timestamps. The data set includes 10 different primitive event types which correspond to actual companies' stock names. The data set was analyzed and event frequencies were calculated for each primitive event. For this data set, we placed the PEs in our tested network configurations using a methodology described shortly. This allowed us to perform a detailed sensitivity analysis (Figs. 9, 10) for all tested algorithms.

• Telecommunication Data Set: The real data set from a large Telecom provider [34,35] includes 160 Million, properly anonymized, mobile phone call records. The calls were monitored by a network composed of ∼20000 antennas and the goal is to detect mobile fraud incidents. The mobile fraud related primitive events, also properly masked for security reasons, that are being monitored involve: (PE1) Calls to premium locations, (PE2) Calls with duration exceeding the threshold XDur, (PE3) Calls whose monetary cost exceeds the threshold XCost, (PE4) Calls with duration higher than (*StDevDur* stands for the standard deviation of the call duration) $Y \cdot StDevDur$ times the average duration for the user, and (PE5) Calls with cost higher (*StDevCost* stands for the standard deviation of the call cost) than $Y \cdot StDevCost$ times the average monetary cost for the user. For company privacy issues, we were not given the exact values used by the Telecom provider of the aforementioned thresholds. We thus used (after discussions with them) reasonable values of XDur = 60 min, XCost = 100 monetary units (all prepaid cards cause zero call cost), and Y in $Y \cdot StDevDur$, $Y \cdot StDevCost$ is 2. In this data set, each PE is associated with an antenna, which is more realistic, but allows us for a more restrictive sensitivity analysis (Fig. 11).

Table 2
EDG, network and optimization parameters.

Name	Measurement unit	Range (**Default**)
Network related parameters		
# of sites	sites	10–20000 (**2000**)
Primitive event distribution diameter	hops	10–100 (**50**)
Primitive event distribution Skew	number	0.01–10 (**0.01**)
Event related parameters		
# of complex events	events	1–9 (**3**)
# of shared primitive events	events	0–3 (**0**)
Operator time window (W)	sec	0.1–10 (**2**)
Latency limit factor	number	1–4 (**3**)

• Real Network Configuration: a network composed of ∼20000 antennas, with approximate coordinates for each antenna [34,35]. The network contains big groups of antennas around big cities and smaller groups around smaller cities or villages. We were able to vary the number of the sites by randomly sampling on the actual antennas.

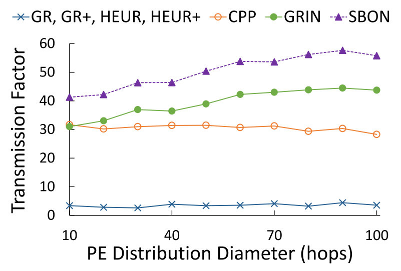
The Real Network Configuration, paired with the Real Dataset and query specifications constitutes a full scale real application case study for our proposed approaches. To perform stress tests on our techniques where we vary every possible network or event related parameter (presented in Table 2 and analyzed below), we also used the Stock Trade Traces over the real network assigning event tuples to sites as analyzed below.

(a) Network Related Parameters: Given the overall frequency of Primitive Events for the Stock Trade Trace dataset, an initial site is picked at random as the Event Distribution Center (EDC) of each PE. Starting from the EDC of a PE, a random walk takes place to determine the sites that detect the particular PE and fuse part of the overall PE frequency to these sites. The average number of source sites for each PE is 10. We also vary the network locality of this event fusion. The tunable parameters for performing this process are (also see Table 2 describing the parameters of our experiments): the PE Event Distribution Skew, which is how much equal is the portion of the overall PE frequency that is assigned to a site; and the PE Distribution Diameter, which is the maximum distance (in hops) among sites in which each PE appears.

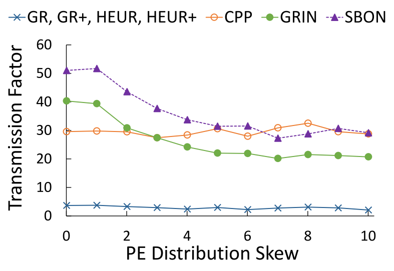
(b) Event Related Parameters: Our query generator simulates queries varying the number of CEs (# of Complex Events in Table 2) and the time windows W used for the respective operators. All parameters are explained in the sensitivity analysis experiments where they are varied. All experiments were executed 100 times under the same random generator parameters.

Finally, the latency limit factor denotes the latency constraint that is imposed in each case, as a multiple of the latency that a centralized collection would incur. As we will see, large values lead to all PaNORAMA algorithms performing similarly (i.e., Fig. 9), while smaller values close to 1 correspond to tight latency constraints and help separate the performance of our techniques.

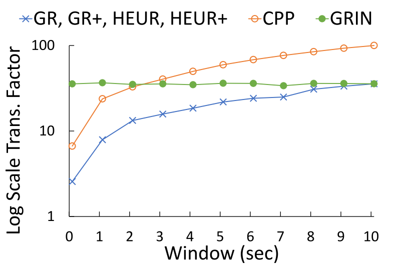
Compared Candidates. We compare the following algorithms:
Baseline Approaches:
(a) Naive: This is a centralized processing (i.e., without in-network placement) algorithm, which does not support push–pull. The communication cost of all other algorithms will be expressed as a fraction of the corresponding cost of this naive algorithm.
(b) Centralized Push–pull (*CPP*): The heuristic algorithm, proposed in [4], applies a push–pull paradigm, but performs central event data collection at a fixed site. In order to avoid placement



(a) Increasing the area in the network where a PE is detected.

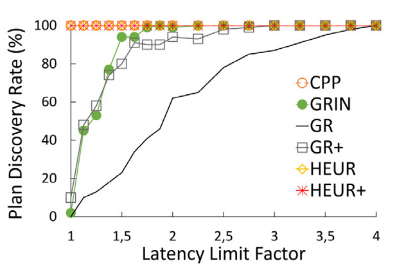


(b) Increasing PEs' frequency skew in network fusion.

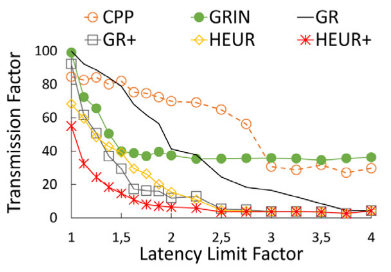


(c) Transmission Factor when increasing the operator time window.

