Extending Complex Event Processing for Advanced Applications

by

Di Wang

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
in
Computer Science
by

April 29, 2013

APPROVED:

Prof. Elke A. Rundensteiner
Advisor

Dr. Badrish Chandramouli
Microsoft Research Redmond
External Committee Member

Prof. Daniel J. Dougherty
Committee Member

Prof. Craig E. Wills
Head of Department

Prof. Mohamed Y. Eltabakh
Committee Member

Abstract

Recently numerous emerging applications, ranging from on-line financial transactions, RFID based supply chain management, traffic monitoring to real-time object monitoring, generate high-volume event streams. To meet the needs of processing event data streams in real-time, Complex Event Processing technology (CEP) has been developed with the focus on detecting occurrences of particular composite patterns of events. By analyzing and constructing several real-world CEP applications, we found that CEP needs to be extended with advanced services beyond detecting pattern queries. We summarize these emerging needs in three orthogonal directions. First, for applications which require access to both streaming and stored data, we need to provide a clear semantics and efficient schedulers in the face of concurrent access and failures. Second, when a CEP system is deployed in a sensitive environment such as health care, we wish to mitigate possible privacy leaks. Third, when input events do not carry the identification of the object being monitored, we need to infer the probabilistic identification of events before feed them to a CEP engine. Therefore this dissertation discusses the construction of a framework for extending CEP to support these critical services.

First, existing CEP technology is limited in its capability of reacting to opportunities and risks detected by pattern queries. We propose to tackle this unsolved problem by embedding active rule support within the CEP engine. The main challenge is to handle interactions between queries and reactions to queries in the high-volume stream execution. We hence introduce a novel stream-oriented transactional model along with a family of stream transaction scheduling algorithms that ensure the correctness of concurrent stream execution. And then we demonstrate the proposed technology by applying it to a

real-world healthcare system and evaluate the stream transaction scheduling algorithms extensively using real-world workload.

Second, we are the first to study the privacy implications of CEP systems. Specifically we consider how to suppress events on a stream to reduce the disclosure of sensitive patterns, while ensuring that nonsensitive patterns continue to be reported by the CEP engine. We formally define the problem of utility-maximizing event suppression for privacy preservation. We then design a suite of real-time solutions that eliminate private pattern matches while maximizing the overall utility. Our first solution optimally solves the problem at the event-type level. The second solution, at event-instance level, further optimizes the event-type level solution by exploiting runtime event distributions using advanced pattern match cardinality estimation techniques. Our experimental evaluation over both real-world and synthetic event streams shows that our algorithms are effective in maximizing utility yet still efficient enough to offer near real time system responsiveness.

Third, we observe that in many real-world object monitoring applications where the CEP technology is adopted, not all sensed events carry the identification of the object whose action they report on, so called "non-ID-ed" events. Such non-ID-ed events prevent us from performing object-based analytics, such as tracking, alerting and pattern matching. We propose a probabilistic inference framework to tackle this problem by inferring the missing object identification associated with an event. Specifically, as a foundation we design a time-varying graphic model to capture correspondences between sensed events and objects. Upon this model, we elaborate how to adapt the state-of-the-art Forward-backward inference algorithm to continuously infer probabilistic identifications for non-ID-ed events. More important, we propose a suite of strategies for optimizing the performance of inference. Our experimental results, using large-volume streams of a real-world health care application, demonstrate the accuracy, efficiency, and scalability of the proposed technology.

Acknowledgment

I express my sincere thanks to my advisor Prof. Elke Rundensteiner for all the guidance, support and encouragement she has given me over the years. I feel extremely lucky to have a wonderful advisor who not only teaches me every piece of knowledge but also the approaches to acquire knowledge by myself. Working with her significantly contributed to my passion for database systems research. Moreover, she always acts like a close friend to me, willing to listen to me and providing help when I most need it. She has been a great role model to me.

My thanks goes to the members of my Ph.D committee, Prof. Dan Dougherty, Prof. Mohamed Eltabakh and Dr. Badrish Chandramouli for the many valuable discussions and suggestions that helped me understand my research areas better.

I would like to express my gratitude to Dr. Richard Ellison III from Univ. of Massachusetts Medical School for providing the invaluable resources and continuous support in the collaborative HyReminder project.

I would like to thank Dr. Yeye He and Prof. Jeff Naughton from Univ. Wisconsin Madison. In our collaboration on the data privacy related topics, they exposed me to many interesting aspects of database research and inspired me to do my very best.

Also, I sincerely appreciate the education and encouragement from Prof. Murali Mani in my Master's study. I also would like to thank Ming Li, Mo Liu, Han Wang, Lei Cao and Song Wang, with whom I have interacted during the research on the event stream processing. My thanks also goes to Abhishek Mukherji, Mingzhu Wei, Di Yang, Rimma Nehme, Xika Lin and all other previous and current DSRG members for their useful discussions and feedback.

Finally, I would like to thank my husband Yeye for his never-ending love and support. He made me the person that I am today. I dedicate this thesis to him.

Contents

1 Introduction **1**

1.1 Complex Event Processing 2

1.2 Motivating Application 3

1.3 State-of-the-Art 6

1.4 Research Challenges 9

1.5 Contributions of the Dissertation 10

1.6 Road Map 14

2 Preliminary **15**

2.1 Complex Event Model 16

2.2 Complex Pattern Query Language 17

3 Active Complex Event Processing and Stream Transaction **20**

3.1 Introduction of Active CEP 21

3.2 Stream Concurrency Problem: Examples 25

3.3 The Active Complex Event Model 27

3.3.1 Active Rule Specification 30

3.3.2 Alternate Models 32

3.4 Stream Transactions 34

3.4.1 Notion of Correctness 34

3.4.2 Stream Transaction Model 35
3.4.3 Stream-ACID properties 36
3.5 Stream Transaction Scheduling 37
3.5.1 Single-Event-Initiated Scheduler 38
3.5.2 Strict 2PL Scheduler 39
3.5.3 Low-Water-Mark Scheduler 40
3.6 Experimental Evaluation for ACEP 45
3.6.1 Experimental Settings 45
3.6.2 Experimental Results 47
3.7 ACEP Infrastructure 51
3.7.1 Architectural Considerations 51
3.7.2 The ACEP Execution Plan 53
3.7.3 Optimization Techniques 55
3.8 Recovery of Transactional Streams 57
3.8.1 Basic Stream Data Backup 58
3.8.2 Recovery for Stream-Transaction 59
3.9 Related Work 61

4 Privacy-Preserving Complex Event Processing via Event Suppression 63

4.1 Introduction of Privacy-Preserving CEP 64
4.2 The Complication: Private Patterns 68
4.2.1 Suppressing Private Query Patterns 69
4.3 Problem Statement and Variants 70
4.3.1 Hard-constraint vs. Soft-constraint 73
4.3.2 Type-level vs. Instance-level 74
4.3.3 Offline vs. Online 74

4.4 Hardness Results . . . 75
4.5 Online Suppression Algorithms . . . 78
4.5.1 Optimal Type-Level Algorithm . . . 79
4.5.2 Hybrid Instance-Level Solution . . . 83
4.6 Pattern Match Cardinality Estimation . . . 88
4.6.1 Estimation by Arrival Rate . . . 90
4.6.2 Estimation by Periodicity . . . 91
4.7 Experimental Evaluation . . . 95
4.7.1 Experimental Setup . . . 95
4.7.2 Algorithms Compared . . . 96
4.7.3 Experiments on Hospital Workload . . . 97
4.7.4 Experiments on Synthetic Workload . . . 101
4.8 Related Work . . . 102
4.9 Conclusion . . . 104

5 Probabilistic Inference of Uncertain Identities over Event Streams 105

5.1 Introduction of FISS . . . 106
5.2 Problem Statement . . . 109
5.3 Proposed Graphical Model . . . 112
5.3.1 Components of the Model . . . 113
5.3.2 Dependencies for Object Association . . . 115
5.4 Inferring IDs Over Streams: Initial Effort . . . 117
5.4.1 Inference using FB Algorithm . . . 118
5.4.2 Discussion of Deficiencies of FB Algorithm . . . 122
5.5 Inference Speedup: Optimization Strategies . . . 123
5.5.1 Pruning Unaffected Variables . . . 123

5.5.2 Early Termination Using Finish-Flags 124
5.5.3 Selective Smoothing via Pattern Matching 127
5.6 Experimental Evaluation 134
5.6.1 Experimental Setup 134
5.6.2 Alternative Approaches Compared 136
5.6.3 Experiments on Inference Accuracy 137
5.6.4 Experiments on Inference Efficiency 138
5.7 Related Work 142
5.8 Conclusion 143

6 Conclusions and Future Work 145
6.1 Summary 145
6.1.1 Interrelationship of Proposed Techniques 146
6.2 Future Research Directions 149
6.2.1 On-line Sensor Anomaly Detection 149
6.2.2 Revisiting Alternative Stream Transaction Models 151

List of Figures

3.1 Representative hand hygiene logic. If the HCW is in the *start-status* node and his behavior matches the *pattern-query* annotating an outgoing edge, then change his status to the *end-status* node. 22

3.2 Stream concurrency examples. Input events are consumed in their *application time*; Read and Write are executed by the system with *system time*. Dashed line represents that the input event triggers a corresponding operation. E.g., $Read_{Q1}(P_k)$ denotes the Read performed by query Q1 on shared table P_k 27

3.3 Lock compatibility (X - incompatibility). 39

3.4 S-transactions and LWM Example. 43

3.5 *readN* vs. throughput (*lockG*=tuple). 47

3.6 *readN* vs. throughput (*lockG*=table). 47

3.7 $writeN$ vs. throughput (*lockG*=tuple). 48

3.8 $writeN$ vs. throughput (*lockG*=table). 48

3.9 Baseline latency. 48

3.10 Overhead of schedulers. 48

3.11 Input rate vs. latency. 48

3.12 Input rate vs. memory. 48

3.13 Active CEP System Architecture 53

3.14 Logical execution plan for Q1 and R1 54
3.15 Backing-up data in a stream system 59

4.1 An adversarial attack using public pattern matches 65
4.2 Problem space with three orthogonal dimensions 71
4.3 Type-level LP-based algorithm 83
4.4 Motivation: suppressing c^{14} can be sub-optimal 85
4.5 Running example of hybrid algorithm 88
4.6 Estimating matches of SEQ(A,B) using periodic pattern 92
4.7 Estimating via periodic pattern for time $2\hat{T}$ 93
4.8 Performance on hospital workload 98
4.9 Performance on synthetic workload 100

5.1 Graphical model for FISS 116
5.2 Forward-Backward inference example 120
5.3 Example of optimized smoothing when *exit(127,*R_1*,*O_1*)* just arrives. *Upper*: partitions of probabilistic events. *Bottom*: random variables with finishing-flags. *Dashed lines*: hyper-links from events to random variables. 127
5.4 Pattern query Q1 specifying affected event 130
5.5 Comparison on Inference Accuracy 138
5.6 Processing Time on Stream of 2000 Events 139
5.7 Smoothing window size vs. throughput 140
5.8 Output strategy vs. num. of output events 141

6.1 Interrelationship of Proposed Techniques 147

List of Tables

3.1 Symbols for performance metric 46

3.2 Parameters for queries and rules. 49

3.3 Architectural considerations 52

4.1 Pattern query characteristics 96

5.1 Summary of symbols 113

Chapter 1

Introduction

1.1 Complex Event Processing

An increasing number of real-world applications require processing of streaming data from different sources with an unpredictable rate to obtain timely responses to complex queries. Examples of such applications come from the most disparate fields: from wireless sensor networks to on-line financial service, from traffic management to click-stream inspection.

These requirements led to the development of Data Stream Management Systems (DSMSs) [15–17, 19], which were designed as an evolution of traditional Database Management Systems (DBMSs). While traditional DBMSs are designed to work on persistent data, where updates are relatively infrequent, DSMSs are specialized in dealing with transient data that is continuously updated. Accordingly, while DBMSs run queries just once to return a complete answer, DSMSs execute continuous queries, a.k.a., long-running queries, which run continuously and provide updated answers as new data arrives.

Recently numerous emerging applications, ranging from on-line financial transactions, RFID based supply chain management, traffic monitoring to sensor networks, generate real-time event streaming data. To meet the special needs of processing event data streams, Complex Event Processing technology (CEP) was developed with the focus on detecting occurrences of particular composite patterns of events, whose occurrence has to be notified to the interested parties in real-time. CEP systems exhibit sophisticated capabilities for pattern matching in huge volume event streams [6, 13, 21, 27, 70, 98]. Indeed, CEP systems put great emphasis on the issue that represents the main limitation of DSMSs: the ability of detecting complex patterns of incoming items, involving sequencing and ordering relationships.

By analyzing and constructing several real-world CEP applications, we found that CEP needs to be extended with advanced services beyond detecting pattern queries. We

summarize these emerging needs in three orthogonal directions. First, for applications which require access to both streaming and stored data, we need to provide a clear semantics and efficient schedulers in the face of concurrent access and failures. Second, when a CEP system is deployed in a sensitive environment such as health care, we wish to mitigate possible privacy leaks. Third, when input events do not carry the identification of the object being monitored, we need to infer the probabilistic identification of events before feed them to a CEP engine. Next we illustrate these needs using a health care application as a representative example.

1.2 Motivating Application

According to US Centers for Disease Control and Prevention, healthcare- associated infections hit 1.7 million people a year in the United States, causing an estimated 99,000 deaths [7]. HyReminder [93] is a real-time hospital infection control system, developed collaboratively between WPI and Univ. of Massachusetts Medical School to continuously track healthcare workers (HCWs) for hygiene compliance (for example cleansing hands, wearing masks and sanitizing as required), and to remind the HCWs at the appropriate moments to perform hygiene precautions - thus preventing the spread of infections in the hospital.

At Univ. of Massachusetts Memorial Hospital, where our HyReminder system is deployed, every HCW wears an electronic badge. The RF readers and Infrared sensors installed over doors, near patient beds and on sanitizers observe each HCW's behaviors. CEP technology is employed to monitor and detect HCW's hygiene performance. For example, according to US hospitals hand hygiene regulations [25], a HCW must sanitize hands after he contacts a patient. Hence, if the CEP engine detects a sequence of HCW's behaviors that the HCW left a patient room (modeled as "exit" event), and did not sani-

tize hands (modeled as "!sanitize" event, where ! represents negation), and then entered another patient room (modeled as "enter" event), then the CEP engine will immediately report such event pattern as a hand hygiene violation.

In order to remind a HCW to perform hygiene precautions upon the detection of a hygiene risk or violation, every HCW's badge has embedded a colorful light to indicate his current hygiene status: "safe" (green), "warning" (yellow) or "violation" (red). For example, if the CEP engine discovers a violation pattern described above, namely the behavior sequence $< exit, !sanitize, enter >$, then we will set the HCW's badge to yellow, as a warning for him to sanitize hands. If the CEP engine later detects that the HCW has been warned but still has not sanitized, at the following time he contacted another patient, we can determine that this behavior pattern is a very dangerous sign of infection spread. We will immediately set the HCW's badge into red, as a serious urge for him to perform hygiene precautions. It is critical for us to support such *real-time reactions*, i.e., updating a HCW's hygiene status, for the pattern queries. Furthermore, the status of each HCW is also used as a condition by pattern queries, e.g., a pattern query may only monitor HCWs in "safe" status. Clearly the reactions for queries that update HCWs' status will *affect query results in turn*. It is absolutely vital to control such *concurrent* updates and accesses so as to assure the correct execution logic. Otherwise we risk for a potentially highly contagious disease to be transmitted to vulnerable patients - thus increasing patient suffering and even causing deaths.

Furthermore, we consider the issue that failures may arise, be it of the static data source from which the CEP system may draw data from, one of the CEP engines in a distributed context, or possibly the machine the stream engine resides, and so on. Therefore it is absolutely vital to build a appropriate *recovery strategy* for CEP that guarantees an efficient recovery yet with a minimum overhead.

Besides correctly detecting HCW's real-time hygiene status, *privacy protection* of

personal health data is another crucial requirement for HyReminder. While it is clear that HyReminder can reveal useful event patterns for hygiene monitoring, it may also disclose event patterns that patients would prefer to keep hidden. For example, an observation that a doctor leaves a patient's room and then immediately enters a psychiatrist's office might serve as an indication that this patient is experiencing psychiatric problems. The issue is that when System-H monitors doctors' behaviors (the intended use of this application), the events being reported may disclose private information about individual patients as a side-effect. We observe that the simplistic approach of not directly reporting private patterns does not prevent their disclosure — the occurrence of several hygiene compliance patterns that HyReminder does report may be used by an adversary to infer the existence of private pattern matches. Therefore, to allow a CEP system to be safely used in a sensitive environment like a hospital, we must take its privacy preservation issues into account.

