Exploiting Sharing Opportunities for Real-time Complex Event Analytics

Elke A. Rundensteiner[1], Olga Poppe[1], Chuan Lei[2], Medhabi Ray[3], Lei Cao[4], Yingmei Qi[5], Mo Liu[6], and Di Wang[7]

[1]Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609

[2]NEC Labs America, 10080 N Wolfe Rd, Cupertino, CA 95014

[3]Microsoft Corporation, 205 108th Ave. NE, Bellevue, WA 98004

[4]IBM T.J.Watson Research Center, 1101 Route 134 Kitchawan Rd, Yorktown Heights, NY 10598

[5]Google, 601 N 34th St, Seattle, WA 98103

[6]Sybase Corporation, 1 Sybase Drive Dublin, CA 94568

[7]Facebook, 1730 Minor Ave, Seattle, WA 98101

{rundenst,opoppe}@cs.wpi.edu, chuan@nec-labs.com, meray@microsoft.com, caolei@us.ibm.com, ymqi@google.com, mo.liu@sybase.com, wangdi@fb.com

Abstract

Complex event analytics systems continuously evaluate massive workloads of pattern queries on high volume event streams to detect and extract complex events of interest to the application. Such time-critical stream-based applications range from real-time fraud detection to personalized health monitoring. Achieving near real-time system responsiveness when processing these workloads composed of complex event pattern queries is their main challenge. In this article, we first review several unique optimization opportunities that we have identified for complex event analytics. We then introduce a family of optimization strategies that consider event correlation over time to maximally leverage sharing opportunities in event pattern detection and aggregation. Lastly, we describe the event-stream transaction model we designed to ensure high performance shared pattern processing on modern multi-core architectures.

1 Introduction

Many streaming systems from sensor networks to financial transaction processing generate high-volume, high-velocity event streams. These events have many dimensions (such as time, location, dollar amount). Each dimension may be hierarchical in nature (such as time measured in years, months, days and so on). In many monitoring applications, it is imperative that a huge workload of expressive event-pattern queries analyze these event streams to detect complex event patterns, aggregate trends and derive actionable insights in near real time.

Motivating Example. Consider an evacuation system where RFID technology is used to track the mass movement of people and goods during natural disasters. Terabytes of RFID data could be generated by such a system. Facing this huge volume of data, an emergency management system must detect and aggregate complex

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

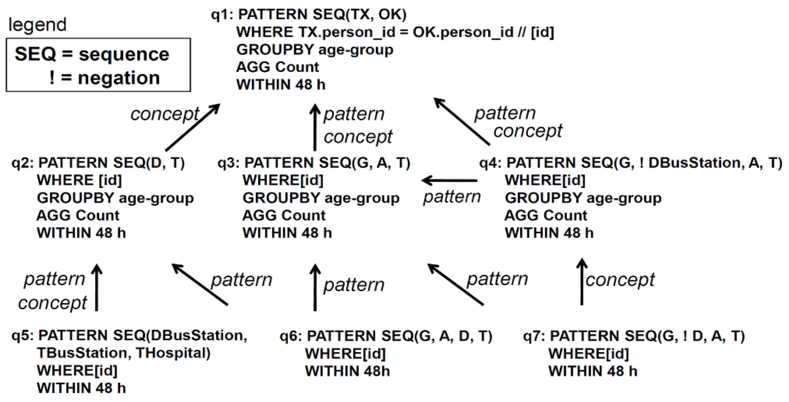


Figure 1: Event pattern queries supporting emergency management at different levels of abstraction [36]

event patterns across multiple dimensions at different granularities in real time. For example, the emergency personnel may monitor people movement as well as traffic patterns of needed resources (such as medicine, food, and blankets) at different levels of abstraction (e.g., bus station, Austin, Texas). Consider Figure 1, where during a hurricane the federal government may monitor people fleeing from Texas to Oklahoma for global resource distribution planning (query q_1); while the local authorities in Dallas may focus on people movement starting from the Dallas bus station, traveling through the Tulsa bus station, and ending in the Tulsa hospital, to determine the need for additional means of transportation (query q_5).

These event queries tend to contain similar or sometimes even identical sub-patterns. Hence, techniques that exploit their similarities for optimization can save computational resources and improve the system responsiveness. Many event-stream-based applications from online advertising, click-stream analytics, social network services to financial fraud detection all feature these huge workloads composed of similar event queries. Thus, performance gains due to leveraging such customized event-optimization technology for shared computations among such event queries could have a tremendous benefit across this wide range of applications.

Challenges. To design an effective event-analytics infrastructure, we must tackle the following challenges.

Rich Application Semantics. Streaming applications have rich semantics. This semantics involves event-sequence construction of arbitrary length; event conjunction, disjunction and negation; expressive predicates; time-, count- or predicate-based windows; event-pattern grouping and aggregation. Therefore, we must develop efficient processing techniques for a large workload of such expressive event-pattern queries.

