Issues in Complex Event Processing Systems

Ioannis Flouris
School of Electronic and Computer Engineering
Technical University of Crete
Chania, Greece
Email: gflouris@softnet.tuc.gr

Nikos Giatrakos
School of Electronic and Computer Engineering
Technical University of Crete
Chania, Greece
Email: ngiatrakos@softnet.tuc.gr

Minos Garofalakis
School of Electronic and Computer Engineering
Technical University of Crete
Chania, Greece
Email: minos@softnet.tuc.gr

Antonios Deligiannakis
School of Electronic and Computer Engineering
Technical University of Crete
Chania, Greece
Email: adeli@softnet.tuc.gr

***Abstract*—Many Big Data technologies were built to enable the processing of human generated data, setting aside the enormous amount of data generated from Machine-to-Machine (M2M) interactions. M2M interactions create real-time data streams that are much more structured, often in the form of series of event occurrences. In this paper, we provide an overview on the main research issues confronted by existing Complex Event Processing (CEP) techniques, as a starting point for Big Data applications that enable the monitoring of complex event occurrences in M2M interactions.**

I. Introduction

Big Data is generally characterized by four main aspects: Volume, which is the enormous amounts of data to be processed; Velocity, which is the speed at which data must be processed; Variety, which represents the multiple representations in the data model; and Veracity, which is the uncertainty in the data. Many modern Big Data applications aim to enhance the processing of human generated data which are though surpassed in volume by data produced during Machine-to-Machine (M2M) interactions.

M2M data are generated in high frequency in every Big Data system and include useful information that can be utilized to identify the occurrence of interesting situations. Complex Event Processing (CEP) systems aim to process M2M data efficiently and immediately recognize interesting situations when they occur. Generally, events can be thought of as single occurrences of interest in time and complex events as situations that comprise a particular composite meaning for the system. CEP applications include, but are not limited to, network health monitoring applications, mobile and sensor networks, computer clusters and smart energy grids, security attacks detection like denial-of service attacks, intrusions or fake identities. In the business sector, accounting, logistics, warehousing and stock trading applications are included among others.

In this paper, we aim to unravel the main research issues tackled in the literature of CEP systems and shortly present the contribution of developed techniques. Figure 1 illustrates the main dimensions of our upcoming analysis. The distance of the colored line from the center of the graph is proportional to the depth of our discussion compared to the abundance of related work for each dimension and our notion of significance. Details on the content of each axis will be given in the sections to come in a clockwise order, as shown by the dotted arc in the figure.

This work was partially supported by the European Commission under ICT-FP7-FERARI-619491 (Flexible Event pRocessing for big dAta aRchItectures)

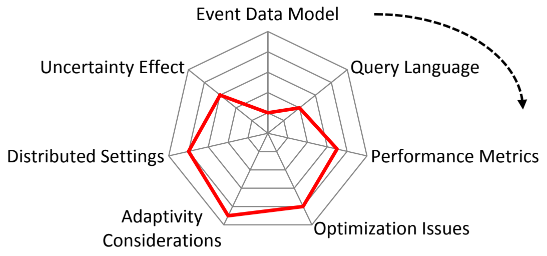

Fig. 1: Dimensions & Depth of our Analysis

II. Event Stream Model

Event tuples are defined by [11] as tuples $e =< s, t >$, where e represents the event of interest, s refers to a list of attributes and t is a list of timestamps, the first being the start of the event and the last the end of it. [2] and [1] define events based on a specific schema using different syntax. [16] and [18] define an event as a single occurrence of interest in time. In any case, events should be defined with a number of attributes denoting the event's meaning, along with one or more time attributes denoting the event's time occurrence and/or its duration. Most research efforts [2], [8], [11], [16] categorize events as primitive and composite (also called complex or deferred). Primitive events are atomic (i.e., non-decomposable) occurrences of interest. Composite (or complex) events are detected from the CEP system based on defined patterns (rules) involving primitive and/or other complex events.