Now let us consider the input sensor data of HyReminder. These events may come from heterogeneous monitoring devices. While some devices are capable to provide events with object identification, e.g., RF sensors, other devices do not capture object identification, e.g., Passive Infrared sensors. In particular, in the hospital RF taggers are attached to medical equipments to track their distributions, while Infrared sensors are installed in nurse stations and doctor offices to check the sterilization procedure of the equipment. Therefore the input stream is composed of *both ID-ed and non-ID-ed* events. As another example in the hospital environment, it is a regulation to restrict the number of staff in an Operation Room [25] in order to prevent airborne infections to the patient in operation. Infrared sensors are thus installed in Operation Rooms to check on the number of staff over time, and to observe the door openings to get an insight into airflow transmission. Simultaneously, RF sensors are equipped to also monitor the hygiene performance of healthcare workers. Events observed in such an environment are mixed with ID-ed and

non-ID-ed events also. Given such a mixed input stream, those non-ID-ed events prevent us from performing *object-based analytics*, such as object tracing, alerting and pattern matching, which usually is the key service provided by a CEP system. Therefore, we need address the fundamental data transformation problem for event streams mixed with ID-ed and non-ID-ed events, namely to translate raw streams into queriable, *probabilistic event streams* with object identification.

1.3 State-of-the-Art

We briefly discuss the state-of-the-art and its respective shortcomings in meeting the requirements identified in Section 1.2, while detailed related work is presented in Sections 3.9, 4.8, and 5.7.

Transactional Stream Processing. Current CEP engines [21, 70, 98] and DSMS systems [6, 15, 27, 30] support "read-only" query processing. Pattern queries only contain *in-place* conditions, i.e., the query qualification is either based on the attributes of events matched in the input stream [21, 27, 70, 98] or on static information pre-loaded yet not updated during query execution [6, 13, 41]. Moreover, reactions for event detection are limited to "side-effect-free" actions like sending a message or logging events that do not affect the event detection in turn [95]. Supporting concurrent update actions within the continuous execution of pattern queries over streams, especially actions that directly affect the pattern query results themselves, is an open problem.

This raises the critical problem of handling the *real-time mutual effects* between queries and reactions for queries in high-volume stream execution. Most data stream systems [6, 15, 27, 30] and CEP engines [21, 70, 98], which commonly use the *push-based* execution paradigm, have not addressed this challenge. In the push-based stream execution paradigm, new events are evaluated by continuous queries immediately upon event arrival

without any safeguard mechanism to synchronize the concurrent accesses and updates during stream execution. Since continuous queries may vary in their complexities and event consumption, they may exhibit different processing delays. As a result, events with different timestamps tend to co-exist and be simultaneously executed in the system. Using this push-based execution for pattern queries and reactions for queries (represented as active rules) may raise unexpected anomalies. We thus need a *stream transaction model* embraces both relational and streaming data processing. In other words, we wish to support both push-based and pull-based query execution.

Moreover, to date no systematic failure recovery service has been designed for transactional stream processing. In classic relational DBMSs, ARIES [72] is the gold standard for transaction recovery. While the recovery strategies sketched in STREAM [15] and Aurora [33] stream systems do not incorporate the transaction concept. In other words, a recovery strategy that offers efficient services for both relational and streaming data is lacking.

Privacy-Aware Event Stream Processing. To our knowledge, the issues of privacy preservation in CEP systems has not yet been addressed, though the problem of privacy-preserving data publishing has been extensively studied in static databases. Most prominently, there is a huge branch of work focusing on structural privacy, including k-anonymity [86] and many of its successors like l-diversity [68], t-closeness [62], m-invariance [99], and e-equivalence [51], that propose to recode the original database in such that the modified database satisfies various structural properties that provides an intuitive level privacy. Structural privacy – while suitable for relational databases that correspond to a finite set of tuples – does not apply straightforwardly to the CEP model where the data is a constantly arriving, possibly infinite stream of events. Our problem also bears some resemblance with online query auditing [57, 73], where the problem is to determine whether to answer or deny answering queries based on the answers produced

for previous queries, with the goal of disallowing the adversary from inferring private information by composing the answers to queries that are answered. However, those query auditing techniques essentially make a one-time decision for the transient query. While in CEP, we need to make continuous decisions for long-running queries. In CEP, it is very likely that matches of a particular pattern query could not be reported at this time but could be safely reported at other times. Hence the query auditing techniques are not directly applicable to our problem.

Probabilistic Inference over Event Stream. Recent research on RFID data cleaning and inference [28, 32, 54, 56, 89] assumes that events detected by RF readers are identified by an exact object identification. Instead they focus on issues like cleaning redundant readings and inferring objects' precise locations from several overlapping sensors. In other words, these works tackled fundamentally different problems from our problem of uncertain object identification inference.

On the other hand, the probabilistic data association (PDA) problem is closer to our target problem, as PDA aims to determine the correct correspondence between measurements and objects [20]. The most widely-used approaches to tackle PDA are MHT [81] and JPDAF [88]. These approaches establish probabilistic models based on the fact that in a typical PDA application, such as identifying targets in radar observations or tracking people in a video, the events never carry any object identification. Hence, if we were to apply the PDA techniques to our event stream mixed with ID-ed and non-ID-ed events, they would fail to take advantage of ID-ed events for inference. This then would result in limited precision as confirmed by our experimental analysis. Besides, existing work of PDA largely focused on modeling [20, 81, 88], while the efficiency of processing has been overlooked - which is now a key objective of this dissertation.

1.4 Research Challenges

Our goal is thus to fill the voids as derived in the state-of-the-art (Sec. 1.3), so as to meet the emerging requirements derived from our representative application. In particular, we will tackle the critical challenges listed below.

- To deal with the concurrent interaction between pattern queries and the actions to queries, a common way is to enforce concurrency control. However, existing concurrency control schedulers [23, 48] are based on the notion of a "database transaction" - the execution of a finite sequence of one-time data manipulation operations on conventional stored data sets [23]. While in our stream environments pattern queries are continuously executed on potentially infinite data streams. This implies that we cannot "finish" the current query as a finite-scoped operation before processing another query. In short, we face a fundamental challenge that the concept of a transaction has not been established for stream processing. Furthermore, transactional execution usually has very strict scheduling constraints, while our CEP applications require rear-real-time responsiveness. It is hard to guarantee the correctness of concurrent streaming operation scheduling as well as to achieve high responsiveness over huge volume event streams.

- The intuitive goal of privacy-preserving CEP is to suppress events to eliminate the occurrence of private patterns, at the same time to maximize the output of useful non-sensitive patterns. A natural way to prevent a private pattern match is to *suppress events* that participate in the match. The tricky part is that private patterns and public patterns are correlated. Namely, dropping an event could avoid disclosing a private pattern match while consequently eliminate some useful pattern matches. We thus need to solve an optimization problem of utility-maximizing continuously over event streams. Given the unpredictable feature of event stream and

the huge number of decisions, i.e., whether to drop or keep an event is a *trade-off* between reporting useful public query matches and disclosing undesirable private query matches, this optimization problem is intricate to find an optimal solution. Further, we have to solve this problem in an on-line fashion to achieve real-time system responsiveness.

- We first need to capture the underlying event stream generation process from the physical world, including the key component−the temporal correlations among events. The goal is to devise a time-varying graphic model, based on which we then infer the object identification of non-ID-ed events. Unfortunately, most probabilistic inference algorithms are off-line and time-consuming. For example, the Forward-backward algorithm [74], a classical inference approach is suitable for our inference logic. However, our experimental evaluation demonstrates that this approach, is not efficient enough to provide near-real-time system responsiveness nor scalable for a high volume event stream. We have to devise the strategies for optimizing the performance of Forward-backward algorithm, while still offering high-precision inference results.

1.5 Contributions of the Dissertation

This dissertation focuses on extending and applying the state-of-the-art complex event processing model and infrastructure to meet the needs derived from our motivating application by especially addressing the challenges presented in the Challenge section (Sec. 1.4). The main contributions of this dissertation include the following.

Stream Transaction Model and Active Complex Event Processing [91–93]. To support real-time actions of pattern queries, we propose to embed active rule support within the complex event processing paradigm. The key challenge is handling interac-

tions among continuous queries and active rules. We identify this problem by introducing the notion of a transaction in the stream context. Such novel stream transaction model empowers us to leverage classical concurrency control approaches originally designed for static databases to now solve the novel stream transaction scheduling problem. I further develop stream transaction scheduling algorithms to solve the problem. The Strict-Two-Phase-Locking (S2PL) scheduler successfully applies a pessimistic concurrency control mechanism to the ACEP context. However, S2PL incurs a large synchronization delay due to its rigorous order preserving. Hence we will provide a unique scheduler called Low-Water-Mark (LWM) which is customized to our stream context to maximize concurrent execution without compromising correctness. We have implemented the proposed techniques within HP CHAOS engine [49] and conducted extensive experiments to evaluate their effectiveness and efficiency.

Furthermore, we study the complications of failure recovery in a transaction stream system. We then propose a new recovery strategy that dynamically combines stream data backup and stream transaction logging. Our strategy enables a tradeoff between the recovery time and the volume of checkpoints required to provide safety.

Privacy-Preserving Complex Event Processing Over Event Streams [51, 90]. First we formally define the problem of utility-maximizing event suppression with privacy preferences. Thereafter, we analyze possible variants of the problem. Our problem formulation enables users to specify the relative importance of preserving certain private leaks versus producing useful public pattern matches. We then design a suite of real-time solutions that eliminate private pattern matches while maximizing the overall utility. We first propose an approach based on linear programming that optimally solves the problem at the event-type level (which suppresses all event instances of the same type). For the computationally intractable instance-level problem (which suppresses individual event instances), we observe that our solution at the event-type level provides a useful basis for

further optimization. Specifically, we introduce the Hybrid solution to tackle the instance-level problem by combining the solution at the event-type level with optimization heuristics based on run-time pattern match cardinality estimation. We further develop two techniques to address the subproblem of pattern match cardinality estimation, one based on event arrival rates, and the other based on periodicity using the state-of-the-art periodicity mining algorithms. Finally we conducted extensive experiments on both real-world and synthetic event streams. We show that overall, our Hybrid solution preserves significantly more utility than alternative approaches. Also our proposed solutions are efficient enough to offer near real time system responsiveness.

Probabilistic Inference of Object Identifications for Event Stream Analytics [94]. We propose a probabilistic inference framework called FISS that efficiently transforms raw streams of ID-ed and non-ID-ed events into queriable streams of events with probabilistic object identifications. Specifically, we first devise a time-varying graphic model to capture the underlying event stream generation process from the physical world. Based on our proposed model, we extend a classical inference approach, the Forward-backward algorithm [74], to infer the object identification of non-ID-ed events. However, our experimental evaluation demonstrates that this approach, though suitable for our inference logic, is not efficient enough to provide near-real-time system responsiveness nor scalable for a high volume event stream. We then devise a suite of strategies for optimizing the performance of the Forward-backward inference. Our key insight is that the Forward-backward algorithm conducts a large number of unnecessary computations during the backward smoothing. We aim to avoid such waste by only computing the "affected" events, i.e., events whose distributions should be revised in the backward smoothing. Our first strategy is to *prune random variables* that can be shown to be unaffected by exploiting the features of ID-ed events. The second optimization, called *finish-flags* mechanism, enables early termination of the backward computation yet without sacrificing inference

precision. Lastly, we propose to represent temporal conditional dependencies using CEP *pattern queries* to capture temporal correlations of events in a large volume stream. And then chasing down "affected" events can be transformed into a pattern matching. Meanwhile, we devise an advanced data structure customized for streaming uncertain events to speed up the optimized backward probability computation. These strategies together lead to a solution that keeps up with high-volume streams while offering high-precision inference results. Finally, the experimental results demonstrate that our proposed model achieves better inference precision compared to the MHT model [1, 81]. Moreover, our optimization techniques for the Forward-backward algorithm make it work 15 times faster than the basic implementation [2].

In summary, this dissertation focuses on the three tasks listed above. It is worth noting that the techniques developed in this dissertation are general in the following ways. First, though the techniques deal with issues in the CEP context, the requirements listed above are also common in the general data stream management systems. Many of our proposed solutions, including the stream transaction model, stream transaction scheduling algorithms in Chapter 3, the stream data cardinality estimation algorithms in Chapter 4 and the probabilistic inference approaches over streams in Chapter 5 are applicable in the DSMS context as well. Second, this work sheds a light on extending CEP with crucial components. For each sub-problem studied in this work, we have explored alternative schemes to enhance the existing CEP model and engine. Our solutions proposed for each sub-problem range from the basic one that needs minimal change of an existing CEP engine, to the other extreme that embeds the functionality inside the CEP kernel. Many experiments further show the trade-off between the infrastructure and performance.

1.6 Road Map

The rest of this proposal is organized as follows. Chapter 2 first provides the background and preliminary materials needed for this dissertation proposal. Chapter 3 presents the models and techniques for active complex event processing. Chapter 4 solves the privacy-preserving complex event processing problem. Chapter 5 studies the probabilistic inference of non-ID-ed events. Chapter 6 discusses the related work. Finally, Chapter 7 contains the future work.

Chapter 2

Preliminary

2.1 Complex Event Model

An event instance is an occurrence of interest in a system which can be either primitive or composite as further introduced below. Each event instance e_i has two time-stamps, *application time*, denoted as $e_i.ts$, and *system time*, denoted as $e_i.sts$ [84]. The application time refers to the discrete moment of the occurrence of the event instance assigned by the event source. While the system time of an event instance is assigned by the CEP engine using the system wall-clock time when the event arrives at the system. $e_i.st$ and $e_i.et$ denote the start and the end application timestamp of an event instance e_i, respectively, with $e_i.st \leq e_i.et$.

A primitive event instance denoted by a lower-case letter (e.g.,'e_i') is the smallest, atomic occurrence of interest in a system. For a primitive event instance e_i, $e_i.st = e_i.et$. For simplicity, we associate e_i a single application timestamp, denoted as $e_i.ts$, where $e_i.ts = e_i.st = e_i.et$. In this proposal, we use the superscript attached to a primitive instance to denote its timestamp, e.g., e^{12}.

A composite event instance, denoted as ce_i, is composed of constituent primitive event instances $ce_i = < e_1, e_2, ..., e_n >$. A composite event instance e occurs over an interval. In this dissertation, following the state-of-the-art literature [27, 70, 98], a composite event output by a pattern query is assigned a single application time when the last event instance that composes the composite event occurs. Specifically, $ce_i.ts = max\{e_j.ts \mid \forall\ e_j \in ce_i$ $\}$. In other literature [21, 61], it is also common to assign the start and end timestamps of ce_i, namely $ce_i.st = min\{e_j.st \mid \forall\ e_j \in ce_i\ \}$ and $ce_i.et = max\{e_j.et \mid \forall\ e_j \in ce_i\ \}$, respectively.

An *event type* is denoted by a capital letter, say E_i. An event type E_i describes a set of attributes that the event instances of this type share. An event type can be either a primitive or a composite event type. *Primitive event types* are pre-defined in the application domain

of interest. *Composite event types* are aggregated event types created by combining other primitive and/or composite event types to form an application specific type. $e_i \in E_j$ denotes that e_i is an instance of the event type E_j. Suppose one of the attributes of type E_j is $attr_j$ and $e_i \in E_j$, we use $e_i.attr_j$ to denote e_i's value for that attribute $attr_j$.

Finally, an *event stream* is an unbounded sequence of events.

2.2 Complex Pattern Query Language

A pattern query specifies how individual events are filtered and multiple events are correlated via time-based and value-based constraints. The syntax of defining a pattern query over event streams we adopt here is commonly used in the literature [70, 98]:

```
CREATE QUERY <query-name>
PATTERN <event-pattern> ON <event-stream>
[WHERE <qualification>]
[WITHIN <window>]
RETURN <output-specification>
```

The `event-pattern` describes an event pattern to be matched. In the event pattern, we say an event e_i is a positive (respectively negative) event if there is no "!" (respectively with "!") symbol used before its respective event type. Positive events appear in the final query result while negative events do not. The `qualification` clause imposes predicates on event attribute, as in state-of-the-art CEP engines. The `window` clause describes the maximum time span during which events that match the pattern must occur. The `output-specification` clause defines the expected output stream form by the pattern query.

The sequence (`SEQ`) pattern is a core functionality for pattern queries that specifies a particular order in which the events of interest must occur.