Real-time System Responsiveness. We target time-critical applications in which milliseconds can make a difference in decision making. Thus, event query computations should be shared or even completely eliminated if we can do so without compromising result quality. These computational savings speed up the decision-making process, improve resource allocation, reduce environmental pollution and even save human lives. However, sharing is not always beneficial. Even if two event queries syntactically share a sub-pattern, the actual sets of matches of these queries may not overlap at runtime [43]. Sharing computations across such event queries may result in negligible performance gain at the cost of adding significant synchronization overhead. Fortunately, while the number of identical sub-patterns in a query workload at times may be limited, other hierarchical relationships among event queries can be exploited for optimization [36].

Correct Event-Stream Execution. Sharing common or similar sub-patterns between several event queries makes these queries interdependent. Indeed, the shared sub-pattern must be computed before the queries that share it. An efficient runtime execution infrastructure should process a workload of such interdependent event queries while leveraging the concurrent execution capabilities of modern multi-core machines. Thus, a concurrency control mechanism is needed that ensures correct concurrent stream processing. Furthermore, if we strive to delay or even skip event-sequence construction while computing event-sequence aggregations, we must assure that no potential event sequence matches are missed under the premise that aggregation is computed on-the-fly and events are instantly pruned upon their aggregation.

State of the Art. Multi-query optimization is an established technology in relational databases [8, 11, 21]. Unfortunately, these techniques cannot be applied directly to shared event-query processing because streaming data is continuously under flux. Thus, the data-driven approach of event processing may trigger the pattern matching process to be spawned in diverse orders based on the arrival of events. The nature of continuous event-stream-processing systems stands in contrast to the traditional static processing frameworks where all data is given a priori and execution can be fully orchestrated.

Many complex-event-processing systems do not exploit sharing opportunities across the event-query workload [7, 13, 35, 50]. While XML-filtering approaches leverage some sharing opportunities, such as shared prefix-matching, they disregard other sharing opportunities [14, 15]. While the approaches proposed in [2, 47] share sub-patterns in the distributed context, they do not provide any guarantee to produce a globally optimal plan for multiple event queries. Several approaches [12, 38] are devoted to the optimization of multiple event queries. However, these approaches neglect inter-query event correlations and thus may miss optimization opportunities. Existing solutions to processing multiple concurrent event queries over different abstraction levels, online event pattern aggregation, and general stream transaction models are either missing or limited by having assumptions that do not hold in our event context.

Key Innovations. In this article, we present an overview of four orthogonal innovations for the optimization of complex event analytics developed by members at WPI and collaborators. Each of these innovations leverages shared processing opportunities unique to event analytics. These innovations include:

1) *Event-Sequence Pattern Sharing.* We analyze the benefit of sharing event-sequence construction considering both intra- and inter-event pattern correlations over time [43]. We show that the problem of optimizing a workload of event-sequence patterns to minimize its CPU processing time is equivalent to the NP-hard Minimum Substring Cover problem [28]. This result then leads us to apply the polynomial-time approximate Local-Ratio algorithm to our problem with proven acceptable bounds on optimality [28].

2) *Hierarchical-Event Pattern Sharing.* Event queries, even if not identical, can still be related to each other in terms of both *concept abstractions* and *pattern refinements*. These relations open up unique opportunities for shared processing of similar event-sequence patterns. This pattern similarity leads us to establish the E-Cube hierarchy composed of event queries at different levels of abstraction [36]. Our efficient processing strategies evaluate all event patterns in the workload in a specific order to reuse their intermediate results.

3) *Shared Event-Pattern Aggregation.* Since all event sequences are discarded once their aggregation is computed, we aggregate event-sequences without constructing them. We achieve such on-the-fly event-sequence aggregation by dynamically maintaining a prefix counter and instantly discarding events after their aggregation. Thus, we reduce the event-sequence aggregation costs from polynomial to linear [42]. This optimization technique is exploited while sharing the aggregation of common sub-patterns in the query workload.

4) *Stream Transaction Model.* Given concurrent accesses and updates to shared event pattern matches, we avoid race conditions by designing an appropriate concurrency-control mechanism. To this end, we introduce our stream transaction model [49]. Since the classical Strict-Two-Phase-Locking algorithm incurs a large synchronization delay due to its rigorous order preservation, we introduce event-centric scheduling methods for real-time streaming applications to maximize concurrent execution.

Our thorough experimental studies using both synthetic and real data sets reveal that these optimization techniques achieve several orders of magnitude performance gain compared to state-of-the-art solutions [36, 42, 43, 49]. Furthermore, our technology was tested out successfully in a real-world setting. In particular, we installed our complex event analytics software in the intensive care units at UMASS Memorial Hospital under leadership of Dr. Ellison, head of infection control at UMASS. We analyzed the results of a clinical evaluation of this technology for improving health-care hygiene [16, 17, 49].