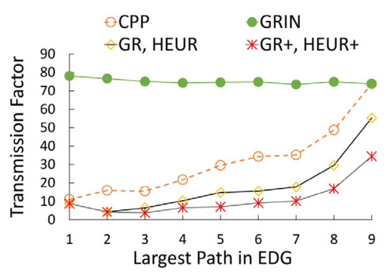
Fig. 9. Cost and quality sensitivity analysis for various network and data related parameters on Stock Data.



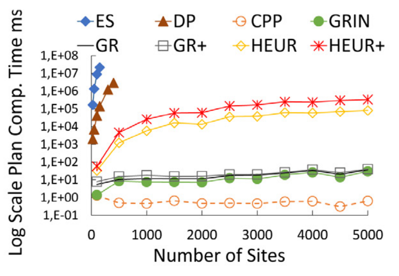
(a) Impact of the Latency Limit Factor on plan discovery rate.



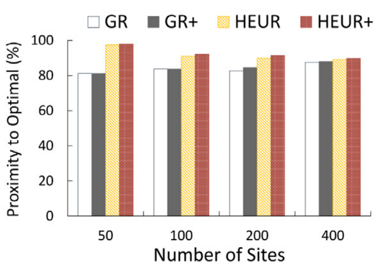
(b) Transmission Factor when increasing the Latency Limit Factor.



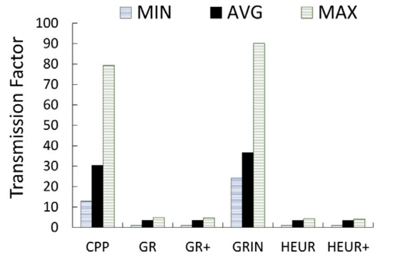
(c) Increasing the largest path in the EDG.



(d) Plan computation time for increasing number sites.

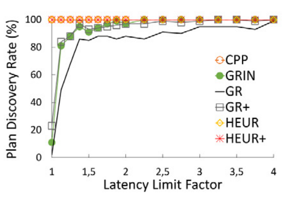


(e) Algorithmic proximity to optimal.

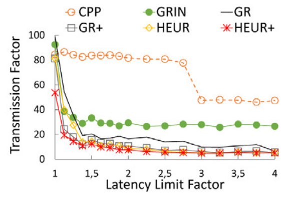


(f) Deviation from average values among different runs.

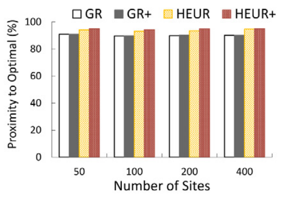
Fig. 10. Cost, Time and quality sensitivity analysis on various network, EDG and optimization related parameters.



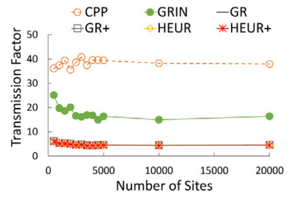
(a) Plan discovery rate for varying latency limit factor.



(b) Transmission factor for varying latency limit factor.



(c) Proximity to *DP* solution.



(d) Increasing number of sites.

Fig. 11. Mobile fraud detection case study.

bias, this site was randomly picked and, to be fair, our algorithms also needed to send the resulting CEs, after the in-network processing, to the same central site,

(c) Greedy In-Network placement without Push–pull (*GRIN*): This is a greedy algorithm that performs in-network operator placement, using our *Greedy* algorithm, but without supporting the push–pull paradigm. Thus, *GRIN* is a variant of our techniques as well.

(d) SBON [37]: We use the highly cited work of SBON [37], to further (together with *GRIN*) exhibit the inefficiency of mere distributed stream processing approaches in CEP settings and the low potential of blended metrics such as network usage (Section 2, Table 1). By design SBON does not admit Pareto or constrained optimization, thus we focus on its communication performance and sensitivity to network related parameters (Table 2, Fig. 9).

Optimal Algorithms: including the introduced Dynamic Programming Algorithm (*DP*), for optimal solutions in the absence of event sharing and the Exhaustive Search Algorithm (*ES*) for optimal solutions if shared events exist on the EDG. These algorithms were examined in smaller scale experiments due to their prohibitive time complexity.

PaNORAMA variations: Greedy, Greedy+, Heuristic and Heuristic+ algorithmic variations described in Section 5.3, hereafter termed as *GR*, *GR+*, *HEUR* and *HEUR+* respectively.

Metrics. The above candidates were compared using these metrics:
(i) Transmission Factor: Defined as the ratio among the communication cost (messages/sec) of an algorithm over the communication cost of naive central data collection (no in-network placement and no push–pull). Lower values are preferable. Under a pay-as-you-go network pricing model, this metric also provides an indication of the pricing cost of our techniques, compared to the naive algorithm.
(ii) Plan Computation Time: Average time a candidate algorithm needs to be executed and output the plan for geo-distributed CEP. Lower values are better.
(iii) Plan Discovery Rate: This metric is defined as the average number of times that an algorithm discovers a solution, given a latency constraint.
(iv) Proximity to Optimal: This is the fraction of the optimal communication cost of *DP* or, under event sharing, *ES* over GR, GR+, HEUR, HEUR+, using the same latency limit factor.

7.1. *PaNORAMA sensitivity analysis*

In Fig. 9 we evaluate all candidate algorithms using the Stock Trade Traces data set. We note that in this figure, the default value of the latency limit factor is 3, in which our PaNORAMA techniques perform similarly. Different values of the latency limit factor will be evaluated shortly. In subgraphs when two or more candidates have negligible (indistinguishable) differences, we plot just the worst of these candidates and merge their names to create a single candidate (i.e., *GR*, *GR+*, *HEUR*, *HEUR+*), for enhanced graph readability. Additionally, since in these scalability experiments we use a high, default network size of 2000 sites, *DP* and *ES* are omitted due to their high time complexity.

Network fusion of events. In Fig. 9(a) we alter the PE distribution diameter. Lower diameter values translate to a smaller distance among the sites where each PE is detected, making in-network placement more effective (close to the sources). Contrary, larger values force in-network placement closer to a central (across the distribution diameter) site, since the events are less localized, which causes *GRIN* to perform worse. *CPP* by design centralizes data at an apriori chosen site and its performance is not affected by this parameter. Our algorithms, which combine different placement and push–pull strategies in their search space, achieve 8–10 times lower Transmission Factor compared to *CPP* and up to 12 times compared to *GRIN*.