Definition 2.1 *A sequence pattern operator (SEQ) specifies a particular order in which the event instances of interest should occur and these event instances form a composite event instance. That is,*

$$SEQ(E_1, E_2, ..., E_n)[S] = \{< e_{i1}, e_{i_2}, ..., e_{i_n} > |(e_{i_1}.ts < e_{i_2}.ts... < e_{i_n}.ts) \wedge < e_{i1}, e_{i_2}, ..., e_{i_n} > \in E_1[S] \times E_2[S] \times ...E_n[S]\}$$

The *accepting event type* refers to the last event type specified in a sequence pattern. We focus on the SEQ with positive accepting event type in this work, while negative accepting event handling refers to [63].

Example. We now explain the clauses using examples drawn from our motivating healthcare application, HyReminder system (Section 1). Assume the raw sensor reading is bound to semantic knowledge, so that each input event has the schema *(timestamp, worker-ID, behavior, location)*.

```
CREATE QUERY Q1 ON estream
PATTERN SEQ(EXIT, !SANITIZE, ENTER)
WHERE EXIT.workerID = SANITIZE.workerID AND
  EXIT.workerID = ENTER.workerID AND
  EXIT.location != ENTER.location
WITHIN 45 sec
RETURN ENTER.HCW-ID, ENTER.location
```

In the pattern query Q1, the SEQ operator `SEQ(EXIT, !SANITIZE, ENTER)` together with the window constraint `WITHIN 45 sec` detects a specific HCW behavior pattern. Namely, a health care worker does not wash hands after exiting a patient room, and then enters another patient room within 45 seconds. The equivalence test on the

common attribute, `[HCW-ID]`, across an entire event sequence ensures that the pattern is regarding the same individual health care worker.

Chapter 3

Active Complex Event Processing and Stream Transaction

3.1 Introduction of Active CEP

Complex patterns of events often capture exceptions, threats or opportunities occurring across application space and time. Complex Event Processing (CEP) technology has thus increasingly gained popularity for efficiently detecting such event patterns in real-time [13, 21, 27, 70, 98]. However, to allow CEP technology to be an end-to-end solution, beyond monitoring the world via pattern queries, we also need to *react* to the risks and opportunities detected by pattern queries in real-time. We now illustrate this need using a healthcare application as a representative example.

Motivating Application. We consider the representative application, HyReminder [93], as a running example. Recall that the HyReminder system is a hospital infection control system developed by WPI and UMass Medical School to continuously track healthcare workers (HCWs) for hygiene compliance (for example sanitizing hands and wearing masks), and to remind HCWs to perform hygiene precautions - thus preventing the spread of infections [93]. As shown in Figure 3.1, every HCW wears a RFID badge which has a three-color light for indicating his (hygiene) status: “safe”, “warning” or “violation”. Pattern queries continuously monitor each HCW’s behaviors observed by sensors. The status of each HCW is continuously changed upon detecting certain event patterns. For example, a detected hygiene violation pattern will change the HCW’s status to “violation” (henceforth his badge light). In order to urge a HCW to perform precautions immediately upon the detected hygiene violation, it is critical for us to support such *real-time reactions* for the pattern queries. Furthermore, the status of each HCW is also used as a condition by pattern queries, e.g., a pattern query may only monitor HCWs in “safe” status. Clearly the reactions for queries that update HCWs’ status will *affect query results in turn*. It is absolutely vital to control such *concurrent* updates and accesses so as to assure the correct execution logic. Otherwise we risk for a potentially highly contagious disease to

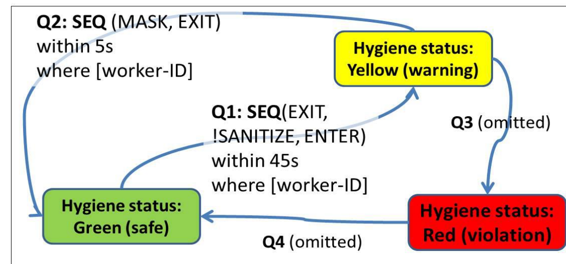


Figure 3.1: Representative hand hygiene logic. If the HCW is in the *start-status* node and his behavior matches the *pattern-query* annotating an outgoing edge, then change his status to the *end-status* node.

be transmitted to vulnerable patients - thus increasing patient suffering and even causing deaths.

Further we found that the requirements derived above are prevalent across applications ranging from algorithmic trading to fraud detection. Unfortunately, current CEP systems support "read-only" query processing. Pattern queries only contain *in-place* conditions, i.e., the query qualification is either based on the attributes of events matched in the input stream [21, 27, 70, 98] or on static information pre-loaded once yet not updated during query execution [6, 13, 41]. Moreover, reactions for event detections are limited to "side-effect-free" actions like sending a message or logging events that do not affect the event detection in turn [95]. Supporting concurrent update actions within the continuous execution of pattern queries over streams, especially actions that directly affect the pattern query results themselves, is an open problem.

Supporting Active Rules. We propose to tackle this unsolved problem by embedding *active rule* support within the complex event processing paradigm, henceforth called Active CEP (ACEP). Active rules in ACEP allow us to specify a pattern query's dynamic condition and real-time actions that in turn may affect the query results.

A critical technical challenge is to handle the *real-time mutual effects* between queries and reactions for queries in the high-volume stream execution. In ACEP we abstract such effects as *interactions among continuous queries and active rules*. Apparently, state-of-

the-art data stream systems [6, 27, 30] and CEP engines [21, 70, 98], which commonly use the *push-based* execution paradigm, have not addressed this challenge. In fact, as we demonstrate via real-world examples in Section 3.2, using the push-based execution for interactions among continuous queries and active rules leads to a variety of anomalies and thus erroneous results.

Introducing Stream Transactions. A common approach to deal with interactions between concurrent accesses and updates is to enforce concurrency control. However, existing concurrency control schedulers [23, 48] are based on the notion of a "database transaction" - the execution of a finite sequence of *one-time* data manipulation operations on conventional stored data sets [23]. While in our stream environments pattern queries are *continuously* executed on potentially infinite data streams. This implies that we cannot "finish" the current query as a finite-scoped operation before processing another query. In short, the concept of a transaction has not been established for stream processing. In this work, we fill this void by introducing the notion of a *transaction in the stream context*.

Furthermore, we design transactional pattern query processing to deal with interactions among continous queries and active rules. This processing is especially challenging because concurrency control poses strict time-based constraints, while our algorithms have to work for high-volume streams yet achieve *near-real-time responsiveness*.

Contributions. In this chapter, we have made the following contributions.

I. We propose the first model of integrating active rules into a stream processing system, called ACEP model, which significantly extends the state-of-the-art CEP model to meet the needs of reacting in real-time by affecting the physical or virtual world. (Section 3.3).

II. To characterize the interactions among continuous queries and active rules, we define the notion of correctness for stream-centric execution given concurrent accesses and updates. Based on this notion, we introduce the stream transaction model along with

stream-specific ACID propositions. To the best of our knowledge, our work is the first at introducing the transaction concept into the stream processing context. (Section 3.4).

III. Our model empowers us to leverage classical concurrency control approaches originally designed for static databases to now solve the novel stream transaction scheduling problem. Our Strict-Two-Phase-Locking (S2PL) scheduler successfully applies a pessimistic concurrency control mechanism to the ACEP context. However, S2PL incurs a large synchronization delay due to its rigorous order preserving. Hence we provide a unique scheduler called Low-Water-Mark (LWM) which is customized to our stream context to maximize concurrent execution without compromising correctness. (Section 3.5).

IV. We implement the proposed techniques within the HP CHAOS CEP engine and conduct comprehensive experimental studies using real data streams. We show that LWM achieves orders-of-magnitude better throughput in high-volume workload compared to S2PL (Section 5.6).

V. We design the infrastructure that efficiently implements the ACEP model. Specifically we integrate the active rule extension into the CEP kernel. This approach is in sharp contrast to the systems that tend to build an external software layer on top of the CEP kernel. Based on this infrastructure we develop a rewriting-based optimization technique to expedite rule condition evaluation and reduce intermediate result sizes, thus improving the ACEP system responsiveness (Section 3.7).

VI. Transactional stream processing poses new challenges for stream system recovery. After describing the complications of failure recovery in ACEP, we propose a new recovery strategy that dynamically combines stream data backup and stream transaction logging. Our strategy enables a tradeoff between the recovery time and the volume of checkpoints required to provide safety (Section 3.8).

3.2 Stream Concurrency Problem: Examples

In the commonly-employed *push-based* stream execution paradigm [6, 21, 27, 70, 98], new events are evaluated by continuous queries immediately upon event arrival without any safeguard mechanism to synchronize the concurrent accesses and updates during stream execution. Since continuous queries may vary in their complexities and event consumption, they may exhibit different processing delays. As a result, events with different timestamps tend to co-exist and be simultaneously executed in the system. Using this push-based execution for pattern queries and reactions for queries (represented as active rules) may raise unexpected *anomalies*. We illustrate the problems with three examples drawn from the HyReminder application[1]. Suppose pattern query Q1 continuously detects the event sequence `(EXIT, !SANITIZE, ENTER) within 45 seconds` for HCWs whose in "safe" status when his `ENTER` event occurs. The checking of a HCW's status is abstracted as a `Read` operation on the shared table storing his current status. Suppose an active rule R1 concurrently monitors the output of Q1: once Q1 produces a match for a specific HCW, R1 changes the HCW's status into "warning". This action is abstracted as a `Write` operation on the shared table. Suppose another query Q2 is also executed. Q2 detects the event sequence `(MASK,EXIT) within 5 sec` for HCWs in "warning" status when his `EXIT` event occurs. Figure 3.2 depicts the following examples respectively.

Example 3.1 ***Correct Query and Rule Processing.*** *Let us consider processing an event sub-stream* $\{exit^2$, $enter^{10}$ *and* $exit^{15}\}$ *with the superscript denoting the event's application timestamp. Suppose before application time 10 the HCW was in the "safe" status and* $enter^{10}$ *leads to a match of Q1. Consequently R1 is triggered and updates the HCW's status to "warning". Later Q2 consumes the next input event* $exit^{15}$ *and recognizes that this is an event for a HCW in "warning" status (by accessing the shared table). The*

[1] We assume that the discussion below is regarding the same individual healthcare worker (HCW) and that the status of a HCW is independent from that of others.

above query output and the value of the shared table are both as expected conforming to the desired application semantics.

Example 3.2 ***Read-too-late Anomaly.*** *Let us process the same event stream as in Example 1 using the push-based execution model. Suppose Q1 evaluates $enter^{10}$ first. Then when Q2 evaluates $exit^{2}$, R1 has already updated the HCW's status into "warning". So Q2 would read the HCW status as "warning", though intuitively at application time 2 the status should have been "safe". Subsequently the query processing result of Q2 is "incorrect"- not matching the semantics required by the application. The problem is that Q2 reads the shared table "too late". As depicted in Figure 3.2(b), the dashed lines for $Write_{R1}(P_k)$ and for $Read_{Q2}(P_k)$ cross, indicating the conflict between these two operations.*

Example 3.3 ***Write-too-late Anomaly.*** *Considering the same Q1, Q2 and R1 as above, now suppose Q2 evaluates $exit^{15}$ first. Thereafter Q1 evaluates $enter^{10}$, and then R1 updates HCW's status into "warning". In this case Q2 should have read the status "warning" because after application time 10 the HCW's status should become "warning". However Q2 reads some other value instead. The problem is that R1 writes "too late": the shared table has already been read by Q2 that theoretically should have executed later. As depicted in Figure 3.2(c), the dashed lines for $Write_{R1}$ and for $Read_{Q2}$ cross - this time in the opposite order.*

It is important to note that the above identified problems of stream concurrency are general, and would indeed equally arise in alternate stream-centric computational models, other than the active rule model employed above, that express the reactions to pattern queries. See Section 3.3.2 for details.

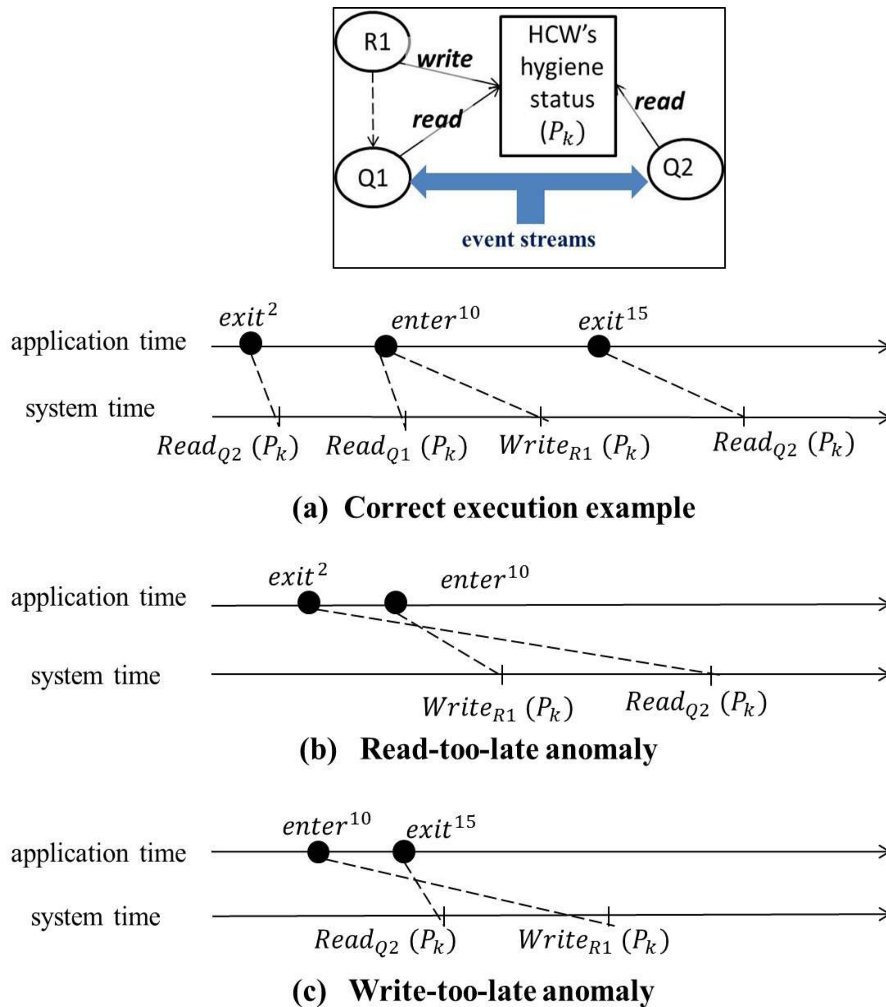


Figure 3.2: Stream concurrency examples. Input events are consumed in their *application time*; Read and Write are executed by the system with *system time*. Dashed line represents that the input event triggers a corresponding operation. E.g., $Read_{Q1}(P_k)$ denotes the Read performed by query Q1 on shared table P_k

.

3.3 The Active Complex Event Model

To support the specification of actions for pattern queries, we design an Active CEP model that integrates active rules into the CEP context.

Event Instances and Types. The input to the ACEP system is a potentially infinite *event stream* that contains all events of interest. Each event instance (e.g., e_i) represents

an instantaneous occurrence of interest. Each event instance has two time-stamps, *application time* and *system time* [84]. The application time for e_i refers to the discrete moment of the occurrence of e_i assigned by the event source, denoted as $e_i.ts$. While the system time of an event instance is assigned by the ACEP engine using the system wall-clock time, denoted as $e_i.sts$. Similar event instances can be grouped into an *event type*. Event types are distinguished by event type names (e.g., EXIT).

Shared Store. In real-world applications, raw event data is typically augmented with semantically richer information. Semantic information regarding the application hence needs to be maintained in the system, referred to as *shared store* through this paper. For illustration, we assume the shared store is organized using the relational model. A table in the shared store can be either *static*, namely the knowledge is loaded once and not updated throughout the system execution (e.g., the mapping between RFID to HCW-ID), or *dynamic*, namely the information can be changed over time (e.g., a HCW's current hygiene status). In ACEP, a shared store thus may be both readable and updatable by multiple queries and active rules. We abstract all data processed in the ACEP system as *ACEP system state*.

Definition 3.1 *Let I be the domain of* ***input event stream****. If i is an input sub-stream in I, then $i =< e_1, e_2, ..., e_n >$ where e_i is an event instance. Let O be the domain of* ***output event streams****. If o is an output stream in O, then $o = \{ce_1, ..., ce_h\}$ where ce_i is a composite event output by a pattern query. Let P be the domain of* ***shared tables****. If p is a shared table in P, then $p = \{t_1, t_2, ..., t_m\}$ where t_i is a shared tuple within the system. All data in the ACEP system, including events and shared tuples, together constitute the* ***ACEP system state****.*

ACEP Query. We now consider basic operations on the ACEP system state prevalent in any ACEP application. For operations over an event stream, we focus on the *dequeue*

operation, i.e., to consume the first available event from the head of a stream[2], and the *enqueue* operation, i.e., to append an event to the end of a stream. These queue-based operations are common across most CEP and stream systems [27, 63, 70, 98]. A CEP query consumes the input stream, executes specified query semantics and then appends the result events to the output stream (if any). Formally we define a CEP query as below.