Outline. This article is organized as follows. We start with our event-analytics model in Section 2. Afterwards, we present our sharing techniques for sequence patterns in Section 3 and abstraction patterns in Section 4. Section 5 is devoted to the shared processing of aggregations over event patterns. We propose our stream transaction model in Section 6. Related work is discussed in Section 7, while Section 6 concludes this article.

2 Event-Analytics Model

Event Data Model. *Time* is represented by a linearly ordered set of time points $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{Q}^+$ the non-negative rational numbers. An *event* is a message indicating that something of interest happened in the real world. An event e has an *occurrence time* $e.time \in \mathbb{T}$ assigned by the event source. Each event e belongs to a particular *event type* E, denoted $e.type = E$. An event type E is described by a *schema* that specifies the set of *event attributes* and the domains of their values. Events are sent by event producers (e.g., RFID tag readers) to event consumers (e.g., an emergency management system) on *event streams*.

Event Pattern Query. Event queries in our event-analytics model consist of clauses similar to other event query languages, for example, SASE+ [1, 50]. These clauses are the following:

Window (`WITHIN` clause) specifies the portion of the potentially unbounded input event stream to be processed by one event-query invocation. Our language supports both fixed-length time or count-based tumbling or sliding windows [3, 33] and variable-length predicate-based windows [19].

Pattern (`PATTERN` clause) defines the structure of event occurrences in the input event stream that must match in order for a complex event to be detected [36, 50]. Let E be an event type, P and P' be event patterns. Then, an event pattern is defined by a composition of operators including event occurrence of type E, event-pattern non-occurrence $!P$, event-pattern conjunction $AND(P, P')$ and disjunction $OR(P, P')$, event sequence of fixed length $SEQ(P, P')$, and event pattern of arbitrary length $P+$.

Predicates (`WHERE` clause) impose additional constraints on event-pattern matches. These constraints are boolean expressions composed of arithmetical and comparison operators on event attribute values and constants.

Grouping and Aggregation (`GROUPBY` and `AGG` clauses) can be applied to event pattern matches. Event pattern matches are grouped, for instance, by the attribute values of matched events. Our language supports common aggregation functions such as *count, sum, avg, min* and *max*.

For example, query q_1 in Figure 1 counts the number of people (`AGG Count`) who fled from Texas to Oklahoma (`PATTERN SEQ(TX, OK) WHERE TX.person_id = OK.person_id`) within 48 hours (`WITHIN 48 h`) per age group (`GROUPBY age-group`). Other event queries in Figure 1 behave similarly.

3 Event-Sequence Pattern Sharing

Event Correlations. We target the efficient detection of event-sequence patterns in data streams via shared concurrent pattern execution [43]. Our solution takes as input a set of pattern queries. It estimates the benefit of sharing the computation of sub-patterns based on the time-ordering across events and the inter-query event correlation hidden in the event streams. Sharing an event sub-pattern between multiple queries is not always beneficial. It may even cause more harm than good by incurring unnecessary concurrency-control overhead. Based on this observation, we design a lightweight yet effective method for estimating the *time-sensitive co-occurrence properties* of event streams to accurately capture the benefit of sharing event patterns. The proposed method takes the following two types of event correlations into consideration: (1) *Intra-query event correlation* estimates the number of event sub-pattern matches per time interval, e.g., the percentage of events of type A that follow an event of type B. This ratio estimates the number of matches produced by a single event pattern. (2) *Inter-query event correlation* estimates the sharing potential across multiple event patterns as the ratio of the number of shared sub-pattern matches to the total number of matches.

Benefit of Event Pattern Sharing. We analyze the degree of sharing of sub-pattern matches in a *sample time period* by tracking the number of matches for a sub-pattern within this time period. This process is periodically repeated to provide the up-to-date statistics. Figure 2 shows that the number of matches of a sub-pattern $SP = SEQ(A, B)$ produced by the two patterns P_1 and P_2 may vary over time. Consequently, the number of

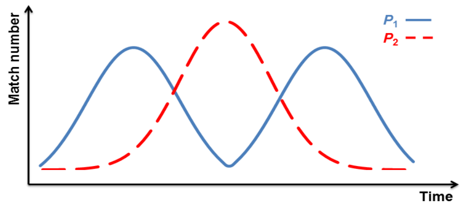


Figure 2: Distribution of event pattern matches over time [43]