III. Query Formulation and Event Detection

Queries submitted in a CEP system aim at detecting complex events on the incoming stream of primitive events based on a given pattern. A pattern includes one or more event types (or classes). For instance, at the bottom of Figure 2, the received event stream includes instances of event types in the set {A,B,C,D}. The most common expression of complex event queries receives the following form [1], [16], [18], [17]:

```
PATTERN OPERATOR(list of TypeOfEvents)
WHERE (event value constraint)
AND (another value constraint)
WITHIN (time)
RETURN (ComplexEvent or Events to return)
```

The prominent operators in the PATTERN OPERATOR clause are {SEQ, AND, OR, NOT, Kleene Closure}. In particular the SEQ pattern requires that all events occur *sequentially*, the AND operator declares conjunction of *all* events included in the pattern, OR corresponds to the occurrence of *any* event, while the Kleene Closure operator means *zero or more* individual occurrences for an event included in the pattern. The WHERE clause is for one or more (in)equality constraints among event attributes separated by logical operators. The WITHIN clause specifies the time window in which all events must occur for a full pattern match and thus, a valid complex event detection. The RETURN clause outputs a complex event or its list of primitive events.

As streaming events arrive on a CEP system, a query pattern is progressively evaluated. This means that before complex event detection via a full pattern match, partial matches of the query pattern take place. These partial matches need to be monitored as they express the potential for an imminent full match. To track the state of a partial match and determine on a complex event occurrence, multiple representations have been used in the literature, mainly depending on the performance metrics that each approach tries to enhance. The basic two categories are graph-based and automata-based representations.

Automata-based Non-Deterministic Finite Automata (NFA) (see Fig. 2) is one of the most popular model representations in CEP systems where the status of a partial match corresponds to a state of the automaton. The NFA model is used in [16], while a variant that uses a match buffer and called NFA^b is used in [1] and [18]. The Finite State Machine (FSM) model [11], [2] is very similar to the NFA. FSMs carry similar functionality with the NFA, where state transitions are made when an event, that satisfies the query's predicates, is detected and that its states signal the partial or full detection of a complex event.

Graph-based Various tree structures have been used to facilitate the detection of complex events. *Left and right deep trees* are used in [8] while [7] uses *petri-nets* for event detection. *Event detection graphs* are deployed in [2] to merge all existing complex event expressions in a single structure. The leaves are primitive events and the internal nodes are operators. This structure offers a general overview of defined complex expressions and serves as a complementary data structure that aims to describe event and pattern dependencies. Event Processing Network (EPN) is another way of modeling the processing for detecting complex events as proposed in [6]. An event processing agent (EPA) is a processing element that applies logic to incoming events and outputs derived (complex) events. The EPN follows an event driven logic where EPAs communicate asynchronously through event channels (EC).

IV. Performance Metrics

CEP systems that assume that all data is delivered in a single source for processing, measure their performance based on how fast they process incoming data. Hence, the main optimization metric is **Throughput** i.e., how many events per time unit are processed [1], [8], [11], [16], [18]. Another performance metric, highly correlated with throughput, is the **time or CPU cost** of the language operators [1], [18].

CEP systems, that assume a collection of streams arrive to different computing sites having a central (coordinator site) which dictates (based on a plan) which primitive event is monitored in which site, opt for the reduction of the **Transmission cost** under the counterweight of detection latency [2]. In such cases, **Detection Latency** is the time between the occurrence of a complex event and its detection from the coordinator.

Lastly, an important issue upon handling large time windows is that of **Memory Management**, since, if proper care is not taken, intermediate results or partial matches stored in main memory may excessively grow.

V. A Taxonomy of CEP Approaches

The architectural scheme of the techniques presented in this section entails a sole event stream received by a single operating site. Thus, the main performance goals involve high throughput or low CPU cost and memory consumption.

A. Predicate-Related Optimizations

A simple way to optimize performance is to evaluate predicates and time windows, the *WHERE* and *WITHIN* clauses of the query, early in a query execution plan. This can be achieved by partitioning the input stream and by employing early filtering in the selected events that will actually be part of complex event detection based on the query. Other approaches include on-the-fly hash-based lookups and early filters deployment on value predicates as events arrive.

1) Pushing Predicates Down: One of the first complete CEP framework was SASE [16]. SASE employs this filtering technique by partitioning an event stream to many small ones, where events in each partition have the same value for the attribute used in a single equivalence test. Before we explain how it works we should present the basics of SASE's proposed query plan. The basic component of the query plan is sequence scan and a construction called SSC. SSC transforms a stream of events into a stream of sequences, with each sequence being a partial match of the query. For sequence scan they use Non-Deterministic Finite Automata to represent the structure of an event sequence. In sequence scan, for each partial match, an NFA is created by mapping successive event types to successive NFA states. To keep track of all simultaneous states, a runtime stack is instantiated to record the set of active states and how this set leads to a new set of active states, as an event arrives, with the use of pointers (see Fig. 2). Sequence construction is invoked when an accepting state is reached during sequence scan. Sequence construction is performed by extracting from the runtime stack a single source Directed Acyclic Graph (DAG). The DAG starts at an instance of the accepting state in the rightmost cell of the stack and traverses back along the predecessor pointers, until reaching instances of the starting state.