Definition 3.2 *Let Φ be the domain of* ***dequeue*** *operations, Ψ be the domain of* ***enqueue*** *operations. A* ***CEP query****, q, is defined as a function that takes as argument an instance of Ψ on the input stream, i.e., an arrival of input event, and returns an instance of Φ on the input stream and zero or more instances of Ψ on the output stream. That is, $q : I \times \Psi \rightarrow \{I \times \Phi\} \times \{O \times \Psi\}$.*

Operations over the shared store are *Read* and *Write*. An ACEP query can be viewed as a combination of shared store accesses (e.g., to check the HCW's hygiene status) and a CEP query (e.g., to detect the pattern `SEQ(EXIT, !SANITIZE, ENTER)`). Namely, an ACEP query can defined as below.

Definition 3.3 *Let Rd be the domain of Read operations and Wr be the domain of Write operations on shared tables. An* ***ACEP query****, q', can be defined as a function that: $q' : I \times \Psi \rightarrow \{I \times \Phi\} \times \{P \times Rd\} \times \{O \times \Psi\}$.*

We further abstract a set of one or more operations, including both operations on event streams and on the shared store, as *ACEP system change*, as formally defined below.

Definition 3.4 *Let Δ be the domain of* ***ACEP system changes****, if δ is an ACEP system change in Δ, then $\delta \in \{\Phi, \Psi, Rd, Wr\}$.*

ACEP Rule. As widely recognized [12, 79], active rules have intricate run-time semantics even in static databases. We are the first to explore active rules in the streaming

[2] Here we mean multi-reader dequeuing (for multi-query).

context and thus focus on the core features. An ACEP active rule is **triggered** by the output of an ACEP query. The **condition** of an active rule is a logical test that, if evaluates to true, causes the action of the active rule to be carried out. Similar to the qualification of a pattern query, the logical test can be based on the attributes of the triggering event or on the content of the shared store. The **action** of an active rule supported in the current ACEP model is the Write operation on the shared store. Such active rules are of great use and ubiquitous in ACEP applications as demonstrated by our motivating examples. Namely, pattern queries read the shared store, while the active rules may write the shared store, both in real-time. Hence the newly updated value of the shared store will in turn affect subsequent query execution. Our model includes a *rule type* that describes the definition of a rule, denoted by upper-case letters (e.g., "R_A"), and a *rule instance* that corresponds to an instantiation of a rule type, denoted by lower-case letters (e.g., "r_{A_i}"). Formally,

Definition 3.5 *An **ACEP active rule**, r, is a function that takes as argument an ACEP query output and returns a boolean value (indicating whether the rule is triggered or not) and a set of operations on the shared store. That is,* $r : O \times \Psi \to \{true, false\} \times \{P \times \{Rd, Wr\}\}$.

3.3.1 Active Rule Specification

For illustration, we adopt a declarative active rule language implementing the model described above based on the commonly used ECA format [79, 97].

```
CREATE [OR REPLACE] RULE <rule-name>
ON OUTPUT <query-name>
[REFERENCING NEW AS <new-event-name>]
[FOR EACH EVENT]
```

```
[WHEN <trigger-condition>]
<action-body>
```

For ACEP queries, we are interested in *sequential pattern queries* commonly supported in most CEP systems [27, 70, 98], namely the SEQ operator that specifies a particular order in which the events of interest must occur. We now explain the clauses using examples drawn from our motivating healthcare application. Assume each event has the schema *(timestamp, HCW-ID, behavior, location)*. Also assume the current hygiene status of every HCW is stored in a shared table named *workerStatus*, with the schema *workerStatus:(workerID, status)*. In this example, the Read operation (resp. Write) on HCW status is expressed as a SELECT clause (resp. UPDATE) supported by standard SQL.

```
CREATE QUERY Q1 ON estream
PATTERN SEQ(EXIT, !SANITIZE, ENTER)
WHERE [HCW-ID] AND
EXIT.location != ENTER.location AND
'safe'=(SELECT status FROM workerStatus
  WHERE workerID=ENTER.HCW-ID)
WITHIN 45 sec
RETURN ENTER.HCW-ID, ENTER.location

CREATE RULE R1
ON OUTPUT Q1
REFERENCING NEW AS newEvent
FOR EACH EVENT
BEGIN
  UPDATE workerStatus SET status = 'warning'
  WHERE workerID = newEvent.HCW-ID
```

END

In pattern query Q1, the SEQ operator `SEQ(EXIT, !SANITIZE, ENTER)` together with the window constraint `WITHIN 45 sec` detects a specific HCW behavior pattern. The attribute enclosed in the square bracket, i.e., `[HCW-ID]` stands for the equivalence test on this common attribute across an entire event sequence. The qualification `'safe'=(SELECT ...)` specifies that such pattern detection should only be applied to the HCW who is in "safe" status when he enters the patient room. The active rule R1 is triggered by any output event produced by Q1, represented as `ON OUTPUT Q1`. The action of R1 (defined in the `BEGIN...END` block) states that once a match of Q1 is detected, a rule instance of R1 will update the HCW's status to "warning". It is worth noting that Q1 checks the HCW's hygiene status for query evaluation, while the output of Q1 can result in changing the HCW's status (via triggering R1). In short, here we have demonstrated how to define a pattern query's dynamic condition and real-time action based on the shared table.

3.3.2 Alternate Models

While we have chosen the active rule model as foundation of our solution, alternate formalizations to express the reactions for pattern queries are also possible. By describing these alternate models below, we show that no matter which computational model is employed, the issues of concurrent accesses and updates during stream execution would still need to be tackled. For this, our core innovation, including the correctness notion, stream-transactions and the scheduling strategies (Sections 3.4, 3.5) could continue to be an elegant solution for executing other models.

Feedback Stream Model. We could collect the results of pattern queries into an intermediate stream and then feed this stream back to the queries. In each query, a predicate could read such feedback stream to determine the selective consumption of original input events. Specifically, we could extend an CEP engine with the following functionalities:

(1) Allow a query to delete and/or modify an event in an intermediate stream. (2) Allow multiple queries to publish their results into one single intermediate stream. Note that there needs to be some atomicity and synchronization between the read and removal from such intermediate streams. Now we employ this feedback stream processing model to express our example application logic in Figure 1.

(i) We create an intermediate stream named *InGreenStatus*, which has the schema *(worker-ID, timestamp)* representing that the worker with the *worker-ID* is on the green status since the *timestamp* moment. The operations on this intermediate stream include: `APPEND` a new event (when a worker becomes in green status); `REMOVE` an existing event (when a worker is not longer in green status); `TEMPORAL-JOIN` with a given `to-join-workerID` and `to-join-timestatmp`. `TEMPORAL-JOIN` works as follows: it first checks whether there exists an event, say e_i, in *InGreenStatus* with the *worker-ID* equals to the `to-join-workerID`; if there exists such e_i, then it checks whether $ei.timestamp \leq$ the `to-join-timestamp`. Similarly, create two additional intermediate streams: *InYellowStatus* and *InRedStatus* with the same semantics as above.

(ii) We then define query Q1. Q1 takes two input streams: the original input stream and the intermediate stream *InGreenStatus*. Q1 first detects the pattern `SEQ(EXIT, !SANITIZE, ENTER)`. And then for a matched event sequence, say ce_i, Q1 performs `TEMPORAL-JOIN` on *InGreenStatus* with `to-join-workerID` = ce_i.*worker-ID* and `to-join-timestamp` = $ce_i.ts$. That is, Q1 checks whether the worker with the *worker-ID* was on green status at the moment when his `ENTER` event occurred. If the `TEMPORAL-JOIN` checking returns "true", Q1 inserts an event of the worker into *InYellowStatus*. Namely, a matched Q1 pattern will lead the worker to yellow status. At the same time, Q1 removes the corresponding event of the worker in *InGreenStatus*, because the worker is not longer in green status. Similarly, we also define query Q2 and Q3.

Given the above design, the read-too-late and write-too-late anomalies still arise. For

example, when Q1 tries to read *InGreenStatus* for checking worker007's status at application time 15, the event *(worker007, 12)* should have been already inserted by Q2. However, Q2 in fact inserted such event after Q1's checking. In this case, Q1 should have read the information that "worker007 was on green status at time 15" but read some other value instead.

State Transition Machine Model. We could model the interaction between queries and actions of queries using a state transition table in the form of *(current value of a shared table P_k (v-current), set of queries to execute ($\{Q_i\}$), new value of the store (v-new))*. The semantics are when an input event e_i arrives, if the current value of P_k equals to *v-current*, then the queries in $\{Q_i\}$ should consume e_i for evaluation. If a query in $\{Q_i\}$ produces matches, then the value of P_k is updated to *v-new*. In this setting, atomicity and synchronization mechanisms are still needed when multiple queries read and update P_k simultaneously.

3.4 Stream Transactions

3.4.1 Notion of Correctness

In this section we introduce our notion of correctness for the simultaneous execution of pattern queries and active rules in the stream context. To better capture the time-based properties of active rules, we first design the timestamp assignment mechanism.

Timestamp of active rule instance. At the moment when the triggering event occurs, the change defined by the rule action is assumed to take effect instantaneously. We model this by associating an application timestamp with each rule instance, denoted as $r_j.ts$, and setting the value to be the same as the timestamp of its corresponding triggering event.

Timestamp of an operation on shared store. For a Write operation $Write_i$ on the shared store performed by an active rule instance r_i, we assign $Write_i.ts = r_i.ts$. For a

Read operation $Read_i$ to retrieve the value of a shared store at the application time $t1$, we assign $Read_i.ts = t1$.

Distinguishing Features of our notion of application correctness include: First, it targets ACEP applications, such as the motivating healthcare system, which apply real-time effect to the external world, and thus *do not tolerate the undo or redo* of any externally visible output or action. Second, in our target applications, a Write operation represents a real-time effect, e.g., changing a HCW's badge color. Hence the order of executing Writes must confirm to their timestamp order. In our formal definition below, let $Write_i$, $Write_j$, $Read_k$ and $Read_l$ be operations on a shared table. We use the symbol $\prec$ to denote the preceding order of two operations in the system time.

Definition 3.6 *(Correctness of Real-time Operations). An algorithm for scheduling operations on a shared table performed by pattern queries and active rules is called* ***correct*** *if every schedule produced by the algorithm exhibits the following properties: (1) if $Write_i.ts < Read_k.ts$, then $Write_i \prec Read_k$; (2) if $Write_i.ts < Write_j.ts$, then $Write_i \prec Write_j$; (3) if $Read_k.ts < Write_i.ts$, then $Read_k \prec Write_i$; (4) if $Write_i.ts = Read_k.ts$, or $Read_k.ts \leq Read_l.ts$, or $Write_i.ts = Write_j.ts$, then the order of execution conforms to what the application specifies.*

3.4.2 Stream Transaction Model

When we attempt to exploit the concurrency control principles from static databases [23, 48] to analyze our stream concurrency problem, a fundamental obstacle we encounter that the concept of a transaction has not been established for stream processing. Traditionally a database transaction corresponds to a user program that is invoked when a user explicitly requests so, while our stream query processing is *triggered by incoming streaming events*. Moreover, a database transaction corresponds to the execution of a sequence of one-time

queries [23, 48], while in our stream system the pattern queries are *continuously running*. Therefore we must first define the notion of a transaction in the stream context.

Definition 3.7 *A* ***stream transaction****, or short s-transaction, in ACEP is a sequence of ACEP system state changes that are triggered by a single input event.*

This definition considers the active rule execution to be an *in-line extension* of the triggering transaction [40]. The top of Figure 3.4 illustrates four s-transactions corresponding to three input events respectively. To focus on covering the core requirements drawn from ACEP applications, we first assume there is *no system failure* of the ACEP engine. We then discuss the system recovery for ACEP engine in Section 3.8.

3.4.3 Stream-ACID properties

A database transaction has been traditionally defined to be an encapsulated sequence of user operations that must be ACID [48]. However, these ACID properties cannot directly apply to our s-transactions due to significantly different transactional models. We thus have proposed a mapping of the classical ACID properties to ***stream-ACID properties*** (s-ACID) as follows.

- ***s-Atomic*** (stream-atomic): all operations stimulated by a single input event should occur in their entirety.
- ***s-Consistent*** (stream-consistent): the execution of stream transactions must guarantee to start in a correct ACEP system state and then end leaving the ACEP system state correct (as per Definition 3.6).
- ***s-Isolated*** (stream-isolated): the ACEP system state changes triggered by a single input event must appear to be executed as if no other

input events are being processed at the same time, i.e., no unexpected interactions among s-transactions.

- ***s-Durable*** (stream-durable): the output of the pattern queries must satisfy the "permanently valid" property of query output defined in [63]. That is, at any given time point, all output events from the system so far satisfy the ACEP query semantics.

3.5 Stream Transaction Scheduling

Our s-transaction scheduling algorithms have two objectives: first, to guarantee the *correct* execution of pattern queries and active rules as per Definition 3.6; second, to assure the near-real responsiveness as required by high performance stream processing. The strict application-time based correctness requirement are in conflict with the high system responsiveness requirement, posing great challenges.

We introduce three solutions for s-transaction scheduling to address this challenge. The first solution, called *Single-Event-Initiated* (SEI) scheduler, requires minimum change of an existing CEP engine and hence is easy to use. The second solution adapts a general-purpose concurrency control mechanism, namely the *Strict Two-phase Locking* (S2PL [23]), to the ACEP context, demonstrating that our proposed notion of s-transactions allows us to leverage existing concurrency control approaches. The third solution, called *Low-Water-Mark* (LWM), successfully combines the optimistic and pessimistic principles to maximize concurrent execution in our stream context and achieves high system responsiveness without compromising the correctness. Preliminary features common across all three algorithms are:

- **Orderness of stream.** Following the state-of-the-art literature [49, 70, 98], we initially assume events are fed into our system in strictly increasing application time order.

- **Rule execution order.** For ACEP real-time applications, we assume that the execution order of rule instances that have the same timestamp makes no difference on the appearance of observable actions. This is a reasonable assumption commonly used in state-of-the-art active database literature [79, 97]. It is also practical in our target applications, since it is the application administrator's responsibility to ensure active rules are well-defined.

- **Termination of s-transaction**. Our current ACEP model uses the no-cascading-trigger assumption. That is, rules are triggered by the stream output of queries while rules do not produce any stream output. Hence cycles in triggering will not arise. We also assume every single s-transaction is finite. Consequently, all s-transactions in ACEP are guaranteed to terminate. This model, while simple, has practical utility as demonstrated by our motivating applications. Similarly, most DBMSs in the literature [12, 40] or in practice (e.g., Oracle) either require finite rule specification or they place a hard-coded threshold on the number of cascading trigger calls allowed.

3.5.1 Single-Event-Initiated Scheduler

Based on Definition 3.7, the most direct scheduling strategy would be to take a single input event at a time and to execute all affected queries and rules to converge, i.e., until no more actions are queued to be executed and all output has been generated. This is an intuitive solution due to its simplicity and thus ease of adoption within any CEP engine. However it suffers from the following drawbacks. First, since it only permits one input event to be processed at a time, it may cause a large delay due to blocking a significant number of input events. Second, s-transactions that would not conflict are unnecessarily delayed. This may underload inter-operator buffers and waste processing cycles.

S2PL		requested	
		Read	**Write**
held	**Read**		X
	Write	X	X

LWM		requested	
		Read	**Write**
held	**Read**		
	Write	maybe	X

Figure 3.3: Lock compatibility (X - incompatibility).

3.5.2 Strict 2PL Scheduler

We now relax the single s-transaction at a time constraint. When multiple s-transactions are executed concurrently, an s-transaction obtains “locks” on the shared tables it accesses to prevent other s-transactions from accessing these tables at the same time and thereby incurring the risk of errors. We now adopt and adapt the Strict Two-Phase Locking approach (S2PL) to our ACEP context. Below we assume locks are performed at the table level, yet our schedulers can be easily extended to support locks with multiple granularities.

If an s-transaction T_i intends to write the shared table P_k, T_i needs to have a *write lock* on P_k, denoted as $xl_i(P_k)$. If a transaction T_i wishes to read P_k then T_i requests a *read lock*, denoted as $sl_i(P_k)$. Mimicking the classical S2PL [23], our S2PL inserts locks into an s-transaction ahead of all query and rule processing (known as Phase I). All locks applied by an s-transaction T_i are released after the s-transaction has successfully ended (known as Phase II).