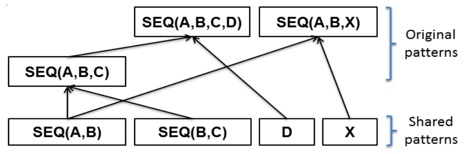


Figure 3: Shared plan of event-sequence patterns [43]

pattern matches for SP that can be shared across P_1 and P_2 also varies over time. This observation leads us to two insights essential for the sub-pattern sharing task: (1) The crests and troughs of P_1 and P_2 never align in this example, even though their average cardinalities over time happen to be similar. Hence, the inter-query correlation between P_1 and P_2 is low. Thus, sharing this sub-pattern between P_1 and P_2 may cause more harm than good due to concurrency control overhead. (2) Even if the cardinalities of the sub-pattern matches happen to be the same for two patterns over time, the match re-use is still not guaranteed since the sub-patterns may not be common for these patterns *at the event-instance level*. Indeed, the benefit of sub-pattern sharing depends on the occurrences of the other sub-patterns in these patterns. In short, cardinality alone is no reliable indicator since individual matches may be non-overlapping. Based on these observations, we design a cost model that accurately estimates the ratio of the cost to compute matches of a *shared* sub-pattern SP for all its parent patterns to the cost of producing all matches of the sub-pattern SP for each parent pattern *separately* as the *redundancy-ratio score*. The lower the score, the higher the benefit of sharing this event sub-pattern.

Shared Event Pattern Plan. Leveraging this redundancy ratio scoring model, we can now tackle the problem of *sub-pattern sharing optimization*. Namely, we aim to find a subset of sub-patterns such that all queries in the given workload share the processing of this subset and the redundancy ratio of this subset is minimal compared to all other possible subsets. We can show that this problem is equivalent to the Minimum Substring Cover problem [43]. Thus, our optimizer can leverage the polynomial-time approximate Local-Ratio algorithm for the Minimum Substring Cover problem to produce the set of sub-patterns to share [28]. Once the set of event sub-patterns is selected, our optimizer iteratively builds up a shared-pattern plan for the workload in a bottom-up fashion. This shared-pattern plan is a graph in which each node is a (sub-)pattern. For example, the original patterns $SEQ(A, B, C)$, $SEQ(A, B, C, D)$, and $SEQ(A, B, X)$ are decomposed into the shared sub-patterns $SEQ(A, B)$, $SEQ(B, C)$, D and X (Figure 3).

4 Hierarchical Event Pattern Sharing

Event-Sequence-Pattern Abstraction Hierarchy. As motivated in Section 1, the number of event-sequence patterns that have syntactically identical sub-patterns (as assumed in Section 3) may be limited. Thus, we now explore effective sharing strategies that also consider hierarchical event queries. This hierarchy is essential for performance optimization in multi-query evaluation since it provides a blueprint for shared online event-query matching. We differentiate between the *concept* and the *pattern hierarchy* [23, 36].

A *concept hierarchy* (Figure 4) is used to summarize information at different levels of abstraction. Many dimensions (e.g., time, location, object type) are hierarchical in nature and thus create a concept hierarchy of

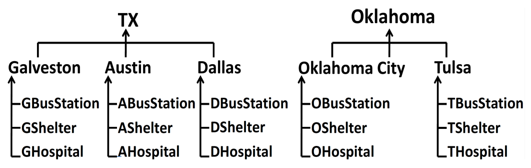


Figure 4: Concept hierarchy of primitive event types [36]

the corresponding event types. Event concept hierarchies for primitive event types are predefined by system administrators based on domain knowledge. An event concept hierarchy is a tree with the most-specific event types as leafs and more-general event types as inner nodes. An event type E_k that is a descendant of an event type E_j is at a finer level of abstraction than E_j, denoted by $E_k <_c E_j$. The non-existence (existence) of a negative (positive) event type at a coarser (finer) concept level enforces more constraints as compared to a negative (positive) event type at a finer (coarser) concept level. In Figure 1, the query q_1 is at a coarser concept level than the query q_2 because TX $>_c$ D and OK $>_c$ T. The query q_4 is at a coarser concept level than the query q_7 since the negative type D in q_4 is coarser than DBusStation in q_7 (D $>_c$ DBusStation).

A *pattern hierarchy* is defined as follows: A query q_k can be drilled-down to a finer-level query q_j by inserting additional event types into the pattern of q_k, denoted by $q_k >_p q_j$. For example, q_6 is at a finer level than q_3 because q_3 enforces the existence of less event types and sequential event relationships than q_6 (Figure 1).

An *E-Cube hierarchy* is a directed acyclic graph where each node is a query q_i and each edge corresponds to a *pairwise refinement relationship* between two queries in terms of either concept or pattern refinement. Each directed edge (q_i, q_j) is labeled with either the label “concept” if $q_i <_c q_j$, “pattern” if $q_i <_p q_j$, or both to indicate the refinement relationship between the queries [25]. Figure 1 shows an example E-Cube hierarchy.