Consider the (total) number of times that a PE is observed in the network. If this PE is not observed an equal number of times at each site where it appears, then the skew of this number of appearances is denoted as the PE Distribution Skew. In Fig. 9(b) we depict the performance of all algorithms varying this parameter. For very skewed distributions of each PE, *GRIN* will place the operator near (or on) the site where the larger part of PE frequency is left, making it more likely to make the right in-network choice and causing its Transmission Factor to decrease in Fig. 9(b). *CPP* remains relatively steady as it neglects in-network placement. Still, our techniques offer 7 to 12 times lower Transmission Factor than *GRIN* and 8 to 17 times lower Transmission Factor than *CPP*.

In Figs. 9(a), 9(b), *SBON* follows *GRIN*'s trends but appears to exhibit the worst performance. This validates our claim in Section 2 about *GRIN* being a *best case* representative for DSP approaches. *SBON* fails to admit Pareto optimality, while instead using a blended, network usage metric. This metric squares the latency on communication links [37]. Thus, although *SBON* does not directly admit latency constraints (it is tested without a latency limit), it favors solutions of low latency, but also potentially higher cost. This was consistent in other experiments and we, thus, omit SBON from our remaining evaluation.

Fig. 9(c) demonstrates the impact of our push–pull optimization, when varying the utilized window size of the CEP operators. As the parameter is increased, the probability that a complex event is detected increases as well. This means that with continuously increasing probability the input events that are initially set to pull mode will need to be transmitted, thus minimizing the benefits of the push–pull optimization (both in our algorithms and in *CPP*, which may end up performing similarly to the naive algorithm). This is why our algorithms start approaching *GRIN* with higher values of these parameters. *GRIN* shows steady behavior as it does not utilize the push–pull rationale.

7.2. *Algorithmic differentiation analysis*

In Fig. 10 we evaluate our algorithmic variations along with DP and ES. We designed this set of experiments with network, event and optimization related parameters that help differentiate our proposed variants and illustrate their strengths and weaknesses. For the default values of parameters that are not varied please see Table 2.

Imposing latency constraints. Fig. 10(a) examines the Plan Discovery Rate of the algorithms that do not fall in the optimal category under various Latency Limit Factors. Note that our PaNORAMA algorithms prune the search space (i.e., do not consider placing each operator at each site) and may end up with no execution plan that satisfies the posed latency limit. As shown in the figure, *CPP*, *HEUR* and *HEUR+* can always come up with a distributed execution plan (note though that this is not theoretically guaranteed for our approaches), while the same happens for *GRIN* for Latency Factors above 2. *GR* naturally exhibits the lowest plan discovery rate, which is expected, as it maintains and considers the fewest candidate solutions per operator. To depict the quality (in terms of their communication cost) of the plans discovered in Figs. 10(a), 10(b) depicts the Transmission Factor for the same experiment. Algorithms that do not find a solution in a particular run use the naive solution for that run. As we decrease the value of the Latency Limit Factor, we essentially restrict the length of the NFA chains of each operator, thus limiting the number of its states and the amount of "pulling". The expected behavior is that this will lead our algorithms to a higher Transmission Factor, which justifies the behavior shown in the figure. As the problem becomes more constrained, *HEUR+* remains the algorithm of choice.

Variations' Sensitivity. We have noticed that the quality of the solutions of the PaNORAMA algorithms becomes more apparent as the queries become more complex (i.e., the EDG contains larger paths). In Fig. 10(c) we create an increasingly larger chain of complex events (CEs), creating EDGs with more levels. To achieve this, we created CEs (1 to 9, in the figure) that always have 3 inputs. The bottom-most CE has as input 3 PEs. Any other CE has as input 2 PEs and the CE at the immediately "lower" level of the EDG. This creates a chain of 1 to 9 CEs. In Fig. 10(c) we confirmed our expectation that the discriminating factor in the performance of our variations would be the use of the (+) feature. In such EDGs, where each CE is input to the next, the best plan of the kept lists is rarely the plan that minimizes the overall Transmission Factor. Our variations that have the ability to check all plans from the kept lists exhibit significant gains (up to 52%) from the ones that do not (*GR*, *HEUR*) as the CE chain grows.

Plan Computation times, proximity to optimality and statistical importance. In Fig. 10(d) we present the plan computation times required by the algorithms in the optimal category *DP*, *ES*, versus the other candidates. Due to the poor scalability of the *ES* and the *DP* algorithm we restrict our experiment to 150 and 400 sites, respectively, for them. The plan computation time of *ES* is more than 2 orders of magnitude higher than *DP*, which in turn is from 160–700 times higher compared to the most costly of the non-optimal algorithms. The *GR*, *GR+* variations, which examine a narrowed search space, require just a few milliseconds. The cost of HEUR, HEUR+ increases by increasing the network size as they maintain a Pareto optimal set per placement (site) of an operator. Fig. 10(e) shows the proximity of our algorithmic variations (computed as the ratio of the Transmission Factor of the optimal algorithm to the Transmission Factor of each corresponding variation) to the optimal solutions, in a scaled down experiment of up to 400 sites. This ratio varied between 81% and 97%, which shows that our algorithms, while much faster than *DP* and *ES*, provide solutions with communication cost close to the solution generated by the *DP* algorithm. To display the statistical significance of our results, due to the random nature of the event fusion, we conducted an experiment with 1000 runs with the default configuration values. In Fig. 10(f) we can see the boundaries of the algorithmic results, for the default configuration values, with more than 90% confidence. It can easily be observed that *GR*, *GR+*, *HEUR*, *HEUR+* all exhibit small deviation from their average Transmission Factor, contrary to *CPP* and *GRIN*, thus demonstrating that their performance is more robust to different parameter settings.

7.3. Mobile fraud detection case study

Imposing latency constraints. We now apply our proposed algorithms on the set of telecommunication data, network setup and application rules described at the beginning of this section. In Fig. 11(a) we observe that with the exception of *GR*, which however exhibits a rate $>$ 80% in most cases, our algorithms quickly approach 100% Discovery Rate even for Latency Limit Factors just above its minimum value of 1. Switching to Fig. 11(b), it is evident that *CPP* can lead to plans that are up to 14 times worse than our approaches, with *GRIN* also being up to 6 times worse. From our approaches, *HEUR+* exhibits the best Transmission Factor, with its benefits, compared to our other algorithms, being larger in more latency constrained setups, while *GR+* also seems as a good candidate, given its balance between communication reduction and plan discovery rate. Please recall that in Fig. 10(d) *GR+* possesses negligible plan computation time. *CPP* and *GRIN* may exhibit even lower plan computation time, but as shown in Fig. 11(b) they produce significantly worse solutions.