The lock insertion algorithm (Algorithm 9) used in Phase I determines Reads and Writes possibly requested by analyzing active rules and queries. We also employ the lock ordering mechanism to avoid deadlock [23]. Consequently, under our in-order-stream assumption, Algorithm 9 ensures no deadlock will occur hence no rollback will be performed. We have thus shown the successful application of a pessimistic concurrency control mechanism originally designed for static databases, namely S2PL, to the ACEP context. However, S2PL incurs a large synchronization delay due to the rigorous lock

Algorithm 1 InsertLock

```
1: for each input event e_i of type E_i do
2:     initiate s-transaction T_i
3:     if E_i = accepting-event-type of query Q_i
       AND ∃ rule R_j monitors Q_i's output as triggering event
       AND action of R_j contains update on P_k then
4:         insert write lock xl_i on P_k
5:     end if
6:     if ∃ pattern query Q_i
       Q_i consumes e_i and Q_i access P_k then
7:         insert read lock sl_i on P_k
8:     end if
9: end for
```

incompatibility as depicted in Figure 3.3.

Example 3.4 *Let us consider Examples 3.2 and 3.3. First for s-transaction T_1 created for input event $exit^2$, the lock inserted by S2PL is $\{sl_1(P_k)\}$. Then for T_2 created for $enter^{10}$, the locks are $\{sl_2(P_k), xl_2(P_k)\}$. Next for T_3 created for $exit^{15}$, the lock is $\{sl_3(P_k)\}$. As we can see, since T_1 holds the read lock first, $xl_2(P_k)$ from T_2 is delayed until T_1 releases it. So the read-too-late anomaly is prevented. Similarly, $sl_3(P_k)$ held by T_3 must wait until T_2 unlocks P_k. Consequently the write-too-late anomaly is also prevented.*

3.5.3 Low-Water-Mark Scheduler

We now propose the Low-Water-Mark scheduling strategy (LWM) customized for stream-transaction execution.

Consistent application-time based timestamping. A distinguishing characteristic of an s-transaction is that operations issued by an s-transaction have concrete application-time based constraints (as per Definition 3.6). For that reason, we propose to *timestamp an s-transaction* (say T_i) based on the application timestamp of T_i's triggering input event (say e_i). That is, $T_i.ts = e_i.ts$. This assignment is consistent with our previous timestamp

Algorithm 2 LWM(using object-oriented design)

1: **class** Scheduler {
2: Map<STransaction, List¡SharedTable¿> lockTable
3:
4: void RequestRead($Read_i^{ts}(P_k)$){
5: **while** $ts > P_k.GetLWM()$ **do**
6: //blocking
7: **end while**
8: $Read_i^{ts}(P_k)$ is executed }
9:
10: void RequestWrite($Write_i^{ts}(P_k)$){
11: **while** $ts! = P_k.GetLWM()$ **do**
12: //blocking
13: **end while**
14: $Write_i^{ts}(P_k)$ is executed }
15:
16: void TimestampLock() {
17: InsertLock() // given in Algorithm 9
18: $xl_{R_j}.ts = e_i.ts$
19: $sl_{Q_i}.ts = e_i.ts$ }
20:
21: void ReleaseLock(T_i) {
22: List<SharedStore> sStore = lockTable.GetValue(T_i)
23: **for** each store in sStore **do**
24: store.xLockQueue.ReleaseLock(T_i)
25: **end for**
26: }
27:
28: **class** SharedTable{
29: Map<time,string> versions
30: XLockQueue xLockQueue
31: SLockQueue sLockQueue
32: string Read(){}
33: void Write(){}
34: time GetLWM(){} }
35:
36: **class** XLockQueue{
37: List<XLock> locks
38: time lwm
39:
40: void AddLock(xl_{R_j}){
41: locks.add(xl_{R_j})
42: MaintainLWM()}
43:
44: void ReleaseLock(T_i){
45: locks.Remove(all locks held by T_i)
46: MaintainLWM()}
47:
48: void MaintainLWM(){

management for the shared store and active rules (Section 3.4.1), which is a core concept underlying our proposed notion of correctness.

Our scheme of application-time based timestamping is in contrast to the mechanism employed by classical Multi-version Concurrency Control protocol (MVCC) or Timestamp Concurrency Control (TSCC). The later generates timestamps using the system counter in a *first-come-first-served* manner [23, 48]. While our scheme models the real-time application logic that at the moment when the input event occurs, the operations triggered by this event are assumed to take effect instantaneously.

Given our timestamping scheme, to allow different s-transactions to access the specific value of the shared store that is appropriate for each s-transaction's application time based progress, we propose to *maintain historic records (multi-versions)* of each shared table. Specifically, when a Write operation updates a shared table, say P_k, the new record of P_k is appended to the end of the sequence of historical records. This technique designed to increase the concurrent execution level mimics MVCC. However, as we demonstrate below, MVCC cannot be applied to solve our s-transaction scheduling problem directly, as it fails to meet our correctness requirement.

Demonstration of failure of MVCC. In MVCC [23, 48], every shared table P_k has an associated Read-timestamp, denoted as $P_k.Rts$, which is set to the maximum of all executed readers' timestamp. If a transaction T_i wants to write P_k and $Ti.ts < Pk.Rts$, then MVCC will *abort* T_i and *restart* T_i with a new, larger timestamp. First, abort of a transaction, representing an undo of externally visible output or action, is not acceptable in ACEP applications, as stated in Section 3.4.1. Second, restarting a transaction with a different timestamp is equally unacceptable. In LWM, an s-transaction's timestamp reflects the application time when its issued operations should take effect. Hence if a restart were to occur, the newly set timestamp would lead the s-transaction to read and write the wrong version thus incur erroneous results.

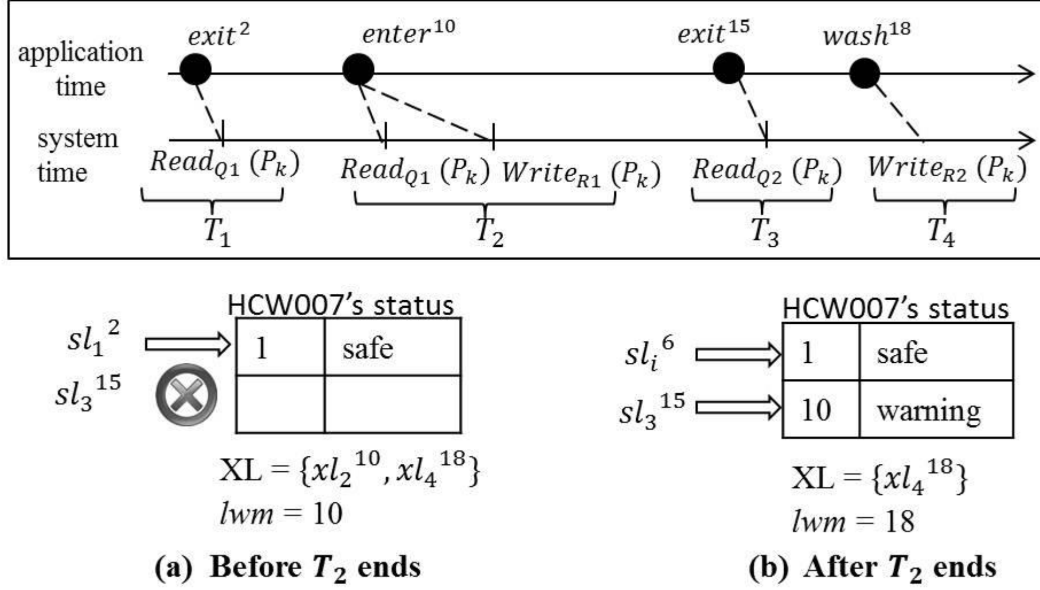


Figure 3.4: S-transactions and LWM Example.

Low-water-mark based scheduling. To avoid those undesired side-effects described above, we leverage the approaches of integrating locking and multiversioning [23]. Our key novel technique is an intelligent strategy to *consistently render the synchronization order* of both locking and timestamping. As the first step, LWM assigns an application-time based timestamp for each lock, after determining locks needed by an s-transaction using Algorithm 9. Specifically, for each write lock $xl_i(P_k)$ (resp. read lock $sl_i(P_k)$) inserted into an s-transaction T_i, we set $xl_i(P_k).ts = T_i.ts$ (resp. $sl_i(P_k) = T_i.ts$).

Observation 3.1 *Given the multi-versioned shared tables and timestamped locks, a write lock $xl_i^{t1}(P_k)$ represents a potential Write on the shared table P_k at application time $t1$. When $Read_i(P_k)$ intends to access P_k at application time $t2$, to avoid the write-too-later anomaly, it is safe to execute the Read only if $t2 \leq t1$. That is, we need to make sure $Read_i(P_k)$ obtains the value of P_k that will never be affected by any future Write.*

Hence, we introduce the *low-water-mark* (lwm for short, a special control parameter) based mechanism to guarantee the correctness. Intuitively lwm represents the oldest

timestamp among all the timstamps of the write locks on a shared table. And lwm is updated whenever a write lock is added or released. lwm is formally defined below.

Definition 3.8 *Given the write lock queue $XL = \{xl_1^{t1}, ..., xl_n^{tn}\}$ on a shared table P_k, the* ***low-water-mark (lwm)*** *of P_k, denoted as lwm_{P_k}, is defined to be*
$lwm_{P_k} = MIN_{j=1}^{n}\{xl_j.ts | xl_j \in XL\}$.

Our LWM scheduler then synchronizes Reads and Writes based on lwms respectively:

- A read lock $sl_i^{ts}(P_k)$ is granted if $sl_i.ts \leq lwm_{P_k}$; otherwise $sl_i^{ts}(P_k)$ is delayed until $lwm_{P_k} > sl_i.ts$. Intuitively, a Read is guaranteed to access the right record of P_k after all Writes earlier than the Read have completed.
- LWM grants a write lock $xl_i^{ts}(P_k)$ only if $xl_i^{ts}(P_k)$ becomes the oldest write lock among all write locks held on P_k, namely only if $xl_i(P_k).ts = lwm_{P_k}$. Basically a Write is not allowed to jump the queue of Write requests on P_k.

Given this novel granting strategy, we now can *relax the lock compatibility* (Figure 3.3). On the one hand, a read lock does not block acquiring a write lock on the shared table. That is, $sl_i^{t1}(P_k)$ will not block $xl_j^{t2}(P_k)$ even if $t1 < t2$. On the other hand, a write lock not necessarily to block a read lock. Namely, $xl_j^{t2}(P_k)$ will not block $sl_i^{t1}(P_k)$ if $t1 < t2$. These compatibilities in turn enable significantly more s-transactions to be executed concurrently.

Example 3.5 *In Figure 3.4 we illustrate LWM using Examples 3.2 and 3.3. Before the s-transaction T_2 created for input event $enter^{10}$ ends, the read lock sl_1^2 is granted and the corresponding Read accesses the proper version, i.e.,* `(1, safe)`*; while the read lock sl_3^{15} is delayed due to the* `lwm` *constraint. After T_2 ends, the* `lwm` *is updated to application time 18. Consequently sl_3^{15} is granted and the corresponding Read obtains* `(10, warning)`*. At this time, any Read operation with timestamp earlier than 18 can be executed immediately and guaranteed to read the right value.*

We now give the pseudocode of the Low-water-mark scheduling algorithm in Algorithm 50. The overall workflow of the algorithm is: the Adding & Timestamping-Lock procedure inserts appropriate locks, (similar to the Phase I of S2PL), and timestamps locks and passes the locking information to the Maintaining-LWM procedure. The Granting-Lock procedure determines whether to delay or execute Read and Write operations based on the low-water-mark of a particular shared table. The Releasing-Lock procedure releases locks, and passes the updated locking information to Maintaining-LWM. The algorithm and proof below assume locking is performed at shared table level. But they can be straightforwardly extended to support locking at different granularities.

3.6 Experimental Evaluation for ACEP

3.6.1 Experimental Settings

We have implemented our proposed ACEP technology within HP CHAOS CEP engine [49]. All queries and rules drawn from our real-world heathcare application are maintained in the *query-rule pool*. The query and rule parameters are listed in Table 3.2.

HP CHAOS system indeed employs multi-threaded processing [49]. Our extension for ACEP also uses multi-thread programming. Specifically, each stream operator is executed by a thread. We execute ACEP on Intel Core 2 Duo CPU 3.0GHz with 3.21GB of RAM running Windows 7 and Java JRE 6.

We use the real-world application workload collected from UMass Medical School Hospital, where our HyReminder system is being deployed for clinical studies. Our input stream adapter feeds workloads collected from the hospital with arrival patterns modeled as the uniform process and with various arrival rates.

Referring to the hand hygiene regulations applied in US hospitals [25], we have created ACEP queries and active rules using the following methodology: (1) model the

Term	Description
e_i^{sts}	An event with system timestamp sts
out_i^{sts}	A query output with system timestamp sts
$numOut$	Number of query outputs so far
T_{ini_i}	System time when the rule instance is created
T_{fin_i}	System time when the rule instance finishes
$numRules$	Number of rule instances executed so far

Table 3.1: Symbols for performance metric

specific sequence of HCW behaviors using *pattern queries*; (2) model the HCW's hand hygiene performance with three status, namely "safe", "warning" and "violation". The status of each HCW is maintained via a shared tuple within a *shared table*; (3) model the logic of a HCW status transitions, namely a certain sequence of behaviors leading the HCW from one status to another status, using *active rules*. We refer an active rule R_j together with the pattern query Q_i whose output is monitored by R_j as a *query-rule pair*. All the application query-rule pairs are maintained in the *query-rule pool*.

Performance Metrics. The performance terms are listed in Table 3.1. The performance metric *throughput* is measured as number of events processed per second. Namely, given a batch of input events of size $numIn$ ($numIn$ is set to be much larger than the maximum window size of all queries), suppose the system time span taken to process the batch is T_{proc}, then $throughput = numIn/T_{proc}$. The metric *average query latency* L_{query} is the average time difference between the time a pattern query Q_i produces an output and the maximum arrival time of any of the event instances that is composed into that output (both system time). Given $out_i = \{e_1, ..., e_m\}$, L_{query} can be measured by:

$$L_{query} = \frac{\Sigma_{i=1}^{numOut}(out_i.sts - max\{e_j.sts | e_j \in out_i, 1 \leq j \leq m\})}{numOut} \tag{3.1}$$

The metric *average rule latency* L_{rule} corresponds to the average time difference between

the time a rule instance is initiated and the time the rule instance finishes, i.e.,

$$L_{rule} = \frac{\Sigma_{i=1}^{numRules}(T_{fin_i} - T_{ini_i})}{numRules} \quad (3.2)$$

The metric *average combined latency* $L_{combined}$ corresponds to the sum of the average query latency and the average rule latency, i.e.,

$$L_{combined} = L_{query} + L_{rule} \quad (3.3)$$

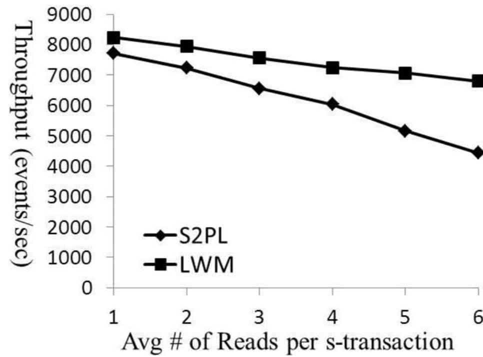


Figure 3.5: *readN* vs. throughput (*lockG*=tuple).

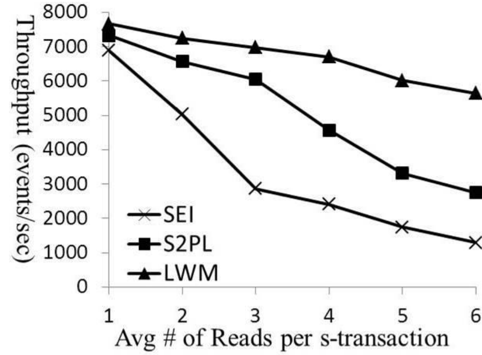


Figure 3.6: *readN* vs. throughput (*lockG*=table).

3.6.2 Experimental Results

In this section we study the performance of our s-transaction scheduling algorithms, namely SEI, S2PL and LWM. The performance metrics we measure include *throughput*, *average query latency* L_{query}, *average rule latency* L_{rule} and *average combined latency* $L_{combined}$,. For S2PL and LWM, we employ the locking at two-level hierarchy of the shared store: *table-level*, e.g., all HCWs' status, and *tuple-level*, e.g., each individual HCW's status.