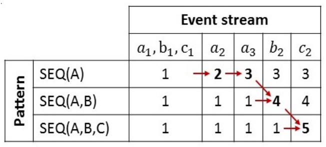
Advanced Event Analytics via Event Pattern Exploration. We now illustrate that a concept or a pattern can be drilled-down into or rolled-up such that we can navigate from one node (with its respective matches) to another node in the E-Cube hierarchy by skipping, adding or replacing sub-patterns. For example in Figure 1, we apply a pattern-drill-down operation on $q_3 = SEQ(G, A, T)$ by adding a $!D$ constraint and get $q_7 = SEQ(G, !D, A, T)$. Similarly, we apply a concept-roll-up operation on $q_2 = SEQ(D, T)$ by one level from Dallas to Texas and from Tulsa to Oklahoma and get $q_1 = SEQ(TX, OK)$.

Optimal E-Cube Evaluation. This E-Cube hierarchy represents the sharing plan for all hierarchical event-pattern queries. For each query q in the E-Cube hierarchy, we have a choice between: (1) Computing q independently from other queries, (2) Conditionally computing q from one of its ancestors or (3) Conditionally computing q from one of its descendants. Our cost model [24, 36] estimates the cost of each option and assigns this cost as a weight on each corresponding directed edge between a pair of queries. Having this directed weighted graph, our goal is to determine an optimal query-evaluation plan ordering, i.e., an ordering of sub-patterns with minimal total execution costs. We show that we can reduce this problem to the Minimal Spanning Tree problem. This reduction allows us to apply the Gabow algorithm [18] to achieve our goal.

5 Shared Event Pattern Aggregation

Online Event Pattern Aggregation. The computation of aggregation over event sequences such as in Figure 1 in our motivating example opens unique opportunities as we illustrate next. We compute an *event sequence count* without ever constructing the actual event sequences. Such online event sequence count can be computed correctly by continuously updating a *prefix counter* in *constant time* upon the arrival of each new event such that a new event, once processed, can be discarded instantly [42].

For example, event sequences matched by the pattern $SEQ(A, B, C)$ are counted in Figure 5. When the events shown on top arrive, the prefix counter for the patterns shown on the left are updated as follows. When the event b_2 arrives, 3 new sub-sequences (a_1, b_2), (a_2, b_2) and (a_3, b_2) are formed using previously arrived events



Pattern	a_1, b_1, c_1	a_2	a_3	b_2	c_2
SEQ(A)	1	**2**	**3**	3	3
SEQ(A,B)	1	1	1	**4**	4
SEQ(A,B,C)	1	1	1	1	**5**

Figure 5: Prefix counters

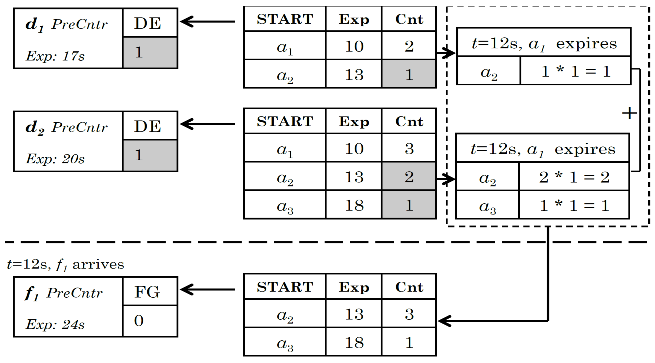


Figure 6: Prefix counters for snapshots [42]

a_1, a_2 and a_3. Thus, the total count of event sequences matched by the pattern $SEQ(A, B)$ is now 4, including the 3 newly formed sequences and a_1, b_1 that we had found before. We observe that when b_2 arrives, we can obtain the count of the event sequences by adding two counts: (1) The count of the sub-pattern $SEQ(A)$ where b_2 would be appended to the matches of this sub-pattern, and (2) The count of the sub-pattern $SEQ(A, B)$. We notice that the actual event sequences do not have to be constructed to update the count. Omitting event sequence construction reduces the aggregation computation costs from polynomial to linear [42].

Other aggregation functions can be supported analogously. For example for *sum*, we maintain an extra *sum* field in each prefix counter on the event type the attribute value of which is to be summed. When an event arrives and causes an update of a count, its respective *sum* field will also be updated.

Negation. Negation requires the non-occurrence of events of the negated event types at certain positions within the event sequences. The arrival of such events can invalidate potential matches. Therefore, when an event arrives whose occurrence is negated in the query, we simply reset the corresponding prefix counter.

Predicates. Local predicates impose constraints on the attribute values of events, for example, *age>20*. Such predicates can filter events before they are aggregated. Equivalence predicates correlate events in a sequence [50]. For example, to monitor people's movement during an emergency, we require the same value of the person identifier attribute in all events contributing to one event sequence matched by the queries in Figure 1. Such predicates partition the event stream into several sub-streams. This partitioning then allows us to compute aggregation separately for each partition using the above described principles.