Quality and scalability Since the rest of the parameters are fixed by the application field, we can only vary the number of participating sites of the telecommunication network. We next perform a significantly smaller experiment with up to 400 sites and check the proximity of our solutions to the ones generated by the *DP* algorithm. In Fig. 11(c) we can see that our proposed algorithms exhibit a 89%–94% proximity to the *DP* one. Finally, in Fig. 11(d) the scalability of our proposed methods is illustrated with increasing number of sites. *CPP* is up to 9 times worse, while the *GRIN* approach is 3–4 times worse than the PaNORAMA approach. Note that the number of PEs is only 5 in this real data set, implying that fewer operators are placed and fewer gains are expected, compared to the Stock Trade Traces data set.

7.4. Rules of thumb for PaNORAMA variations

Up to this point, the efficiency of our proposed algorithms compared to other candidates has been validated in a variety of setups and parameter settings. We conclude our evaluation by providing a list of rules of thumb in accordance with the main findings previously discussed in our experimentation. These rules are intended to guide future adopters with respect to which PaNORAMA variation is more suitable given the characteristics of a studied application scenario.

- **Strict Latency Constraints — High Plan Computation Time Allowed:** The *HEUR+* variation is preferable when the application poses strict network latency constraints, but it allows higher plan computation times. The latter mainly depends on the volatility of the streaming setting itself, i.e., how often statistics change so that higher plan computation times do not provide outdated solutions. In this particular occasion, *HEUR+* can, practically always, provide an execution plan which yields the best transmission factor among the PaNORAMA variations.
- **Strict Latency Constraints — Lower Plan Computation Time Required:** The *HEUR* variation is preferable when the application poses strict network latency constraints, but it is less willing to wait for an execution plan to be computed. In such a case, *HEUR* retains the ability to always provide an execution plan and this plan is computed much faster (around 5 times compared to *HEUR+* in Fig. 10(d)). However, the transmission factor of the computed plan may fall short compared to, not only *HEUR+* but also *GR+*.
- **Mediocre Latency Constraints — Lower Plan Computation Time Required:** Based on the observations drawn throughout our experimental study, the *GR+* PaNORAMA variation is not capable of providing an execution plan when severe latency constraints are posed by the application. In case these constraints are looser, however, *GR+* quickly provides an execution plan that is the second best in terms of transmission factor among the proposed variations.
- **Loose Latency Constraints — Minimum Plan Computation Time Required:** The *GR* PaNORAMA variation is useful in settings where the fast computation of an execution plan is of utmost important, i.e., in highly volatile streaming settings. This is because the main characteristics of *GR* are that it surely provides a plan only when the latency constraints are loose, it possesses the lowest plan computation time and still outperforms other candidates proposed in the literature, such as *GRIN*, *CPP* or *SBON*.

8. Conclusions and future work

In this work we presented a novel approach for detecting complex events in geographically distributed, streaming event applications. Our work employs in-network CEP operator placement along with incorporating the push–pull paradigm, in an effort to reduce the communication cost while also controlling network latency. Our proposed techniques vary from optimal (but very time consuming) algorithms to fast and efficient greedy ones that intuitively handle the reduction of the placement search space. Additionally, at a systems level we elaborate on how many existing CEP systems can be modified to support our algorithms. Our experiments, focusing on real data sets and a real topology, reveal the superiority of our approach compared to prior work.

There exist other optimization metrics that have been used in the literature, which focus on the performance of algorithms destined to operate on a single site. For instance, parallel CEP approaches [12,16–21] aim at optimizing throughput and/or computational latency at a single site. These techniques are orthogonal to ours. Besides, prior work comments that the maximum processing rate that such centralized approaches can reach is network bound [23]. Our work is devoted to loosening this network bound, which will also have a positive effect on throughput and latency within sites.

Our future work concentrates on more challenges encountered in real-world settings where constraints beyond network latency exist. In particular, intra-site capacity constraints related to CPU utilization and memory consumption should also be taken into consideration. For instance, if in-network placement assigns multiple heavy load CEP operators to a site, the memory or CPU capacity of the site must be enough to carry out their evaluation, otherwise the provided solution is not a feasible one. Our ongoing work re-examines the Pareto front of possible solutions or attempts to find solutions that were erroneously pruned because they were dominated by the infeasible one. Moreover, there are memory consumption versus network latency trade-offs due to the fact that the push–pull application requires caching events until they are pushed or get expired. Although in the real scenario used in our evaluation these constraints and trade-offs had only a minor effect, they may arise in resource constrained environments such as sensor network settings.

Besides, network latency is not the only dimension that may affect alignment with SLAs and QoS requirements. Network congestion is another real-world aspect that affects QoS. The difficulty in this case is that the in-network operator placement and the push–pull application may cause themselves heavy traffic, leading to network congestion. Extensions of our techniques will account for network congestion by incorporating (inbound, outbound) bandwidth capacity limits per link during geo-distributed CEP evaluation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has received funding from the EU Horizon 2020 research and innovation program INFORE under grant agreement No 825070.

Appendix. Communication cost & latency computation

Consider a solution provided by our algorithms which includes, for each operator Op_i of the EDG, a placement at a network site S_j and a prescribed push–pull strategy, expressed via a corresponding NFA, for evaluating the operator (see Section 3.2). The question is how the communication cost of this operator is formed. Before we formally present our cost calculation formulas, we discuss the intuition behind them. The first factor to consider is that all events of certain event types that are input to the operator may need to travel through the network to site S_j. Each such event type is included in a state k of the prescribed NFA. Therefore, each state k of the adopted NFA has a communication cost which we term below as *State Cost*. This State (communication) Cost is composed of (i) the amount (frequency) of, relevant to k, events that are produced within a window interval W, (ii) the times these events need to be re-transmitted while following the routing paths (hops) of the network from the sites that produce the events to S_j. The second factor that should be accounted for, is that the events of a state k are pushed to S_j conditional upon the occurrence of other events that are included in all previous states of the NFA. Therefore, the aforementioned State Cost is not always charged. It is charged when, withing W, all events of the previous NFA states have occurred somewhere in the network. This happens with a certain probability of reaching a point in W at which the events of state k need to actually be pushed to S_j. We term this probability as *State Reachability Probability*. The latter probability multiplied by the State Cost quantifies the expected communication cost of a particular NFA state. Summing up these costs up to the final NFA state forms the overall communication cost of the operator. Moreover, we need to account for NFA states and, thus, event types that are shared among multiple operators placed at the same site.