Varying Number of Reads vs. Throughput. We evaluate the throughput while

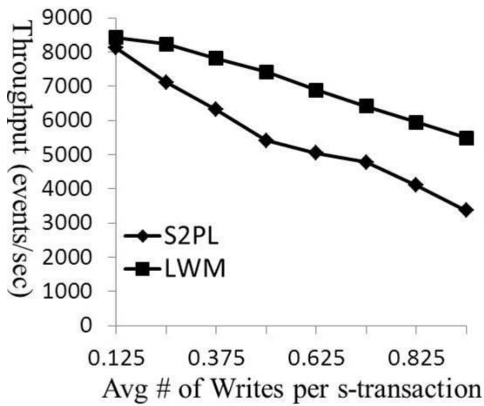


Figure 3.7: $writeN$ vs. throughput (*lockG*=tuple).

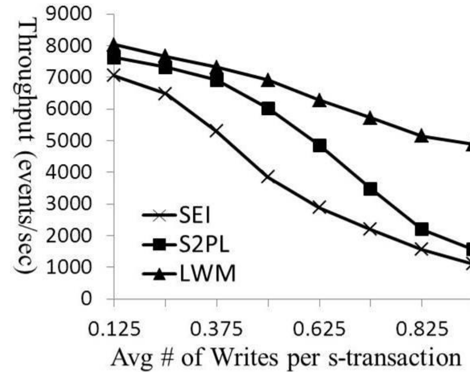


Figure 3.8: $writeN$ vs. throughput (*lockG*=table).

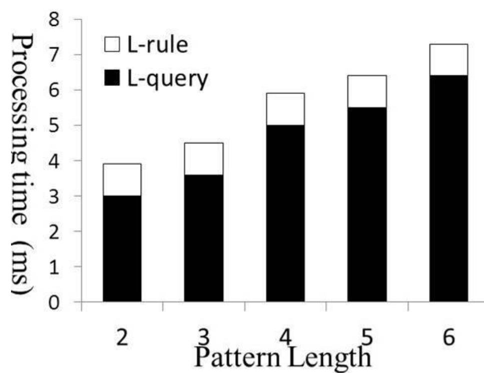


Figure 3.9: Baseline latency.

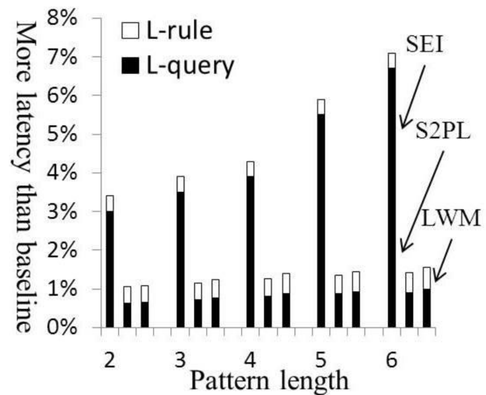


Figure 3.10: Overhead of schedulers.

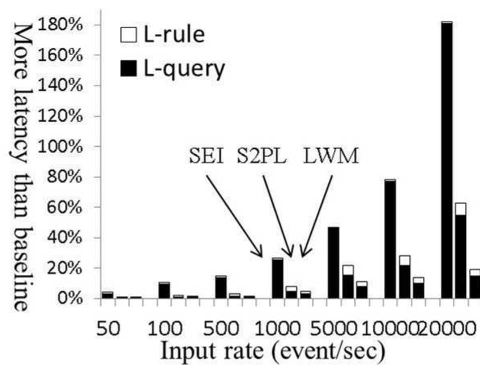


Figure 3.11: Input rate vs. latency.

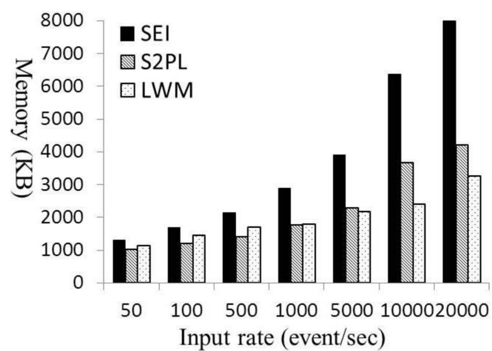


Figure 3.12: Input rate vs. memory.

Par.	Description
seqL	length of the pattern in a query
win	time-based window size (in sec)
readN	average num. of Read per s-transaction
writeN	average num. of Write per s-transaction
lockG	lock granularity (table/tuple)

Table 3.2: Parameters for queries and rules.

varying the average number of Read operations per s-transaction ($readN$) from 1 to 6. The average number of Write operations per s-transaction ($writeN$) is set to 0.25 by selecting the corresponding query-rule pairs from the pool. S2PL and LWM perform locks at tuple level. Figure 3.5 shows that LWM scales more gracefully than S2PL. This is mainly because LWM is capable to grant significantly more locks concurrently.

We next measure the throughput when the lock granularity for S2PL and LWM is at the coarsest level, i.e., $lockG = table$. Figure 3.6 shows the results for the three schedulers when we vary $readN$ from 1 to 6. SEI's performance degrades rapidly as $readN$ increases. Because more processing delay per s-transaction leads to a longer blocking period for subsequently arriving events, which in turn causes more processing delays to accumulate. Compared to Figure 3.5, the performances of both S2PL and LWM degrade due to the coarse lock granularity reducing the concurrency level. However LWM still outperforms the other schedulers, i.e., LWM achieves 2.5x higher throughput than S2PL and 3.4x higher than SEI on average, and gains 3x higher throughput than S2PL and 4.5x higher than SEI when $readN = 6$.

Varying Number of Writes vs. Throughput. We here consider the throughput while varying the average number of Write operations per s-transaction ($writeN$) from 0.125 to 1.0. The average Reads per s-transaction ($readN$) is set to 1. The lock granularity is set to be at tuple level. Figure 3.7 shows that the throughputs drop as $writeN$ increases. One main reason is that the average processing complexity per s-transaction rises. The

increasing cost also comes from more write-locks being requested. LWM beats S2PL due to allowing more overlaps among s-transactions.

Next we set the lock granularity to be at table level. As per Figure 3.8, the throughput of S2PL and LWM become more sensitive to $writeN$ compared to Figure 3.7. Especially when $writeN = 1$, S2PL performs similar to SEI, because on average one table-level write lock is held for every s-transaction and S2PL in fact serializes every s-transaction while synchronizing very few (if any) concurrent executions.

Overhead of Schedulers: Varying Pattern Length vs. Latency. In this set of experiments, we set up the *underloaded* scenario, in which the input rate is set to be much less than the saturation level of the engine. In this scenario, no transaction conflict will occur hence no transactional scheduling is needed. We first measure the empirical baseline of L_{query} and L_{rule} by executing a single query-rule pair without running any transactional scheduler. The baseline results are depicted in Figure 3.9.

And then we execute the schedulers in the underloaded scenario so as to observe their overhead. Figure 3.10 depicts L_{query}, L_{rule} and $L_{combined}$ relative to the baseline (as per Figure 3.9). For example, 2% in the figure corresponds to 2% more latency than the baseline. In this experiment the pattern query length ($seqL$) varies from 2 to 6, and S2PL and LWM perform locks at the tuple granularity. SEI incurs 3x to 6x larger combined latency than S2PL and 3x to 5x larger than LWM. The primary cause is that S2PL and LWM only invoke locks for s-transactions that may potentially read or write the particular tuple. Instead SEI gives every s-transaction the exclusive access to all shared tuples, hence brings significant overhead. LWM bears a slightly larger combined latency than S2PL in this underloaded scenario due to maintaining multi-versions and lwms.

Varying Input Rate vs. Latency. We measure the latency while varying the input rate from 50 to 20000 events/sec. The parameters are $readN = 1$, $writeN = 0.25$, $seqL = 3$, $lockG = tuple$. Figure 3.11 shows the latency relative to the baseline (Figure 3.9). We

observe that the latency rises as the input rate increases because more events have to be held back waiting to be processed as more events arrive per unit of time. LWM incurs $1/9$ of the combined latency of SEI and $1/3$ of the combined latency of S2PL when input rate is 2000 events/sec. This confirms the effectiveness of our design of LWM maximizing concurrent execution.

Varying Input Rate vs. Memory. From Figure 3.12 we can see that, the memory consumption for all schedulers increases with input rate as expected. LWM spends extra memory to maintain multi-versions of the shared store. However, when input rate $\geq$ 5000, the memory used to buffer input events due to processing delays becomes dominant.

3.7 ACEP Infrastructure

In this section we design the infrastructure that efficiently implements the ACEP model described in Section 3.3. We first consider possible architectures.

3.7.1 Architectural Considerations

Alternative 1: Loosely-coupled. One option would be to consider the CEP engine as a black box (an off-the-shell component) for executing pattern queries. That is, we would not change anything within the kernel of the engine. However existing CEP technology only provides pattern matching services. We would need to design several extensions in order to achieve the full functionality of ACEP. The extensions for supporting active rules would be done as loosely-coupled software components on top of the CEP engine, in the form of a dedicated middle-ware or a hard-coded application.

Alternative 2: Build-in. Instead of adding loosely-coupled components on top of the CEP engine, our proposal is the development of a novel architecture that directly realizes the desired active rule functionality as part of the CEP engine itself, henceforth referred

as “Build-in” solution.

Table 3.3 outlines the primary differences between these infrastructure design choices. The metric of *Concurrency* refers to the real-time concurrency control problem raised by the interactions among pattern queries and active rules in the streaming context. The Loosely-coupled solution cannot easily support real-time concurrency control because the external components could not for instance control the underlying CEP engine to block certain query processing for the sake of assuring correctness. Moreover, the performance of the Loosely-coupled approach may suffer as efforts have to be made for crossing the border between the CEP engine and the application layer repeatedly for the handling of possibly one single rule. In contrast, the Build-in approach would establish real-time concurrency control as part of the ACEP services hence offers better guarantees. Also the Build-in approach provides integrated rule processing, which not only holds the promise of fine-grained optimizations but also assures that the message passing is done in a timely fashion. In summary, we chose the Build-in architecture for ACEP.

Metric	Loosely-coupled	Build-in
Concurrency	user-defined, on top of kernel	coherent system service
Communication	messages across layers	integrated process
Optimization	only application level	fine-grained, kernel level

Table 3.3: Architectural considerations

Figure 3.13 illustrates the system architecture of our ACEP system. The *Semantics Repository* maintains the shared store within the application space. The *Transaction Manager* handles real-time concurrency control and provides reliable interfaces to read and write the shared store for the CEP engine and the Rule Manager. The *Complex Event Processor* supports long-running pattern matching queries with access to the shared store. A query issued by the application is rewritten by the query optimizer into an efficient form and then passed to the CEP engine for execution. The *Rule Manager* is tightly integrated with the CEP processor. The rule rewriter collaborates with the query optimizer to convert

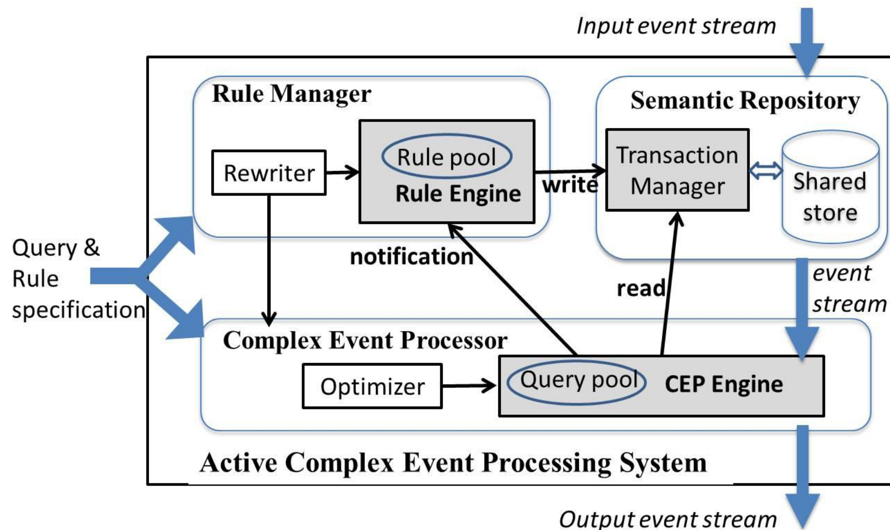


Figure 3.13: Active CEP System Architecture

a rule into an alternate, potentially more economical form. Rule event detection is implemented by placing the checking and notification code inside the kernel of the CEP engine. The rule action may write to the shared store and in turn affect the query processing.

3.7.2 The ACEP Execution Plan

We now present our query and rule processing techniques for the ACEP system. A *query plan* in ACEP consists of a data-flow pipeline of stream operators, including *SEQ, window, static-predicate, active-predicate, result-construction*. The *SEQ* operator employs a non-deterministic finite automaton (NFA) for pattern retrieval [98]. We also push *window* filtering into *SEQ*. The *static-predicate* and *result-construction* operators are standard CEP operators supported in our Ecube engine [44]. We thus focus on the active-predicate, in which the evaluation includes the access to the shared store. A *rule plan* in ACEP consists of three sub-components: event-detection, condition-evaluation and action-execution.

Let us take Q1 and R1 defined in Section 3.3 for illustration. A dataflow of pattern

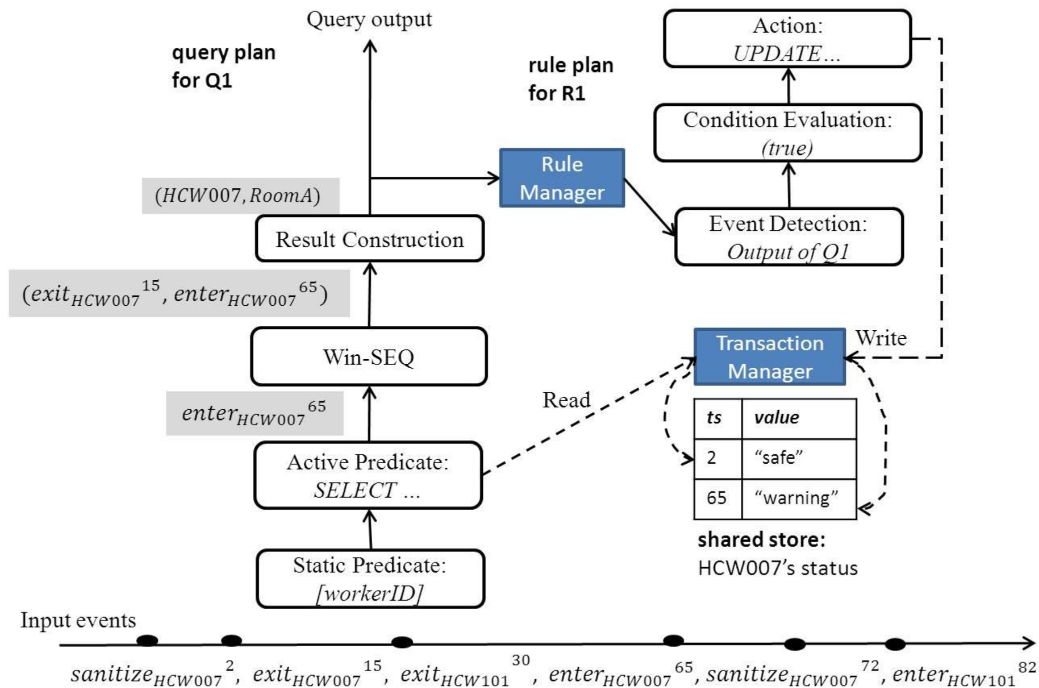


Figure 3.14: Logical execution plan for Q1 and R1

query Q1 and active rule R1 is depicted in Figure 3.14. As we can see in the figure, the left shows the query plan for Q1, the top right depicts the rule plan of R1, and the bottom right illustrates a relevant shared store. An example input sub-stream is depicted at the bottom, where the super-script of an input event denotes the event's application timestamp and the sub-script denotes the HCW-ID of the event.

Existing data stream systems and CEP engines [6, 21, 44, 70, 98] employ the *push-based* execution model. In this execution model, new events are evaluated by all registered queries immediately upon their arrival at the system. The SEQ operator and static-predicate operator in ACEP follow this execution model. Specifically, the execution of the SEQ and static-predicate in query plan Q1 is pipelined: if a newly arriving event triggers a match of Q1 (e.g., $enter^{65}_{HCW007}$), a corresponding event sequence is emitted from SEQ operator right away (e.g., $(exit^{15}_{HCW007}, enter^{65}_{HCW007})$), and pipelined through the subsequence operators.

However, different from the above mentioned existing systems, the active-predicate in ACEP may block the pipelining. That is, the Transaction Manager may delay a Read operation for concurrency control purposes. For example, when Q1 requests to read HCW007's status value, another rule may concurrently be updating this value. Then the Read is delayed until the update completes, so that the value obtained is correct based on the application's logic.