One simple solution for the **single equivalence test** is to partition the stream first and then run the query plan bottom-up for each partition, but the authors propose a more elaborate approach. Using an auxiliary data structure, called Partitioned Active Instance Stack or PAIS (see Fig. 2), they simultaneously create the partitions and build a series of these stacks for each partition without incurring any overhead to the events that do not participate in the query. With PAIS sequence

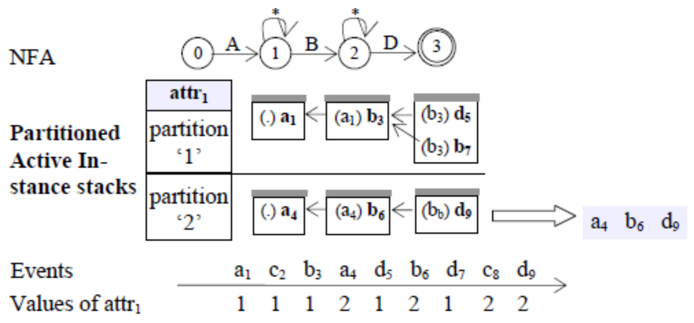

Fig. 2: PAIS example from [16]

construction is only performed in stacks in the same partition, thus producing fewer intermediate results.

When **multiple equivalence tests** are present in a query the authors propose two different filtering approaches. The first called *Multi-PAIS* performs aggressive cross-attribute (NFA) state transition filtering in sequence scan, but this technique fails to prune superfluous results in sequence construction. The second approach, called *Dynamic Filtering*, pushes the most selective equivalence test down to sequence scan and then pushes all other tests to sequence construction. This approach cannot filter as many events in sequence scan, thus having more instances in the stacks, but does not need to pay the overhead of cross-attribute transition filtering and multi-stack maintenance.

The same principle can be applied to simple predicates and time windows. Again, there are two approaches for **pushing time windows down** which are not mutually exclusive. The first is to apply windows in sequence scan and the second in sequence creation. The former filters some of the events, so that they are not included in the stacks, and the latter searches those stacks and checks the time window on-the-fly for each event sequence.

2) Postponing with Early Filters: [18] uses an optimization to prune inconsistent events based on value predicates on the kleene closure operator. Although predicate evaluation is basically performed (or postponed) during edge evaluation for state transition in an NFA model, an additional filtering technique is proposed to prune repeating events, based on an existing predicate upon them. [18] advocates that it is incorrect to evaluate all value predicates on the fly as events arrive, since there is the case of non-deterministic actions on the NFA model. Hence, the value predicates are categorized based on their consistency into 4 categories that can be applied to the kleene closure operator: (a) true-value consistent predicates,(b) false-value consistent predicates, (c) true and false-value consistent predicates and (d) inconsistent predicates. Based on these categories, it can be decided by the system whether a predicate can be evaluated on the fly and prune nonconforming events, or such an evaluation is not possible and therefore must be postponed until the result creation phase.

3) Hashing for Equality Predicates: [8] addresses the issue of equality predicates and multi-class equality predicates. It proposes the use of hash-based lookups, whenever it is possible, in an effort to reduce search costs for equality predicates between different event classes. Multi-class predicates can be attached to the operators as other predicates. The mechanism employed involves the use of buffers of events in all nodes of a tree model representation, with the leaves being primitive events and the internal nodes being operators. Based on the equality predicate, a hash table is created in the buffer (of the leaf node that takes part in the equality predicate) as events arrive and when the predicate is employed (further up in the tree) hash lookups are performed to prune results, that do not conform with the predicate.

Hash tables are created on the buffers of leaf nodes, either to some or all the involved events depending on the used operator. If the operator is AND, then a hash-table is created on all participating event buffers. If the operator is SEQ, a hash table is created to the first event in the sequence that takes part in the predicate. Hash construction can be easily extended with **multiple equality predicates**, where the use of a secondary hash table is possible for sequential patterns to facilitate faster pruning of irrelevant events.