To formalize the above discussion, we define a numbering on the possible NFAs with $M \geq 1$ states and let $A(Op_i, M, \mu)$ the μth possible $M-$state NFA that is admissible by Op_i. Let $e \in A(Op_i, M, \mu, k)$ to denote that an event $e \in Op_i$ is input to a particular state $k \leq M$ of the μth NFA admissible by Op_i. Recall that the OR and AGGREGATION operators have exactly 1 state, while the AND and SEQ operators may have a maximum number of steps equal to the number of input events. In this section, for symbol uniformity, we abusively assume an identity operator mapping PEs to themselves, i.e., $Op_a = a$ if $a \in E$.

The **State Cost (SC)**, for a candidate placement at $S_j \in S$ and NFA (push–pull) alternative, is the event rate within W time units triggering a push–pull step corresponding to state $A(Op_i, M, \mu, k)$. We use M', μ' to denote the fact that the NFA participating in the recursive formula may be of a different number of states and numbering compared to NFA $A(Op_i, M, \mu)$:

$$\begin{aligned} &SC(S_j, A(Op_i, M, \mu, k)) \\ &= \sum_{\substack{\forall Op_a \in A(Op_i, M, \mu, k), \\ \forall S_b \in S : Op_a \mapsto S_b}} hops(S_j, S_b) \\ &\cdot \sum_{\substack{\forall A(Op_a, M', \mu', l) \in \\ A(Op_a, M', \mu')}} SC(S_b, A(Op_a, M', \mu', l)) \cdot 2 \cdot W \end{aligned}$$

The above formula says that in order to compute the state cost for Op_i that has been placed at S_j, being evaluated using the μth possible $M-$state NFA that is admissible by Op_i, we do the following: for every operator Op_α (primitive or complex event) that is input to the kth state of the NFA and is placed at site S_b, we sum up the product of the hops in the minimum latency path connecting S_j, S_b multiplied by the total state cost of the NFA based on which Op_α is evaluated at S_b. The latter is the overall event rate of Op_α stemming from S_b. S_b is unique for operators (complex events) because we examine solutions that place them at a particular site, but primitive events may originate from multiple sites and therefore the $\forall S_b \in S : Op_a \mapsto S_b$ in the first summation. Having received events at previous NFA states, the pull request will ask for events occurring W time units before or after the already pushed events. This explains the $2 \cdot W$ factor above.

Notice that in case of an Op_i receiving a single primitive event $Op_a = a$, we are going to (virtually) have a 1-state NFA triggered with the frequency of the PE. Then, the first summation will simply perform an addition over all sites that receive PE a. The subsequent summation, because we have a 1-state NFA, will reduce to $f(a, S_b)$. Therefore, overall we get a summation of $hops(S_j, S_b) \cdot f(a, S_b)$ terms for all sites where $Op_a = a$ appears, i.e., $\forall S_b \in S : Op_a \mapsto S_b$. This is equivalent to the hoped frequency for host site S_j, $hf(a, Sj)$ as defined in Section 4.1.

The **State Reachability Probability (SRP)** is the probability of reaching state $A(Op_i, M, \mu, k)$ of a given NFA. This is a function of the frequency of the event(s) that activate the state $A(Op_i, M, \mu, k)$ (in case of NFA states triggered by other operators – CEs this entails that they should reach their final state), as well as the frequency of events that are supposed to have activated all previous NFA states, i.e., $\{A(Op_i, M, \mu, m)\}_{m<k}$, within W time units:

$$\begin{aligned} &SRP(A(Op_i, M, \mu, k)) \\ &= \sum_{\forall Op_l \in A(Op_i, M, \mu, k)} SRP(A(Op_l, M', \mu', M')) \\ &\cdot \prod_{\forall Op_a \in \bigcup_{m=1}^{k-1} A(Op_i, M, \mu, m)} SRP(A(Op_a, M'', \mu'', M'')) \cdot W \end{aligned}$$

Notice that in case of a PE, where we (virtually) have a 1-state NFA that is activated based on the event arrival rate, *SRP* trivially reduces to $F(a)$, i.e., the global frequency of PE $Op_a = a$ (Section 4.1). Moreover, in case an event is shared by a set of operators assigned to S_j, it will simultaneously appear on transition edges of multiple NFAs. In this case the state will be activated if it is activated in *at least one* NFA. For simplicity, we assume that in case states in NFAs of different operators share at least one input event, $A(Op_i, M, \mu, k)$ is shared in its entirety. Therefore, the SRP for shared event inputs, termed $SSRP(S_j, A(Op_i, M, \mu, k))$, among operators simultaneously placed on site $S_j \in S$, will be given (we only describe the rationale of calculation to avoid cumbersome formulas) by $1 - \prod(1-\beta_l)$ where β_l the SRP in any NFA examined at S_j that includes that state. For communication cost calculation we further need $Srd(S_j, A(Op_i, M, \mu, k))$ to denote the degree of sharing of an examined state at site S_j.

Communication Cost. The cost for a particular Op_i being placed at $S_j \in S$ and executed based on a particular push–pull strategy (NFA) is given by:

$$cs^{\mu}_{i,j}.cost = \sum_{\substack{\forall A(Op_i, M, \mu, k) \\ \in A(Op_i, M, \mu)}} \frac{SC(S_j, A(Op_i, M, \mu, k))}{Srd(S_j, A(Op_i, M, \mu, k))} \cdot SSRP(S_j, A(Op_i, M, \mu, k))$$

Division by *Srd* is required to charge the cost of shared operators only once, upon being co-located at S_j. In case of absence of shared events, $Srd(S_j, A(Op_i, M, \mu, k)) = 1$ and $SSRP(S_j, A(Op_i, M, \mu, k)) = SRP(A(Op_i, M, \mu, k))$.

Network Latency. Again, consider a solution provided by our algorithms which includes, for each operator Op_i of the EDG, a placement at a network site S_j and a prescribed push–pull strategy, expressed via a corresponding NFA. Each state k of the prescribed NFA receives a number of input event types and the network latency of that state is the maximum among the latencies of its input events. This is because, for k to be activated given that its input events have occurred somewhere in the network, in the worst case, it needs to wait an amount of time proportional to the latency of the slowest network path followed by input data of an event type. The aforementioned maximum latency, on the other hand, is formed by the time the pull request needs to reach sites with relevant data and an equivalent amount of time required for S_j to receive a reply with relevant, to the pull request, events. In addition, the overall latency up to the point of reaching k incorporates the latency of the previous state or, if the input to k is a CE produced by another operator of higher latency, the latency added by that operator for reaching the final state of its own NFA. Given these, the latency of the candidate solution is the latency accumulated at the final state of the NFA prescribed for Op_i, placed at S_j.