The output of Q1 is passed to the rule plan of R1 in a timely fashion, so that the output can trigger fast reaction to the current situation. If a rule instance's triggering event has been detected and its condition holds, then its action is submitted to Transaction Manager for execution. The Transaction Manager schedules Write operations on the shared store and guarantees the correctness based on the application's business process.

3.7.3 Optimization Techniques

We present the technique of pushing rule conditions down to CEP query, to illustrate that the tight coupling of rule and query processing in our proposed Build-in infrastructure holds promise for exploring mature database optimization techniques.

The Rule Manager needs to be notified for each output event by the pattern query that is monitored by an active rule. Hence if a large number of triggering events are passed to the Rule Manager but later do not lead to final actions due to rule condition disqualification, tremendous work in query processing and rule event notification is wasted. For efficient rule processing, we thus adopt rewriting-based techniques to convert a rule into an alternate, potentially more economical form in ACEP.

An important optimization for preventing unnecessary events from being passed to the rule manager is to evaluate the rule conditions as early as possible, similar to the idea of pushing predicates down into a query plan. Consider the following query and rule derived from our HyReminder application:

```
CREATE QUERY Q2 ON estream
PATTERN SEQ(ENTER,!MASK, EXIT)
WHERE [HCW-ID] WITHIN 60 sec

CREATE RULE R2
ON OUTPUT Q2
REFERENCING NEW AS newEvent
FOR EACH EVENT
IF newEvent.location = 'highly-contagious-area' AND
BEGIN
  UPDATE workerStatus SET status = 'violation'
  WHERE workerID = newEvent.HCW-ID
END
```

Clearly utilizing the basic approach described in Section 3.7.2 will entail passing all output events of Q2 to the rule processing. However only few of the tuples will end up triggering the rule (especially when the rule condition is highly selective). Intuitively, it would be more efficient to *rewrite* the rule into the following:

```
CREATE QUERY Q2 ON estream
PATTERN SEQ(ENTER,!MASK, EXIT)
WHERE [HCW-ID] AND
EXIT.location = 'highly-contagious-area'
WITHIN 60 sec

CREATE RULE R2
ON OUTPUT Q2
REFERENCING NEW AS newEvent
```

```
FOR EACH EVENT
BEGIN
    UPDATE workerStatus SET status = 'violation'
    WHERE workerID = newEvent.HCW-ID
END
```

The above rewriting algorithm aims to decrease the total amount of events being processed by the rule manager. The primary overhead would be the new query registered to the CEP engine, assuming the original query cannot be replaced by the newly re-written one. Our system performs a cost-based analysis for each rule before executing the rewriting algorithm. In short, the choice of rewriting is based on the *expected number of events* that would be filtered by the rule's condition.

3.8 Recovery of Transactional Streams

In this section, we explore the crash recovery mechanism with stream transaction semantics. Recovery in a transactional stream management system poses special requirements as follows.

(1) Stream sources do not stop delivering data to the engine while the system is down, requiring the engine to "catch up" following a crash.

(2) We need to keep both relations and streams in a consistent state after a crash.

(3) The output stream is directly visible by the external world, thus may do not tolerate an undo or a duplicate.

Therefore, we will first study the issue of stream-based data backup, which is an essential part for all stream management systems. And then we discuss the transaction recovery strategy, which need to be specially designed for transactional streams. We also assume that the input stream arrives in a strictly increasing time order, while leaving the

discussion of recovering an out-of-order stream as future work.

3.8.1 Basic Stream Data Backup

Figure 3.15 depicts where the data reside in a typical stream system. As we can see, events in the *Input buffer* are the newest and waiting to be processed by the queries. We have to frequently backup these events, because we cannot afford to lose any event that has not been processed yet. We also keep the timestamp of the latest checkpoint for events in the input buffer, denoted as T_{input}.

When a continuous query is executed in memory, we typically utilize customized data structures to store the events of interest and intermediate results. Figure 3.15 illustrates a NFA and associated queues for Q1, which is widely adopted in CEP engines. As we can see, events are organized based on their event types, and indexes are built between correlated events. To facilitate the "catch-up" task after a failure, we can also backup such *Query data structures*. We use T_{Q_j} to denote the latest checkpoint of a query Q_j. We may choose different volumes of checkpoints for a query based on their run-time progress.

Lastly, output events are enqueued to the *Output buffer*. We use T_{output} to denote the latest checkpoint of the output buffer.

To summarize, T_{input}, T_{Q_j} and T_{output} all refer to the application timestamp of the events that have been backed-up. Therefore we can directly compare the time different between them.

During a failure recovery, for each running query Q_j, we can directly bring the intermediate results of a query to memory, and then re-load the raw events with timestamp from T_{Q_j} to T_{input}. Apparently backing up the intermediate results incurs significant overhead, but it helps expedite the recovery process. An intuitive criteria of whether and how often to add checkpoints for intermediate results is the likelihood of crash recovery.

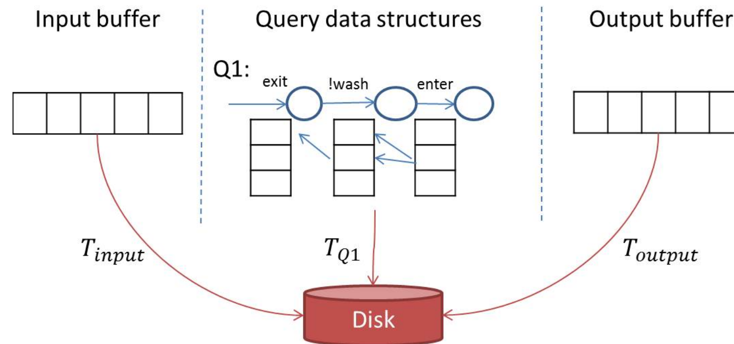


Figure 3.15: Backing-up data in a stream system

3.8.2 Recovery for Stream-Transaction

Now we augment the basic stream system with the notion of transaction and relation data. Using the Stream-Oriented model, a relation will be transformed into an internal stream. Let us consider an example scenario where a system failure makes the transactional stream processing incorrect. Suppose a query intends to update two tables R_1 and R_2 in a single transaction. But after updating R_1 the system fails, while R_2 still has not been updated yet. This leaves the system in an inconsistent state.

To deal with inconsistent system states such as above during a failure recovery, we would leverage the golden standard ARIES [72] for database recovery. Three main principles lie behind ARIES: (1) Write ahead logging; (2) Repeating history during Redo; and (3) Logging changes during Undo. In our Stream-Oriented model, we apply these principles as follows.

First of all, we need to keep logfiles in the stream system. Any update to a stream (i.e., `APPEND` operation) and to a relation (i.e., `WRITE`) is first recorded in the log, and the log must be written to disk before changes to the stream are written to disk.

On restart after a crash, we first restore the data from the latest checkpoint. And then we redo the actions of the stream system before the crash, including all the changes of

uncommitted transactions that were running at the crash time [3]. Redo may generate duplicate events for streams. We thus also need to perform *de-duplication* for streams during the Redo.

Handling Undo. Lastly, we need to undo all uncommitted changes to leave the stream system in a consistent state. In order to support undo during a crash recovery, a special "*revision*" punctuation for internal streams, which represents the modification over a previous event, is also needed. Unfortunately, as we state earlier, the final output stream may or may not tolerant an undo, depending on the particular application.

However we argue that by using a concurrency control protocol that produces *recoverable schedules*, undo over the final output stream can be avoided. The reason is as follows. We know that a transaction has three states: active, commit and abort. Causes of an abortion in our stream-driven processing include: (i) an operation cannot be completed due to hardware or network problem. (ii) an transaction is aborted by the concurrency control protocol. Any aborted transaction will be rolled back. But for an transaction aborted due to hardware/network reason, it will be redone. In other words, once such an uncommitted transaction is redone during the Redo phase, we do not need to undo it any more. On the other hand, by using a conservative concurrency control protocol that does not abort transaction, like SS2PL and Low-water-mark algorithm, we will not face the second condition anyways. Therefore, we ensure that no undo is needed for final output stream during a crash recovery.

In conclusion, the recovery strategy in a transactional stream system dynamically combines the stream-based data backup and the ARIES-style transaction logging.

Optimization. Moreover, further optimization is needed for improving the performance of the transactional recovery mechanism. For example, given the fluctuation of

[3] The underlying logic is that all uncommitted transactions should be rolled back. But we cannot do such rollback directly, because we do not know which portion of an uncommitted transaction has been executed before the crash. Therefore, we need to first re-do the uncommitted transaction to acquire all the changes it would make. And then in next phase, we can undo these changes all together.

streaming data, it is important to provide a wide spectrum of recovery models, ranging from a high-volume checkpoints/fast-recovery approach to a low-volume checkpoints/slow-recovery approach. So that we can trade-off between the recovery time and the volume of checkpoints in accord with the arrival rate of input streams. For example, we may even skip some checkpoints when the stream system is already busy with a burst of inputs.

3.9 Related Work

The scheme of supporting active rules in ACEP borrows several principles from active databases [12, 79, 97], e.g., the "in-line" coupling model [40] and the ECA format. However, ACEP rules differ significantly from active rules in static databases. ACEP rules have the unique formalization of being triggered by the output of continuous queries over infinite streams. Its distinguishing application time based correctness (Def. 3.6) and challenging requirement of near-real-time responsiveness (Sec. 3.5) are also special to the stream context. To our best knowledge, no previous work addressed the issues of defining active rules in the CEP context, nor in the more general data stream processing context.

CEP technologies exhibit sophisticated capabilities for pattern matching in huge volume event streams [6, 13, 21, 27, 70, 98]. However, effort in supporting actions of pattern queries, especially actions that will subsequently affect the pattern matching results, is lacking. Our ACEP technology tackles this unsolved problem by supporting active rules within the CEP context.

Existing CEP engines allow applications to specify business logic as an additional layer of software on top of the engine [6, 13]. Instead, our ACEP system supports active rule facilities as an integrated service of the CEP engine, which frees us from the limitations of the layered approach, providing performance benefits and optimization opportunities as demonstrated in this chapter.

Authors in [95] presents a methodology to define the reactions for RFID event detection. But it only deals with "trivial" reactions like sending a message or logging events, that do not affect the event detection in turn. Such scenario is simpler than the query-rule interaction problem we study in our work - thus the concurrency can be safely ignored in [95].

Transaction processing has been intensively explored in RDBMS. See [23] for a survey of this field. To the best of our knowledge our work is the first attempt at introducing the transaction concept into the stream processing context. Our invention of the stream-oriented ACID proposition and stream-transaction allows us to leverage existing concurrent control principles and further optimize them.

Our innovation then triggers several research efforts on transactional stream processing. Most notable, Botan, Fischer, Kossmann and Tatbul [24] extended our notations to the more general data stream management context, and emphasized the importance of continuous query execution over both static and streaming data in the face of concurrent access and failures. However, while in ACEP we keep both streams and relations in their original forms, the model in [24] only functions on relational data. Namely the authors of [24] propose to transform a stream to an unbounded relation and to represent a continuous query as a set of one-time queries. Though their proposal results in a unified model, such transformation would sacrifice the performance.

Chapter 4

Privacy-Preserving Complex Event Processing via Event Suppression

4.1 Introduction of Privacy-Preserving CEP

While CEP technology has received significant attention from both the database research community [11, 70, 98] and industry [3, 5], its privacy implications have been overlooked. However, reporting CEP patterns without taking privacy preference into account may lead to undesirable privacy violations, if sensitive knowledge patterns are discovered from the output stream. We explain with a concrete motivating example below.

Motivating Applications. Let us consider the real-time health care system HyReminder [93] again. As an example of a pattern query in HyReminder, a doctor who exits a patient room (represented by an `Exit-patient-room` event) should perform hand sanitization (indicated by a `Sanitize` event) within a short period of time. We represent such a hygiene compliance pattern using Q1 below. Similarly, a doctor should wash hands (a `Wash` event) before entering an office (say, an `Enter-psychiatrist-office` event) to prevent spreading contagious diseases, as expressed by Q2 below. These regulations are commonly known as "wash-in, wash-out" in hospital jargon [25].

Q1: `SEQ(Exit-patient-room, Sanitize)`

`WITHIN 2 min`

Q2: `SEQ(Wash, Enter-psychiatrist-office)`

`WITHIN 2 min`

While the benefit of such CEP applications is apparent, the privacy implications of this CEP query model are more subtle. That is, sensitive event patterns that reveal an individual's private information may exist. For example, an observation that a doctor leaves a patient's room and then immediately enters a psychiatrist's office might serve as an indication that this patient is experiencing psychiatric problems. This event sequence, when expressed as a "private" pattern, can be written as:

P1: `SEQ(Exit-patient-room, Enter-psychiatrist-office)`

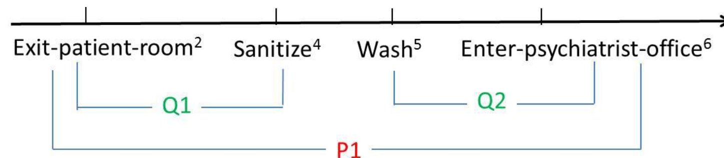


Figure 4.1: An adversarial attack using public pattern matches

```
WITHIN 5 min
```

The issue is that when HyReminder monitors doctors' behaviors (the intension of this application), the events being reported may disclose private information about individual patients as a side-effect. Even though HyReminder will never directly reveal such private information, the occurrences of hygiene compliance patterns that it does report may be used by an adversary to infer the existence of private pattern matches.

As an example, in Figure 4.1, there is a stream of four events associated with the same doctor observed in HyReminder. Each event has a superscript indicating its time-stamp in minutes. Over this particular event stream, there exists one match for Q1, that is (`Exit-patient-room`^{2}, `Sanitize`^{4}), and also one match for Q2: (`Wash`^{5}, `Enter-psychiatrist-office`^{6}). Now by concatenating the query matches of Q1 and Q2, an adversary can infer that a match for private pattern P1, namely (`Exit-patient-room`^{2}, `Enter-psychiatrist-office`^{6}) also exists. This example illustrates that the existence of private pattern matches can still be inadvertently revealed even when only legitimate queries are reported. It underlines the importance of taking private preferences into account when deploying CEP systems in sensitive environments like a hospital.

As another example in which privacy issues can arise in CEP systems, consider a hospital inventory management and asset tracking system [8], which is also deployed in the hospital. Such an inventory management system uses events triggered by RFID tags attached to medical equipment or expensive medicine to provide real-time inventory information, which, among other things, can be used to track supply usage and reduce

inventory costs. Privacy can again be an issue here. Consider the combined output produced by the HyReminder and the inventory management system: the fact that a doctor picks up and checks out some expensive cancer medicine, a pattern reported by the inventory management system, before he/she sanitizes and enters a patient room, a pattern reported by the HyReminder, would be an indication of the patient's health condition. As we can see, the output of the well-intentioned CEP systems, if not handled carefully, can have undesirable effects on patients' privacy.

At a high level, *the real-world problem* we consider is when a CEP system is deployed in a sensitive environment such as health care, retail and financial, the user wishes to mitigate possible privacy leaks while ensuring that the useful nonsensitive patterns can still be reported by the CEP engine. Specially, we consider two kinds of event patterns: those that the user wishes to detect (which we term *"public" patterns*), and those that the user prefers not to reveal because of privacy concerns (which we term *"private" patterns*). The decision of what is public and what is private is made by the system administrator or data owner. The goal of our work is thus to minimize private pattern matches while maximizing public pattern matches.

Event Stream Suppression. A natural way to prevent a private pattern match is to *suppress events* that participate in the match. For example, in Figure 4.1, dropping any event in the sequence ensures that one match of Q1 or Q2 would not be reported, thus preventing P1 from being inferred. In this work we focus on such event suppression mechanisms, while other possible mechanisms are briefly discussed in Sec. 4.2.1.

There are numerous ways in which events in the input stream could be suppressed. It is important to note that whether to drop or keep an event is a *trade-off* between reporting useful public query matches and disclosing undesirable private query matches. We thus provide a *utility-maximizing framework* that allows users to quantify their preferences by setting relative weights of public and private patterns. Then our goal is to offer event

suppression decisions to optimize the overall utility in an online fashion.

To the best of our knowledge, to date no practical solution for this problem of utility-maximizing event suppression with privacy preference in the CEP context has been proposed. Our previous work [52], which is a collaboration with Univ. of Wisconsin Madison, identifies the privacy concern in CEP systems. But in [52] we only studied the hardness of one particular variant of the problem from a theoretical perspective, while no practical algorithm nor empirical studies have been conducted. To fill this void, in this chapter we analyze possible variants of the utility-maximizing event suppression problem, propose real-time algorithms to solve the problem, as well as experimentally evaluate our algorithms using both real-world and synthetic streams.