B. Buffering Techniques

The use of buffers, to facilitate optimizations, is a common practice among CEP systems. Both [16] and [1] employ the use of stacks upon the NFA states, that evaluate the query upon the arrival of new related events, which are grouped into buffers for each distinct run along the NFA. [8] also deploys buffers upon the query evaluation tree structure's nodes. These buffers enable the use of optimization techniques, in order to enhance the overall system efficiency in both throughput and memory management.

1) Shared Buffer: [1] uses buffers to encode partial and complete matches for each run. The basic principle is to share both storage and processing across multiple runs in the NFA-based query plans. The initial approach is to build a buffer for each single run and then merge such individual buffers into a shared one for all the runs. Each individual buffer contains a series of stacks (similar to Fig. 2), one for each state of the NFA except the final state. Each stack contains pointers to events that triggered a transition in the NFA's state and thus are included into the buffer. Further, each event has a predecessor pointer to the previously selected event in either the same stack or the previous one. For any event that triggered a transition to the final state, a traversal across the predecessor pointers reveals the full detection of a complex event, as specified by the evaluated query.

They consequently combine those buffers into a shared one to reduce the memory and processing overhead. This process is based on merging the corresponding stacks of individual buffers, by merging the same events in those stacks while maintaining the predecessor pointers. This process though can result in erroneous results, since it is unable to distinguish predecessor pointers from different runs. To alleviate this problem they use an identifier number (version) for each individual run to label all such pointers created in that particular run. An additional issue is that runs cannot pre-assign version numbers, since non-deterministic states can spawn new runs at any time. Thus, the version number is dynamically grown as the run proceeds in the form of $id_1.(id_j) * (1 \leq j \leq t)$, where t refers to the current state. This technique guarantees that the version number is compatible with a possibly ancestor run (spawned by it), since they would share a common prefix. The versioned shared buffer can thus encode compactly all possible runs. To extract a detection of a complex event, the algorithm takes the version number of the run and traverses from the most recent event (in the last stack) along the compatible predecessor pointers to the event that started the run.

2) Operator-specific optimizations: The buffers implemented by [8] resemble the stacks used by [1]. They use a buffer in each node of the tree plan to store intermediate results (in the case of internal nodes) and incoming events (in the case of leaves). The buffers store pointers pointing to the primitive event as well as its start and end timestamp. Another important aspect of this buffer is that it is stored in a sorted order, based on the end time. This feature allows the intermediate evaluation of query's time window.

With the use of buffers as described above an optimization is proposed to evaluate conjunction queries (i.e. queries with AND operator). The query evaluation algorithm works like sort-merge join. It maintains a cursor on both input buffers, initially pointing to the oldest not-yet-matched event in each buffer. In each step of the algorithm, it chooses the cursor pointing to the earlier event and combines that event with all earlier events in the other cursor's buffer. It finally produces events in end-time order, since the processing order starts with the earliest event at each time step.

Using the same principles, evaluation algorithms are proposed for all operators. The OR operator simply outputs all events that were allowed inside the buffers. The NOT operator is pushed down to the event selection, rather than negating composite events on top of the query plan. The basic principle in the algorithm is that it tries to prune from the buffers all events that the occurrence of a negated event disqualifies based on the time window of the query. The $Kleene$ closure operator uses the buffers to gather the events specified (either a specific number or all qualified events) within the time window.

C. Query Rewriting/Reordering

Query rewriting is a popular optimization technique that allows a non-optimized query expression to be rewritten to a more efficient one. The rewritten query must produce exactly the same results as the original one and must exhibit enhanced performance upon the optimization objectives. In general, the reordering in queries rearranges the monitoring or the processing of events, while query rewriting transform the query itself using semantically equivalent query with cheaper or fewer operators.