More formally, the latency for a particular Op_i being placed at $S_j \in S$ and executed based on a particular push–pull strategy (NFA) is computed as follows. Let $ML_{jb}(Op_a)$ express an aggregated latency value for event Op_a, defined as :

$$ML_{jb}(Op_a) = \begin{cases} \sum_{\forall (S_\phi, S_\psi) \in m\ell path_{j-b}} \ell_{\phi-\psi} & , Op_a \in Op \\ mp\ell_{a,j} & , Op_a = a \in E \end{cases}$$

The above holds due to the fact that in case of $Op_a = a$, i.e., $a \in E$, the PE may be produced in a variety of sources. Then, site S_j that gathers relevant events needs to wait an amount of time proportional to the maximum of the latency among the sources (more than one S_bs). This is $mp\ell_{a,j}$ (Section 4.1). In case of $Op_a \in Op$, the CE is derived from a single source where the respective CEP operator has been placed and, thus, the charged latency involves the minimum latency path between sites S_j, S_b derived via $m\ell path_{j-b}$ (Section 4.1).

$$cs^{\mu}_{i,j}.lat = Lat(S_j, A(Op_i, M, \mu, M)) = \max_{\substack{\forall Op_a \in A(Op_i, M, \mu, M), \\ S_b \in S : Op_a \mapsto S_b}} \left\{ \begin{array}{c} ML_{jb}(Op_a) + max\{Lat(S_b, A(Op_a, M', \mu', M')), \\ ML_{jb}(Op_a) + Lat(S_j, A(Op_i, M, \mu, M-1))\} \end{array} \right\}$$

Notice that $Lat(S_b, A(Op_a, M', \mu', M'))= 0$ for PEs: $Op_a = a$ for $a \in E$ since their (virtually) 1-state NFA reaches its final state simultaneously with the PEs occurrence. Moreover, we note that $Lat(S_j, A(Op_i, M, \mu, 1))$ involves a maximum of $ML_{jb}(Op_a) + Lat(S_b, A(Op_a, M', \mu', M'))$ for event types set in push mode (first NFA state), since there is neither a pull request causing latency while being sent nor a previous NFA state. Those aside, the first ML_{jb} accounts for the latency of sending the pull request. If Op_a is indeed an operator outputting CEs, $Lat(S_b, A(Op_a, M', \mu', M'))$ expresses the latency required for Op_a to reach its final state M'. $ML_{jb}(Op_a) + Lat(S_j, A(Op_i, M, \mu, M-1))$ expresses the latency of reaching the previous state $(M-1)$ of Op_i and receiving the response from S_b to S_j for Op_a. It is trivial to see that if $Lat(S_b, A(Op_a, M', \mu', M')) > ML_{jb}(Op_a) + Lat(S_j, A(Op_i, M, \mu, M-1))$ then $Lat(S_b, A(Op_a, M', \mu', M'))$ is the latency factor being charged because, until state M' is reached, $A(Op_i, M, \mu, M-1)$ has been activated and the pull request has been sent. On the other hand, if $Lat(S_b, A(Op_a, M', \mu', M')) \leq ML_{jb}(Op_a) + Lat(S_j, A(Op_i, M, \mu, M-1))$ then the term $ML_{jb}(Op_a) + Lat(S_j, A(Op_i, M, \mu, M-1))$ is the latency being charged because $A(Op_a, M', \mu', M')$ has been activated and awaits for the pull request from S_j.

References

[1] M. Dayarathna, S. Perera, Recent advancements in event processing, ACM Comput. Surv. 51 (2) (2018) 33.

[2] O. Etzion, P. Niblet, Event Processing in Action, Manning Publications Co, 2011.

[3] N. Giatrakos, A. Artikis, A. Deligiannakis, M. Garofalakis, Complex event recognition in the big data era, Proc. VLDB Endow. 10 (12) (2017) 1996–1999, http://dx.doi.org/10.14778/3137765.3137829.

[4] M. Akdere, U. Cetintemel, N. Tatbul, Plan-based complex event detection across distributed sources, in: Proceedings of VLDB, 2008.

[5] Y. Mei, S. Madden, Zstream: A cost-based query processor for adaptively detecting composite events, in: Proceedings of SIGMOD, 2009.

[6] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, M. Mock, Issues in complex event processing: Status and prospects in the big data era, J. Syst. Softw. 127 (2017) 217–236.

[7] A. Adi, O. Etzion, Amit - the situation manager, VLDB J. 13 (2) (2004) 177–203.

[8] R.S. Barga, J. Goldstein, M. Ali, M. Hong, Consistent streaming through time: A vision for event stream processing, in: Proceedings of 3rd Biennial Conference on Innovative DataSystems Research, CIDR, 2007.

[9] A. Demeers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. White, Cayuga: A general purpose event monitoring system, in: Proceedings of 3rd Biennial Conference on Innovative DataSystems Research, CIDR, 2007.

[10] O. Poppe, C. Lei, E.A. Rundensteiner, D. Dougherty, Context-aware event stream analytics, in: Proceedings of the 19th International Conference on Extending Database Technology, EDBT, EDBT '16, 2016, pp. 413–424, URL https://openproceedings.org/2016/conf/edbt/paper-112pdf.

[11] Y. Qi, L. Cao, M. Ray, E.A. Rundensteiner, Complex event analytics: Online aggregation of stream sequence patterns, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, ACM, New York, NY, USA, 2014, pp. 229–240, URL http://doi.acm.org/101145/25885552593684.

[12] E. Rabinovich, O. Etzion, A. Gal, Pattern rewriting framework for event processing optimization, in: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, DEBS '11, ACM, New York, NY, USA, 2011, pp. 101–112, URL http://doi.acm.org/101145/20022592002277.

[13] M. Ray, C. Lei, E.A. Rundensteiner, Scalable pattern sharing on event streams*, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, ACM, New York, NY, USA, 2016, pp. 495–510, URL http://doi.acm.org/101145/28829032882947.

[14] D. Wang, E.A. Rundensteiner, R.T. Ellison III, Active complex event processing over event streams, Proc. VLDB Endow. 4 (10) (2011) 634–645, http://dx.doi.org/10.14778/2021017.2021021.