Sliding Window. When the window slides, multiple events expire and multiple new events become relevant. One expired event might invalidate an arbitrary number of event sequences and thus require an update of the aggregation results. However, the expiration of most events has no affect on the aggregated value. We determine the minimum subset of events whose expiration could indeed affect the aggregation result in [42].

Aggregation Sharing. Shared aggregation of single events is well-studied [30, 34, 51]. However, shared aggregation of event sequences poses new challenges such as pushing the aggregation through the sequence-construction process to save the resources. We could consider the sharing of common sub-sequences between multiple similar queries (Section 3). To minimize the CPU costs, event queries that have common sub-patterns are chopped into sub-patterns to aggregate them separately using the highly scalable techniques introduced above. We then stitch these partial results together to get the final results for the original pattern requests.

S2PL		Requested	
		Read	Write
Held	Read		X
	Write	X	X

LWM		Requested	
		Read	Write
Held	Read		
	Write	may be	X

Figure 7: Lock incompatibility [49]

However, events expire over time. Let $\#s_1$ and $\#s_2$ be the counts of the sub-patterns s_1 and s_2 respectively. When a triggering event of s_2 arrives, $\#s_1$ might become invalid due to the expiration of some of the matches aggregated by s_1. This situation risks causing erroneous aggregation results. To support event expiration, we maintain snapshots for each sub-pattern. The idea is the following: For each first event in a sequence, we store the expiration time point and the number of sequences that start with this event in the snapshot. When a first event expires, we ignore its respective count, since all the sequences it participates in expire too. For example, assume the pattern $SEQ(A, B, C, D, E, F, G)$ is chopped into 3 sub-patterns $s_1 = SEQ(A, B, C)$, $s_2 = SEQ(D, E)$, and $s_3 = SEQ(F, G)$. When the event f_1 arrives at time $t = 12s$, we consider only non-expired counts (they are highlighted in Figure 6). First, we multiply the count of each match of the sub-pattern $SEQ(D, E)$ with the counts in its respective snapshot of the sub-pattern $SEQ(A, B, C)$. Second, we sum up the counts for the same first event across all matches (a_2 in our example). Third, we store the resulting counts in the snapshot of the sub-pattern $SEQ(A, B, C, D, E)$ for future reference.

6 Facilitating Concurrency for Efficient Complex Event Analytics

Stream Transactions. In prior sections we have illustrated various strategies to detect shared sub-patterns and then to reuse their partial results. To achieve high system responsiveness, we leverage modern multi-threaded solutions on multi-core architectures instead of forcing all computation to proceed sequentially. Thus, to avoid race conditions, read and write operations on shared storage (e.g., results of a shared sub-pattern) must be synchronized. Traditional transaction models should be reexamined since: (1) Events are not static, rather they continuously arrive on streams. (2) Event queries are standing, they continuously monitor these event streams [6]. (3) Neither abort nor restart of a transaction at a later time point (used in MVCC [5]) may be acceptable for externally visible output or actions typical for real-time streaming applications [45, 49].

Towards Event-Stream Transactions. Here we briefly introduce an appropriate notion of transactions in the context of event streams, which we henceforth refer to as stream transaction. A *stream transaction* is a sequence of all system changes that are triggered by a single input event. Two operations are called *conflicting* if they are performed on the same data item and at least one of them is a Write. An algorithm for scheduling operations on a shared data item performed by event queries is then considered to be *correct* if every schedule produced by the algorithm processes conflicting operations in order by their application time stamps.

Let us now examine one simple transaction model in this context. Similarly to the classical MVCC [5], the historic records of each shared data item could be maintained. We then define the *low-water-mark* as the oldest time stamp among all the time stamps of Write locks. A Read lock is granted if all Writes earlier than the Read have completed. A Write lock is granted if it is the oldest Write lock among all Write locks on this data item. Given this lock-granting strategy, we can relax the lock incompatibility in two ways (Figure 7): (1) A Read lock does not block acquiring a Write lock since the previous version is read while a new one is created. (2) A Write lock does not necessarily block a Read lock if earlier versions can be read. This modification allows for faster responsiveness compared to the sequential Strickt Two Phase Locking (S2PL) [49].

This stream transaction model is generic since it is applicable to the sharing techniques described above. However, a customized transaction model that considers the semantics of the view-maintenance operations on a shared common view might be more efficient. Ray et al. [43] introduce a customized stream transaction model for shared views. This model defines the compatibility of read, append, and purge operations on shared views. It then uses S2PL to schedule transactions composed of such operations. Future work could focus on developing concurrent processing models that best support each of the different shared analytics optimization scenarios.