In [11] the issue of query reordering is addressed for three operators $Union$, $Next$ and $Exception$, that have the same functionality as AND, SEQ and NOT, respectively. The main intuition is to transform the original patterns to equivalent ones with lower CPU cost. For the $Union$ operator, since the detection of the complex event relies on detecting one of each involving events without any order, different ordering of the events may result in more efficient evaluation. Based on the commutative and associative property of this operator, the authors map the problem to finding the optimal prefix code for compression of data, i.e., a set of bit strings in which no string is the prefix of the other. A prefix code is also represented by a binary tree, where each internal node is a one or a zero and each leaf represents the frequency of the character. They use the greedy *Huffman* algorithm to create that tree and with a depth first traversal of the tree the optimal order of the $Union$ patterns is generated. The $Next$ operator also has the associative property but not the commutative one, i.e., the order of the events cannot be altered. The authors propose a dynamic programming solution, of cubic time and quadratic space complexity, where the lowest cost pattern can be found by enumerating all equivalent patterns and computing each cost. The same principle applies for the $Exception$ operator and therefore the same algorithm is used.

Query rewriting is tackled in [8] by using algebraic rule-based transformations for all operators. The transformations - performed by the authors - differ from the solution proposed by [11] since the basic transformation is algebraic and can, thus, alter the used operators. E.g. the expression SEQ(A, AND(!B, !C), D) is equivalent with the expression SEQ(A, !OR(B, C), D) where ! is the NOT operator. [8] implements a series of such equivalence rules, that can generate an exponential number of equivalence expressions for a given query pattern. Instead of searching all those expressions exhaustively, the query is transformed when the rewritten expression has a smaller number of operators or the expression contains lower cost operators. The higher cost operator is the AND operator, followed by the SEQ operator, while the cheapest one is the OR operator. After the query is simplified, the authors propose an algorithm for reordering the operators in a similar conceptual way as in [11]. Using a dynamic programming algorithm, they try to find the optimal order of the events, that are included in the query. The optimal order, though, can be found in a set of different (left-deep, right-deep, bushy) tree structures, which are derived from a single logical query plan. They finally use an algorithm that generates quadratic subsets and has cubic time complexity.

In [9] an assertion-based pattern rewriting framework is presented for two operators, namely all (AND) and $sequence$ (SEQ). The patterns are split into disconnected components that can be independently processed. The acquisition of the disconnected components is achieved by converting the pattern into Conjunctive Normal Form (CNF). They proceed by recognizing independent components by creating a variable dependency graph using variables in the WHERE clause of the query as nodes and their connection as edges. Splitting the pattern into maximal number of independent partitions implies the finest granulation that can be performed.

VI. Adaptivity Considerations

Adaptivity can be thought of as the ability of a CEP system to alter the query execution plan at runtime, when it judges (based on recent statistics) that a different plan could improve performance metrics. As a result of potentially high variability in input stream rates and selectivities, an initially optimal plan may no longer be optimal after running for some time.

The first adaptive complex event query processor was introduced by [8] and called ZStream. To recompute the plan on the fly, ZStream maintains a running estimate of event statistics, using sampling operators attached to the leaf buffers. More precisely, it uses simple windowed averages to maintain the rates of each input stream and the selectivity of each predicate. When any statistic used in a plan varies by more than some error threshold, the operator ordering algorithm is rerun and the new plan is installed, if the performance improvement predicted by a cost function is greater than a performance threshold. More sophisticated adaptive strategies may try to reuse some of the already stored intermediate results and minimize the recalculation, or incorporate parallelism when plans are changed. With the use of this technique, the system is able to adjust to changing data distributions, without suffering from the degradation in performance of a poorly selected or outdated query plan.

VII. Distributed Settings

There are two basic insights of how a distributed system can perform. In [11] the idea that is introduced is that distributing the query processing across multiple sites enhances drastically the system's performance, in terms of throughput. On the other hand, in [2] the architectural scheme that is considered has multiple streaming sources and a single base-processing site. Each source is a receiver of an input stream of events and the base site is a coordinator that communicates with all the sources, for detecting complex events.

The basic insight in [11] is that distributing the query operators elevates the problem of memory management, as well as system throughput. Memory management is enhanced, since the partial matches (automaton instances) are distributed through many sites, and are therefore able to deal with larger time windows. Throughput is also optimized, through the fact that each site receives fewer events for fewer queries, thus the overall system is able to process more events per second. The main optimization in this paper is that the queries are reordered to more efficient ones, as described in previous sections.