[15] E. Wu, Y. Diao, S. Rizvi, High performance complex event processing over streams, in: Proceedings of SIGMOD, 2006.

[16] G. Cugola, A. Margara, Complex event processing with t-rex, J. Syst. Softw. 85 (8) (2012) 1709–1728, http://dx.doi.org/10.1016/j.jss.2012.03.056.

[17] G. Cugola, A. Margara, Low latency complex event processing on parallel hardware, J. Parallel Distrib. Comput. 72 (2) (2012) 205–218.

[18] M. Hirzel, Partition and compose: Parallel complex event processing, in: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS '12, 2012, pp. 191–200, URL http://doi.acm.org/101145/23354842335506.

[19] R. Mayer, C. Mayer, M.A. Tariq, K. Rothermel, Graphcep: Real-time data analytics using parallel complex event and graph processing, in: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, 2016, pp. 309–316.

[20] R. Mayer, M.A. Tariq, K. Rothermel, Minimizing communication overhead in window-based parallel complex event processing, in: Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems, DEBS '17, ACM, New York, NY, USA, 2017, pp. 54–65, URL http://doi.acm.org/101145/30937423093914.

[21] N.P. Schultz-Møller, M. Migliavacca, P. Pietzuch, Distributed complex event processing with query rewriting, in: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS '09, ACM, New York, NY, USA, 2009, pp. 4:1–4:12, URL http://doi.acm.org/101145/16192581619264.

[22] L. Woods, J. Teubner, G. Alonso, Complex event detection at wire speed with fpgas, Proc. VLDB Endow. 3 (1–2) (2010) 660–669.

[23] E. Zeitler, T. Risch, Massive scale-out of expensive continuous queries, Proc. VLDB Endow. 4 (11) (2011) 1181–1188.

[24] S.R. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, Tinydb: an acquisitional query processing system for sensor networks, ACM Trans. Database Syst. 30 (2005) 122–173.

[25] Microsoft, Azure Stream Analytics, https://azure.microsoft.com/en-us/services/stream-analytics/, [Online]. (Accessed 24-November-2018).

[26] Microsoft, Bandwidth Pricing Details, https://azure.microsoft.com/en-us/pricing/details/bandwidth/, [Online]. (Accessed 24-November-2018).

[27] Esperonstorm, https://github.com/tomdz/storm-esper.

[28] Ibm proactive technology online on storm, https://github.com/ishkin/Proton/.

[29] Microsoft, Azure HDInsight Documentation, https://docs.microsoft.com/en-us/azure/hdinsight/, [Online]. (Accessed 24-November-2018).

[30] Google, Google Cloud Platform Pricing Calculator, https://cloud.google.com/products/calculator/, [Online]. (Accessed 24-November-2018).

[31] Google, Stream Analytics Solutions, https://cloud.google.com/solutions/big-data/stream-analytics/, [Online]. (Accessed 24-November-2018).

[32] WSO2, WSO2 Complex Event Processor, https://wso2.com/products/complex-event-processor/, [Online]. (Accessed 24-November-2018).

[33] Amazon, Cloud Services Pricing – Amazon Web Services (AWS), https://aws.amazon.com/pricing/services/, [Online]. (Accessed 24-November-2018).

[34] I. Flouris, V. Manikaki, N. Giatrakos, A. Deligiannakis, M. Garofalakis, et al., Complex event processing over streaming multi-cloud platforms: The ferari approach: Demo, in: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, 2016, pp. 348–349.

[35] I. Flouris, V. Manikaki, N. Giatrakos, A. Deligiannakis, M. Garofalakis, e. al, Ferari: A prototype for complex event processing over streaming multi-cloud platforms, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, 2016, pp. 2093–2096.

[36] J. Chen, L. Ramaswamy, D.K. Lowenthal, S. Kalyanaraman, Comet: Decentralized complex event detection in mobile delay tolerant networks, in: 2012 IEEE 13th International Conference on Mobile Data Management, IEEE, 2012, pp. 131–136.

[37] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: Proceedings of the 22Nd International Conference on Data Engineering, ICDE '06, IEEE Computer Society, Washington, DC, USA, 2006, p. 49, http://dx.doi.org/10.1109/ICDE.2006.105.

[38] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator placement for distributed stream processing applications, in: Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, DEBS '16, ACM, New York, NY, USA, 2016, pp. 69–80, URL http://doi.acm.org/101145/29332672933312.

[39] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, L. Fleischer, Soda: An optimizing scheduler for large-scale stream-based distributed computer systems, in: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag New York, Inc., 2008, pp. 306–325.

[40] P.R. Pietzuch, B. Shand, J. Bacon, A framework for event composition in distributed systems, in: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware, Middleware '03, 2003, pp. 62–82, URL http://dl.acm.org/citation.cfm?id=15159151515921.

[41] B. Schilling, B. Koldehofe, K. Rothermel, Efficient and distributed rule placement in heavy constraint-driven event systems, in: Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications, HPCC '11, 2011, pp. 355–364, http://dx.doi.org/10.1109/HPCC.2011.53.

[42] Y. Ahmad, U. Çetintemel, Network-aware query processing for stream-based applications, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, VLDB '04, VLDB Endowment, 2004, pp. 456–467.

[43] B. Gedik, H. Andrade, K.-L. Wu, P.S. Yu, M. Doo, Spade: the system s declarative stream processing engine, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, 2008, pp. 1123–1134.

[44] G.G. Koch, B. Koldehofe, K. Rothermel, Cordies: Expressive event correlation in distributed systems, in: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS '10, 2010, pp. 26–37, http://dx.doi.org/10.1145/1827418.1827424.

[45] G. Li, H.-A. Jacobsen, Composite subscriptions in content-based publish/subscribe systems, in: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware, Middleware '05, 2005, pp. 249–269, URL http://dl.acm.org/citation.cfm?id=15158901515903.

[46] V. Kumar, B.F. Cooper, K. Schwan, Distributed stream management using utility-driven self-adaptive middleware, in: Autonomic Computing, 2005 ICAC 2005 Proceedings. Second International Conference on, IEEE, 2005, pp. 3–14.

[47] S. Rizou, Concepts and algorithms for efficient distributed processing of data streams, University of Stuttgart. http://dx.doi.org/10.18419/opus-3209.

[48] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, I. Stoica, Low latency geo-distributed data analytics, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, ACM, New York, NY, USA, 2015, pp. 421–434, URL http://doi.acm.org/101145/27859562787505.