7 Related Work

Complex Event Query Processing. Existing event processing systems focus on the specification and optimization of automaton-based [1, 13, 50] and query-plan-based [40] execution paradigms. Liu et al. [35] consider nested event patterns and introduce a top-down iterative approach for processing such queries. However, these approaches neither address the issue of supporting queries at different concept and pattern hierarchy levels nor do they develop efficient computation strategies for the shared execution of multiple event queries.

Sharing Multiple Event Queries. Shared event query processing is part of the native architecture of TelegraphCQ [9]. Madden et al. [38] proposed an adaptive tuple-level sharing technique. However, routing individual tuples among operators introduces considerable overhead. Instead, our approach produces a stable sharing plan and re-optimizes only if there is significant change in statistics [31].

Hong et al. [29] introduce materialization-based optimization techniques into XML-stream query processing. This approach does not consider windows, event correlations, view maintenance and concurrent query access to these views. YFilter [14] is limited to prefix-matching. In contrast, our technique shares sub-patterns at arbitrary positions. Ray et al. [44] propose continuous sliding-view maintenance over event streams for a single query. Sharing such views among multiple queries is not considered.

Hierarchical Event-Query Sharing. Traditional OLAP technologies focus on static pre-computed and indexed data sets. They aim to quickly provide answers to analytical queries that are multi-dimensional in nature [10, 22, 27]. OLAP techniques allow users to navigate the data at different abstraction levels. However, these solutions either do not support real-time streams [20, 26, 37], or they are set-based instead of sequence-based [22]. Furthermore, these approaches do not support concept hierarchies. They provide neither result reuse strategies nor any cost analysis for patterns expressing event sequence and negation.

Shared Event Query Aggregation. The optimization of CEP aggregation is critical for high performance pattern matching over event streams [1, 13, 40, 50]. However, no specific technique has been proposed to date to optimize the on-the-fly computations of event-sequence aggregation. Instead, existing approaches apply aggregation as a *post-processing step* that takes place after all event sequences have been constructed. Obviously, this is an inefficient solution. Incremental techniques [30, 34] have been proposed to avoid re-computations among overlapping sliding windows. Zhang et al. [51] maintain aggregates using multiple levels of temporal granularity: older data is aggregated using coarser granularity while more recent data is aggregated with fine detail. However, these approaches do not address our sequence aggregation problem, that is, they compute aggregation over individual events rather than over event sequences that are continuously detected in real time.

Aggregation is well-supported in static sequence databases [32, 37]. These approaches assume that the data is statically stored and indexed prior to processing. In contrast, our approach targets dynamic streaming data where results are produced continuously upon event arrival and events are discarded once they are aggregated.

Range-based aggregation approaches [32, 48] aggregate independent data records within a certain time range. Some approaches [41, 46] consider aggregation for patterns with recursion. However, these approaches work with independent individual data records. In contrast to that, our approach aggregates event sequences matched by expressive event patterns, i.e., interdependent multi-record matches.

Stream Transaction Models. Botan et al. [6] adapt the traditional database transaction model to event stream processing. That is, a transaction is a sequence of user-defined operations. Events must be processed in order by their arrival time stamps. Other stream transaction models [4, 39] define a transaction as a sequence of operations triggered by one or more input events. Events are usually batched and their processing is ordered by event time stamps. However, these approaches are too restrictive, since they process events in strict order and disallow concurrent operations on the same data item, unlike our proposed Low-Water-Mark scheduler [49]. This strictly ordered processing strategy slows down execution and results in poorer system responsiveness.

8 Conclusion and Future Work

In this article, we have presented an overview of four innovative techniques for scaling shared event analytics, namely: (1) To effectively share identical sub-patterns, we consider intra- and inter-query correlation, match distribution over time and match sharing at the event instance level. (2) Since the number of identical sub-patterns in an event query workload may be limited, we also share computations among hierarchical event queries. (3) While computing event sequence aggregation, we do not construct the actual event sequences and thus reduce the computation costs from polynomial to linear. Multiple aggregation event queries share the aggregation computation of their common sub-patterns. (4) Our stream transaction model guarantees correct concurrent execution of multiple inter-dependent event queries sharing their intermediate results.

In the future, we will extend our online shared aggregation approach to a broader class of event queries. For time-critical decision making applications, certain urgent insights are useful only if derived within a strict time constraint. Thus, we will define different consistency levels and propose prioritized scheduling algorithms to ensure prompt responsiveness using limited resources. Furthermore, these techniques have been proposed in the context of a central albeit possibly multi-threaded architecture. The next logical step would be to explore their effectiveness in context of deploying complex event analytics on an distributed computing platform.