The processing is distributed with the use of a greedy algorithm for choosing operator deployment plans. The algorithm reuses already deployed operators and deploys the remaining operators in a bottom-up fashion. Existing operators are stored in a hash map for fast retrieval. First, a submitted query is traversed top-down to find the largest equivalent deployed operator in the hash map, if any, starting with the entire query. If found, the expression is replaced with a marker containing the operator identifier and location to allow operators to be connected once deployed. Next, the remaining operators are deployed bottom-up. The location of each operator is selected by recursively placing the left and right sub-expressions of the operator and then the operator itself. An operator is placed by calculating the cost of placing the operator on each site and selecting the lowest cost site. This approach has $O(Q \cdot N)$ time complexity where Q is the number of operators and N is the number of sites. This approach has the advantage of selecting good deployment plans, but the disadvantage that these plans are not optimal.

In [2] the main concern is that the *latency* for detecting the complex events is user (or system) specified and that the *communication cost* for communicating with the coordinator site is controlled. With the above in mind, plans are generated that try to balance the communication cost with the detection latency. They use FSMs for the physical event detection plans and event detection graphs for the logical model representation of all the existing queries. The event detection graph depicts (in a single graph) all available queries, with leaf nodes being primitive events and internal nodes being operators or complex events. Using the event detection graph they are able to create *pareto optimal* plans, that take into account event sharing across multiple queries, in the sense of *Multi-Query Optimization*, as well as event frequencies and acceptable latency values. With these optimal plans being deployed, based on the cost-latency model that they propose, they are able to generate monitoring plans (FSMs), which conform with the chosen cost and latency constraints. These plans are kept in the coordinator site and through push and pull messages the coordinator detects complex events. The activation of the FSM's final state, after the occurrence of all the related events in the way that the query specifies, signals the detection of a complex event.

Based on the number of states of the FSM, they can monitor any number of events that participate in the query. The activation of a new state (through the detection of the current state's primitive event) marks the monitoring of a new set of events. The basic idea behind the number of selected states of an FSM (which is the monitoring order of events) is that; *"processing the higher frequency events conditional upon the occurrence of lower frequency ones eliminates the need to communicate the former in many cases, thus has the potential to reduce communication cost in exchange of increased detection latency"* [2]. In that way, [2] trades communication for latency, by varying the number of FSM states.

The plan generation is done by traversing the event detection graph in depth-first manner, running the plan generation algorithm on each node, in order to create a set of plans with a variety of cost and latency characteristics. At the parent node of each complex event, where all plans are propagated, the selection of the plan marks the selection on the children nodes. This hierarchical plan composition takes also into account shared primitive events by multiple queries, so that it will be taken into consideration at the plan selection. The plan generation algorithm is a Dynamic Programming algorithm that achieves the minimum global cost for a given latency value, but has exponential time complexity and is, thus, only applicable to small problem instances. This is the reason that the authors applied a heuristic algorithm that runs in polynomial time and, although it cannot guarantee optimality, it produces near optimal results in their experimental evaluation.

Plan execution then commences by activating all the FSMs' starting states that trigger the continuous monitoring of some primitive events at the various sites, by informing them which events are of interest by the coordinator. Once such an event is detected at a source site, it is *pushed* to the coordinator who makes the transition in the respective FSM to the next state and then *pulls* from the sites the next primitive event that the FSM's following state monitors. FSMs are non-deterministic since the starting state is always activated and there may be multiple active states at a time. Once a final state is reached, inside the time window specified by the query, the coordinator detects a complex event.

VIII. Handling Uncertainty Aspects

Sources that produce event entities that stream into a CEP system may incorporate imprecisions in the data [4],

[10], [12], [17], [13], [5], [14], [15]. These imprecisions may appear on either the attributes that comprise an event, causing *Content Uncertainty*, or even on an event apparition i.e., *Uncertain Event Occurrence*. A third type of uncertainty, regards *Uncertain Rules* (patterns) quantifying the hesitancy with which application experts define a query pattern that truly signals the occurrence of a composite event.

Content uncertainty is propagated to an event occurrence since the uncertainty on attribute values yields probabilistic decisions on whether these attributes satisfy posed predicates for event's occurrence [5]. In turn, uncertain event occurrences lead to probabilistic pattern matches as the event instances in a full match actually appear with a probability ≤ 1. Uncertainty propagation from attributes to events and among events relies on the assumptions that are made with respect to the dependency of the involved entities. The assumptions existing uncertain CEP techniques employ, in a nutshell, involve [3]: *Independence* of events, *Markovian property* adoption and *Bayesian Network* construction.