[49] A. Rabkin, M. Arye, S. Sen, V.S. Pai, M.J. Freedman, Aggregation and degradation in jetstream: Streaming analytics in the wide area, in: 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 14, USENIX Association, Seattle, WA, 2014, pp. 275–288, URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin.

[50] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the borealis stream processing engine, in: Proceedings of the Conference on Innovative DataSystems Research, CIDR, 2005.

[51] M. Balazinska, H. Balakrishnan, M. Stonebraker, Contract-based load management in federated distributed systems, in: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation, vol. 1, NSDI'04, USENIX Association, Berkeley, CA, USA, 2004, p. 15, URL http://dl.acm.org/citation.cfm?id=12511751251190.

[52] A. Chatzistergiou, S.D. Viglas, Fast heuristics for near-optimal task allocation in data stream processing over clusters, in: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM '14, ACM, New York, NY, USA, 2014, pp. 9–1588, URL http://doi.acm.org/101145/26618292661882.

[53] X. Gu, P.S. Yu, K. Nahrstedt, Optimal component composition for scalable stream processing, in: Distributed Computing Systems, 2005 ICDCS 2005 Proceedings. 25th IEEE International Conference on, IEEE, 2005, pp. 773–782.

[54] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, M.J. Franklin, Flux: An adaptive partitioning operator for continuous query systems, in: Data Engineering, 2003 Proceedings. 19th International Conference on, IEEE, 2003, pp. 25–36.

[55] Y. Zhou, B.C. Ooi, K.-L. Tan, J. Wu, Efficient dynamic operator placement in a locally distributed continuous query system, in: OTM Confederated International Conferences on the Move to Meaningful Internet Systems, Springer, 2006, pp. 54–71.

[56] E. Kalyvianaki, W. Wiesemann, Q.H. Vu, D. Kuhn, P. Pietzuch, Sqpr: Stream query planning with reuse, in: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 840–851, URL http://dx.doi.org/10.1109/ICDE.20115767851.

[57] A. Vulimiri, C. Curino, P.B. Godfrey, T. Jungblut, J. Padhye, G. Varghese, Global analytics in the face of bandwidth and regulatory constraints, in: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, USENIX Association, Oakland, CA, 2015, pp. 323–336, URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/vulimiri.

[58] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopulos, N. Mamoulis, A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement, J. Comput. System Sci. 79 (3) (2013) 349–368.

[59] U. Srivastava, K. Munagala, J. Widom, Operator placement for in-network stream query processing, in: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, PODS '05, 2005, pp. 250–258, URL http://doi.acm.org/101145/10651671065199.

[60] L. Ying, Z. Liu, D. Towsley, C.H. Xia, Distributed operator placement and data caching in large-scale sensor networks, in: INFOCOM 2008 the 27th Conference on Computer Communications. IEEE, IEEE, 2008, pp. 977–985.

[61] L. Amini, N. Jain, A. Sehgal, J. Silber, O. Verscheure, Adaptive Control of Extreme-Scale Stream Processing Systems, in: Distributed Computing Systems, 2006 ICDCS 2006 26th IEEE International Conference on, IEEE, 2006, p. 71.

[62] A. Benzing, B. Koldehofe, K. Rothermel, Efficient support for multi-resolution queries in global sensor networks, in: Proceedings of the 5th International Conference on Communication System Software and Middleware, ACM, 2011, p. 11.

[63] T. Repantis, X. Gu, V. Kalogeraki, Synergy: Sharing-aware component composition for distributed stream processing systems, in: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Springer-Verlag New York, Inc, 2006, pp. 322–341.

[64] F. Starks, V. Goebel, S. Kristiansen, T. Plagemann, Mobile distributed complex event processing – ubi sumus? quo vadimus? in: Mobile Big Data, Springer, 2018, pp. 147–180.

[65] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, R. Grimm, A catalog of stream processing optimizations, ACM Comput. Surv. 46 (4) (2014) 46.

[66] N. Cipriani, M. Eissele, A. Brodt, M. Grossmann, B. Mitschang, Nexusds: a flexible and extensible middleware for distributed stream processing, in: Proceedings of the 2009 International Database Engineering & Applications Symposium, ACM, 2009, pp. 152–161.

[67] M. Weidlich, H. Ziekow, A. Gal, J. Mendling, M. Weske, Optimizing event pattern matching using business process models, IEEE Trans. Knowl. Data Eng. 26 (11) (2014) 2759–2773.

[68] A.Y. Halevy, Answering queries using views: A survey, VLDB J. 10 (4) (2001) 270–294.

[69] A. Labrinidis, N. Roussopoulos, Balancing performance and data freshness in web database servers, in: VLDB, 2003, pp. 393–404.

[70] L. Bright, A. Gal, L. Raschid, Adaptive pull-based policies for wide area data delivery, ACM Trans. Database Syst. 31 (2) (2006) 631–671.

[71] V. Peralta, Data Freshness and Data Accuracy: a State of the Art, Instituto de Computacion, Facultad de Ingenieria, Universidad de la Republica.

[72] A. Silberstein, J. Terrace, B.F. Cooper, R. Ramakrishnan, Feeding frenzy: selectively materializing users' event feeds, in: SIGMOD, 2010, pp. 831–842.

[73] J. Agrawal, Y. Diao, D. Gyllstrom, N. Immerman, Efficient pattern matching over event streams, in: Proceedings of SIGMOD, 2008.

[74] Flink, https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html, [Online]. (Accessed 24-November-2018).

[75] WSO2, Creating a Storm Based Distributed Execution Plan, https://docs.wso2.com/display/CEP410/Creating+a+Storm+Based+Distributed+Execution+Plan, [Online]. (Accessed 24-November-2018).

[76] O. Poppe, C. Lei, S. Ahmed, E.A. Rundensteiner, Complete event trend detection in high-rate event streams, in: Proceedings of the 2017 ACM International Conference on Management of Data, ACM, 2017, pp. 109–124.

[77] Heron, https://apache.github.io/incubator-heron/docs/migrate-storm-to-heron/, [Online]. (Accessed 24-November-2018).

[78] Flink, https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/libs/storm_compatibility.html, [Online]. (Accessed 24-November-2018).

[79] Ignite, https://apacheignite-mix.readme.io/v2.7/docs/storm-streamer, [Online]. (Accessed 24-November-2018).

[80] I. inetats, stock trade traces., http://davis.wpi.edu/dsrg/stockData/eventstream3.txt, [Online]. (Accessed 24-November-2018).