Acknowledgements

Section 4 is the result of a successful collaboration with the researchers from HP Labs, in particular, Chetan Gupta, Song Wang and Abhay Mehta. The authors also thank Kara Greenfield and Ismail Ari for productive collaboration. For Section 6, the authors collaborated with UMass Medical School, in particular, Richard T. Ellison III. We thank Dr. Ellison for his leadership of installation of our event analytics system in the intensive care units at UMASS Memorial Hospital. This work was supported by the following grants: NSF IIS-III-1018443, NSF IIS 0917017, NSF CRI (Equipment Grant), HP Lab Innovation Research Grant, UMMS-WPI CCTS Collaborative Grant, and Turkish National Science Foundation TUBITAK under career award 109E194.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160. ACM, 2008.

[2] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *In VLDB*, 1(1):66–77, 2008.

[3] A. Arasu, and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[5] P. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., 1997.

[6] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul. Transactional stream processing. In *EDBT*, pages 204–215, 2012.

[7] B. Cadonna, J. Gamper, and M. H. Bohlen. Sequenced event set pattern matching. In *EDBT*, pages 33–44, 2011.

[8] U. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *VLDB*, pages 384–391, 1986.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 668–680, 2003.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *In SIGMOD*, 26(1):65–74, 1997.

[11] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190 – 200, 6-10 Mar 1995.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[13] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[14] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, 2003.

[15] Y. Diao, P. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, pages 341–342, 2002.

[16] R. Ellison, D. W. Constance M. Barysauskas, Elke A. Rundensteiner, and B. Barton. A prospective controlled trial of an electronic hand hygiene reminder system. In *IDWeek Conference, Advancing Science Improving Care*, 2013. Abstract 314.

[17] R. Ellison, D. W. Constance M. Barysauskas, Elke A. Rundensteiner, and B. Barton. A prospective controlled trial of an electronic hand hygiene reminder system. *Open Forum Infectious Diseases*, 2(4):1–8, Dec. 2015.

[18] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[19] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid. Exploiting predicate-window semantics over data streams. *In SIGMOD*, 35(1):3–8, Mar. 2006.

[20] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID FlowCubes for multi-dimensional analysis of commodity flows. In *VLDB*, pages 834–845, 2006.

[21] J. Grant and J. Minker. On optimizing the evaluation of a set of expressions. *Int. J. of Computer & Information Sciences*, pages 179–191, 1982.

[22] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[23] C. Gupta, S. Wang, A. Mehta, M. Liu, and E. Rundensteiner. Computing a hierarchical pattern query from another hierarchical pattern query, Apr. 25 2013. Patent US20130103638 A1.

[24] C. Gupta, S. Wang, A. Mehta, M. Liu, and E. Rundensteiner. Determining an execution ordering, Apr. 5 2016. Patent US9305058 B2.

[25] C. Gupta, S. Wang, A. Mehta, M. Liu, E. Rundensteiner, and M. Ray. Nested complex sequence pattern queries over event streams, Mar. 29 2016. Patent US9298773 B2.

[26] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream Cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[27] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[28] D. Hermelin, D. Rawitz, R. Rizzi, and S. Vialette. The minimum substring cover problem. In *Int. Conf. on Approximation and Online Algorithms*, pages 170–183, 2008.

[29] M. Hong, A. J. Demers, J. E. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, pages 761–772, 2007.

[30] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[31] C. Lei and E. A. Rundensteiner. Robust distributed stream processing. In *ICDE*, pages 817–828, 2013.

[32] A. Lerner and D. Shasha. AQuery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.

[33] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[34] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates

in data streams. In *SIGMOD*, pages 311–322, 2005.

[35] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. High-performance nested CEP query processing over event streams. In *ICDE*, pages 123 – 134, April, 2011.

[36] M. Liu, E. A. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.

[37] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. OLAP on sequence data. In *SIGMOD*, pages 649–660, 2008.

[38] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, 2002.

[39] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: Streaming meets transaction processing. *In VLDB*, 8(13):2134–2145, 2015.

[40] Y. Mei and S. Madden. ZStream: A Cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

[41] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *SIGMOD*, pages 440–451, 1997.

[42] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.

[43] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, 2016. (To appear).

[44] M. Ray, E. A. Rundensteiner, M. Liu, C. Gupta, S. Wang, and I. Ari. High-performance complex event processing using continuous sliding views. In *EDBT*, pages 525–536, 2013.

[45] E. Rundensteiner, D. Wang, and R. Ellison. Active complex event processing or infection control and hygiene monitoring, Oct. 6 2011. US Patent App. 13/077,401.

[46] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, June 2004.

[47] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, pages 4:1–4:12, 2009.

[48] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

[49] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *In VLDB*, 4(10):634–645, July 2011.

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

[51] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal aggregation over data streams using multiple granularities. In *EDBT*, pages 646–663, 2002.