Due to the fact that probabilistically enumerating possible event instances and potential partial pattern matches generates an exponential amount of possible worlds i.e., probabilistic (sets of) tuples, processing costs and memory requirements are tremendously increased. To optimize performance, techniques such as those presented in Section V are applicable to filter out inconsistent possible worlds. Still, these techniques do not consider the uncertainty dimension. A form of a predicate that is tailored for probabilistic event handling is that of a *confidence parameter* that can be incorporated in the posed query to filter highly improbable possible worlds given a threshold value. Confidence based pruning is explicitly taken into consideration in [12], [13], but virtually any probabilistic event processing engine is capable of utilizing such a predicate.

Lahar [10] utilizes a combination of query rewriting and sharing techniques to improve CEP performance. A query of interest is broken down to a number of subgoals and query classes are identified based on the variables that are shared amongst the subgoals. Efficient algorithms for processing each query class are accordingly proposed. Of particular interest are the Regular and Extended Regular query classes discussed in [10], due to their ability to be processed in an online fashion. In the Regular query class, the identified subgoals do not share any variables. They are translated to regular expressions using a four step procedure which initially defines a set of symbols on which a simple automaton operates. Then, the set of all possible worlds is translated into a sequence of subsets of the previous symbols, while a third step involves query translation into a regular expression. Finally, the distribution of a Markov chain that is induced by the possible worlds is recovered and used to evaluate the regular expression. Extended Regular queries, share variables across all their subgoals, but can be decomposed to regular queries and be processed independently, propagating uncertainty in the final outcome based on the independence assumption.

IX. Conclusion

Our work identified and shortly reviewed the basic research issues encountered by techniques tailored for CEP systems in the light of their Big Data essence. We focused on the inherent high-rate streaming nature of events, the complexity of query languages and the often distributed network of event generating sources. Thus, we divided our analysis based on the adopted architecture (distributed or not); the optimizations introduced subjected to proper performance metrics (Velocity and Volume); the adaptivity to changing data distributions (Velocity); and the uncertainty aspects (Veracity) studied by each framework.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of SIGMOD*, June 2008.

[2] M. Akdere, U. Cetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. In *Proceedings of VLDB*, August 2008.

[3] E. Alevizos, A. Skarlatidis, A. Artikis, and G. Paliouras. Complex event recognition under uncertainty: A short survey. In *EPForDM, EDBT Workshop*, 2015.

[4] A. Artikis, O. Etzion, Z. Feldman, and F. Fournier. Event processing under uncertainty. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 32–43, New York, NY, USA, 2012. ACM.

[5] G. Cugola, A. Margara, M. Matteucci, and G. Tamburrelli. Introducing uncertainty in complex event processing: model, implementation, and validation. *Computing*, pages 1–42, 2014.

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

[7] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri-nets. In *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS '94)*, pages 2–9, February 1994.

[8] Y. Mei and S. Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of SIGMOD*, June 2009.

[9] E. Rabinovich, O. Etzion, and A. Gal. Pattern rewritting framework for event processing optimization. In *Proceedings of DEBS*, July 2011.

[10] C. Ré, J. Letchner, M. Balazinksa, and D. Suciu. Event queries on correlated probabilistic streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 715–728, New York, NY, USA, 2008. ACM.

[11] N. P. Schultz-Moller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query reqwriting. In *Proceedings of DEBS*, July 2009.

[12] Z. Shen, H. Kawashima, and H. Kitagama. Probabilistic event stream processing with lineage. In *Proceedings of Data Engineering Workshop (DEWS)*, June 2008.

[13] Y. Wang, K. Cao, and X. Zhang. Complex event processing over distributed probabilistic event streams. *Computers & Mathematics with Applications*, 66(10):1808 – 1821, 2013.

[14] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Complex event processing over uncertain data. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 253–264, New York, NY, USA, 2008. ACM.

[15] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Efficient processing of uncertain events in rule-based systems. *Knowledge and Data Engineering, IEEE Transactions on*, 24(1):45–58, Jan 2012.

[16] E. Wu, Y. Diao, and S. Rizvi. High performance copmlex event processing over streams. In *Proceedings of SIGMOD*, June 2006.

[17] H. Zhang, Y. Diao, and N. Immerman. Recognizing patterns in streams with imprecise timestamps. *Proc. VLDB Endow.*, 3(1-2):244–255, sep 2010.

[18] H. Zhang, Y. Diao, and N. Immerman. Optimizing expensive queries in complex event processing. In *Proceedings of SIGMOD*, pages 217–228, June 2014.