Contributions. Specially, our technical contributions include:

1. Our first contribution is to formally define the problem of utility-maximizing event suppression with privacy preferences. Thereafter, we analyze possible variants of the problem (Section 4.3). Our problem formulation enables users to specify the relative importance of preserving certain private leaks versus producing useful public pattern matches. We then study the computational hardness of the problem (Section 4.4). This sheds light on designing practical algorithms to solve the problem efficiently.

2. Our second contribution is the design of a suite of real-time solutions that eliminate private pattern matches while maximizing the overall utility. We first propose an approach based on linear programming that optimally solves the problem at the event-type level (which suppresses all event instances of the same type). For the computationally intractable instance-level problem (which suppresses individual event instances), we observe that our solution at the event-type level provides a useful basis for further optimization. Specifically, we introduce the *Hybrid solution* to tackle the instance-level problem by combining the solution at the event-type level with optimization heuristics based on run-time pattern match cardinality estimation. We further develop two tech-

niques to address the subproblem of *pattern match cardinality estimation*, one based on event arrival rates, and the other based on periodicity using the state-of-the-art periodicity mining algorithms (Section 4.5).

3. To better understand the performance of our proposed solutions, we conducted extensive experiments on both real-world and synthetic event streams. We show that overall, our Hybrid solution preserves significantly more utility than alternative approaches. Also our proposed solutions are efficient enough to offer near real time system responsiveness. In addition, we demonstrate that in real world scenarios, periodicity information is needed in order to produce accurate cardinality estimation. The performance advantage of periodicity-based estimation is reproduced and reaffirmed using synthetically generated data streams (Section 5.6).

4.2 The Complication: Private Patterns

Next we introduce the notion of a **private query pattern**. The existence of private patterns and the preference of preventing their disclosure sets our problem apart from conventional CEP literature.

In terms of syntax, private query patterns are just like SEQ queries, i.e., they consist of a sequence of events and an associated time window. The fundamental difference, however, is that while as many query pattern matches as possible should be reported, private pattern matches are undesirable and should not be reported. In the remainder of this work we will refer to these two types of patterns as **private query patterns** and **public query patterns**, respectively. Q1 and Q2 mentioned in the motivating application, for example, are public query patterns, while P1 is a private query pattern. When the context is clear, we simply refer to public (resp. private) query patterns as public (resp. private) queries or public (resp. private) patterns for short. We denote the set of public

query patterns by $\mathcal{Q}$ and the set of private query patterns by $\mathcal{P}$.

There are multiple ways in which an adversary could compromise privacy in search of private patterns. For example, an adversary could potentially break into a CEP system to access the original data stream. This security aspect of CEP systems, while important, is beyond the scope of this dissertation. Instead we focus on the scenario in which the adversary does not violate security, but rather violates privacy by observing the reported matches of public patterns and using that information to infer the occurrence of private pattern matches. For the rest of this chapter we will focus on the externally observable public event sequence instead of the original event sequence.

In this chapter, we focus on queries with positive event types. The problem of supporting negation in event stream suppression is an intriguing area for future work.

4.2.1 Suppressing Private Query Patterns

A natural way to suppress a private pattern match is to suppress events that participate in the match. For example, in Figure 4.1, dropping any event in the sequence ensures that one match of Q1 or Q2 would not be reported, thus preventing P1 from being inferred.

Recall that for now we only consider queries with positive events. In this case using event suppression can ensure *a desirable property*, namely we only report a *subset* of the original CEP query matches. This means no spurious matches that do not exist in reality will ever be produced. This is desirable because the opposite is troubling — it would be disturbing if a hospital hygiene compliance system reported false hygiene compliance/violations because they were generated by the privacy enforcement system.

While we focus on event suppression in this work, other possible approaches also exist. In the classical database anonymization literature, a frequently used technique is *generalization* [62, 68, 86, 99], in which a specific value is "generalized" so that its presence reveals less information. Similarly one could also adopt "event generalization"

for our problem, by generalizing detailed events into generic events. For example, the event `Enter-psychiatrist-office` can be generalized to `Enter-room`, and similarly `Exit-patient-room` can be generalized to `Exit-room`. Although event generalization does mitigate privacy concerns, it can cause trouble in producing query matches. For example, in Figure 4.1, if we only observe the generalized `Exit-room` event instead of the specific `Exit-patient-room`, matches for Q1 cannot be produced in a deterministic manner. Even if the generalized events can be interpreted in a probabilistic way, the matches so produced are no longer a subset of the original query matches, violating the desirable property that event suppression offers.

Another possible data manipulation approach is to "*shuffle*" events around by altering their timestamps. This alternative is also problematic, again because it risks introducing spurious query pattern matches that do not exist in reality, thus violating the important subset property outlined above.

Therefore, we observe that event suppression is a simple yet effective mechanism to preserve privacy in the context of CEP. As such, in this dissertation, we will focus on the event stream suppression strategy. Other possible approaches to tackle the privacy complication are interesting direction for future work.

4.3 Problem Statement and Variants

In this section, we first formally define the problem of utility-maximizing stream suppression for privacy preservation. To better understand the problem, we then study its possible variants by exploring three important dimensions of the problem.

Problem Statement. Since there are multiple ways in which events could be suppressed to preserve privacy, a natural question is which events should be dropped over others. An intuitive answer is to suppress events such that more "useful" public pattern

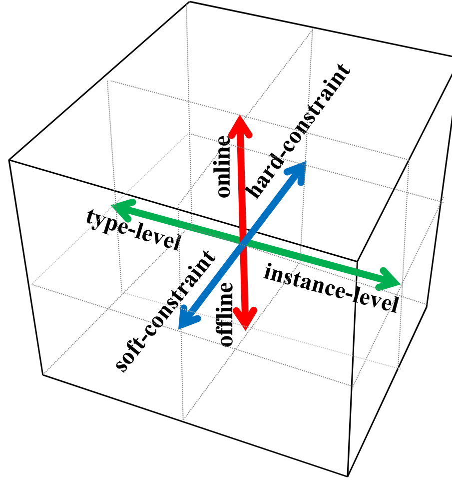


Figure 4.2: Problem space with three orthogonal dimensions

matches are kept. In order to quantify the preference of a public query match, we define a positive **utility weight** $w(Q_i)$ for each public query Q_i, which measures the usefulness of reporting one match for Q_i. For example, in the hospital where HyReminder is deployed, another public pattern, henceforth referred to as Q3, may be needed to produce a query match if a doctor exits a highly-contagious patient room, does not sanitize his hands, and immediately enters the ICU. A match for Q3 represents a grave violation of hygiene regulations. In the application such a match should thus be assigned a higher importance than a match for query Q1 or Q2. Assigning appropriate weights to different queries in real-world applications requires domain expertise. In this work we assume that the utility weights for queries are provided by domain experts as part of the input to the system. We then define *utility gain* as a weighted sum of all public pattern matches as below.

Definition 4.1 *Let $w(Q_i) \in \mathbb{R}^+$ be the utility weight of public query $Q_i \in \mathcal{Q}$, and let $\mathcal{C}(Q_i, S)$ be the number of matches for Q_i over event stream S. The **utility gain** generated for query Q_i is:*

$$U(Q_i, S) = w(Q_i) \cdot \mathcal{C}(Q_i, S) \tag{4.1}$$

The utility gain generated over the entire query set $\mathcal{Q} = \{Q_i\}$ is:

$$\mathcal{U}_{\mathcal{Q}} = \sum_{Q_i \in \mathcal{Q}} U(Q_i, S) \tag{4.2}$$

To model the fact that private pattern matches are undesirable, each such match is assigned a *negative* utility value, or a *utility penalty*. In contrast to the utility gain contributed by a public pattern, the *utility loss* is the side-effect caused by private patterns. We then formally define utility loss as follows.

Definition 4.2 *Let $w(P_j) \in \mathbb{R}^-$ be the utility penalty weight of private pattern $P_j \in \mathcal{P}$, and $\mathcal{C}(P_j, S)$ be the number of matches for P_j over event stream S. The **utility penalty** or **utility loss** associated with P_j is:*

$$U(P_j, S) = w(P_j) \cdot \mathcal{C}(P_j, S) \tag{4.3}$$

Since our goal is to both maximize utility gain and minimize utility loss, our objective function is an *aggregate objective function*. Such aggregate functions are commonly used in the multi-objective optimization literature [85]. While it is natural to define utility as a linear function of the number of matches, there may exist applications in which utility is best defined in other alternative manners (e.g., a submodular function with diminishing returns). Considering such utility variants for stream suppression is an interesting area for future research.

Definition 4.3 *The overall utility generated over $\mathcal{Q} = \{Q_i\}$ and $\mathcal{P} = \{P_j\}$ is thus:*

$$\mathcal{U}_{\mathcal{Q}+\mathcal{P}} = \sum_{Q_i \in \mathcal{Q}} U(Q_i, S) + \sum_{P_j \in \mathcal{P}} U(P_j, S). \tag{4.4}$$

The problem of utility-maximizing stream suppression for privacy preservation can

then be described as follows.

Problem 4.1 *(Utility-Maximizing Stream Suppression) Given an input event stream S, a set of public queries $\mathcal{Q}$, a set of private queries $\mathcal{P}$, and a utility weight function $w(\cdot) \in \mathbb{R}$ associated with $\mathcal{Q}$ and $\mathcal{P}$, find a subset S' of S such that the total utility $\mathcal{U}_{\mathcal{Q}+\mathcal{P}}$ is maximized for the chosen S' over all possible subsets of S.*

4.3.1 Hard-constraint vs. Soft-constraint

In some applications, the user requires that the adversary cannot infer even a single private pattern match. In other words, the constraints imposed by the private patterns are "hard", assuring no private pattern match is ever disclosed. We term such suppression requirement **hard-constraint**. To enforce the hard-constraint, the user can set the utility penalty weights for private patterns to *negative infinity*. In this case we guarantee that there is no privacy leak at all.

What is more important, our problem formulation allows *trade-offs* between reporting useful public query matches and disclosing undesirable private query matches. We term this flexible formulation **soft-constraint**. This soft-constraint problem variant is especially appealing in applications where the user has multiple preferences. For example, in a hospital environment as described in Section 4.1, the administrator cares about many different aspects of the hospital, including controlling in-hospital infections, regulating staff's activities, as well as preserving patients' privacy, and so on. In this scenario, the administrator wants a solution that gauges the benefit of reporting a public pattern match and the penalty of potential privacy leaks. Our soft-constraint variant meets this need by ensuring that a pattern match will be output only when its benefit outweighs its penalty in expectation, otherwise, it is suppressed.

Therefore, the soft-constraint variant will be the focus of this paper. Supporting the

more general soft-constraint variant is a key contribution of our work, as the prior related literature [52] only studied the hard-constraint variant in theoretical properties.

4.3.2 Type-level vs. Instance-level

Mirroring the classical taxonomy proposed for relational data privacy [59], our problem can be classified into "type-level" (corresponding to the "global recoding" in relational k-anonymity), or "instance-level" (corresponding to "local recoding").

More specifically, the **type-level** problem makes simplified suppression decisions at the *event type* level by either suppressing or preserving all events of the same type. That is, events of the same type are either all suppressed or all preserved, irrespective of when they occur in the stream.

On the other hand, the **instance-level** problem treats each event differently based on its run-time context, i.e., the previously arrived events in the active windows of queries, and the expectation of future events. An instance-level solution can suppress one event of a certain type while preserving another of the same type. It thus allows flexible event suppression decisions that offer more opportunities for utility optimization, but as a consequence it presents a harder optimization problem.

4.3.3 Offline vs. Online

Orthogonal to the two dimensions above, our problem can be further classified into offline and online variants.

The **offline** event suppression produces decisions after the whole stream has arrived. Although this is not a practical assumption, studying the offline variant does allow us to eliminate the hardness arising from the randomness of the online problem and focus on optimal event suppression in a deterministic setting.

In the **online** event suppression, decisions have to be made in real-time for the currently arriving event without complete knowledge of future events. While this problem variant is more fitting for CEP applications, which typically demand real-time responsiveness, it is also intuitively more difficult. Challenges arise because (1) We must estimate future events and pattern matches that are probabilistic in nature, and (2) Even if future estimates are accurate, smart suppression decisions are still needed to maximize utility, which is in essence similar to the offline variant.

In summary, we will tackle the online event suppression problem with either hard or soft-constraint for both type-level and instance-level solutions.

4.4 Hardness Results

In this section, we study the hardness of the utility-maximizing event suppression problem. This sheds light on designing practical algorithms to solve the problem (Section 4.5). The proofs of our theorems are presented in Appendix. We first show in Theorem 4.1 that the instance-level offline variant is NP-hard.

Theorem 4.1 *The problem of instance-level utility-maximizing event suppression is NP-hard in the total number of event instances. It remains NP-hard even if each query contains exactly two event types.*

Proof. We reduce the *Maximum Weighted Edge Biclique (MWEB)* problem [87] to offline instance-level event suppression problem with soft-constraint. In the decision version of MWEB, given a bipartite graph $G = (V_1, V_2, H)$ where edges take on both positive and negative weights $W(h) \in \mathbb{R}, h \in H$, the problem is to determine if there exists a bipartite subgraph whose sum of edge weights is no less than a given number U. For each vertex $v \in V_1 \cup V_2$, construct an event type E_v and one event instance e_v of type E_v. The input

event sequence S is a randomly ordered sequence that consists of one instance e_v for all $v \in V_1 \cup V_2$. For each edge $h = (u, v) \in H$, if the edge weight is positive $W(h) > 0$, we build two public queries $Q_h = SEQ(E_u, E_v)$ and $Q'_h = SEQ(E_v, E_u)$, both with utility weight $W(h)$, and infinite window size. Similarly if the edge weight is negative, we build two private queries $P_h = SEQ(E_u, E_v)$ and $P'_h = SEQ(E_v, E_u)$ also with utility weight $W(h)$ and infinite window size. We first show that if there is a solution to MWEB with edge weight no less than U, then there exists a solution to the event suppression problem that has utility no less than U. Let $G' = \{V'_1, V'_2, H'\}$ be the subgraph of G that has weight C, where $C \geq U$. We can preserve every event instance e_v if $v \in V'_1 \cup V'_2$. The set of query matches would correspond directly to the set of edges H' in G', thus producing a total utility of C. Since we know $C \geq U$, this proves the forward direction. Now we need to show that if there is a solution to the event suppression problem with utility at least U, there is a bi-clique in the original graph that has an edge weight at least U. Let $T = e_v$ be the set of event instances preserved in event suppression solution. The subgraph G_T of G induced by $V = \{v : e_v \in T\}$ has a one-to-one correspondence between an edge in G_T and a query match in the solution T. The edge weight of G_T is thus the same as the utility of the solution T, which is no less than U. This completes our proof. □

Moreover, we show that the instance-level utility-maximizing event suppression problem is unlikely to be approximable in polynomial time.

Theorem 4.2 *Let n be the total number of event instances in the stream. There is a fixed constant $\epsilon \geq 0$ such that if there is n^ϵ factor approximation algorithm for instance-level utility-maximizing event suppression problem, then RP = NP.*

Proof. It was shown in [87] that there exists a constant $\epsilon > 0$, such that unless RP = NP, the MWEB problem cannot be approximated within a factor of n^ϵ in polynomial time, where n is the number of vertices in the graph. We note that the reduction from MWEB

above is value preserving. That is, if a MWEB problem instance has a weight value of U, then the corresponding event suppression problem we construct also has a utility value of U. We show that the soft-constraint variant cannot be approximated within n^ϵ by contradiction. Suppose there exists an polynomial time algorithm that approximates the offline variant within a factor of n^ϵ. Then we would have found an algorithm that approximates MWEB within n^ϵ. This contradicts with the inapproximability result in [87] under standard complexity assumptions, thus proves the inapproximability of the offline soft-constraint variant of the event suppression problem. □

The hardness and inapproximability results illustrate the unfortunate fact that unless we are dealing with streams that have a small number of events, utility-maximizing event suppression at instance-level is unlikely to be even approximated efficiently. The problem is not much simpler even if we focus on very simple query constructs (e.g., two event types per query). Given that stream systems typically need to handle potentially a large number of event instances in real-time, efficient optimal solutions with a good quality guarantee are unlikely to exist.

Even for the less ambitious type-level variant, we now show the surprising result that this problem exhibits a similar hardness result as follows.

Theorem 4.3 *The problem of type-level utility-maximizing event suppression problem is NP-hard in the total number of event types.*

Proof. We show this problem variant is NP-hard using a proof similar to the proof in Theorem 4.1, which is for the offline instance-level variant. In proving the hardness of offline instance-level variant, each problem instance we construct for each instance of MWEB problem has exactly one event instance per event type. In this particular case, an instance-level solution is also a type-level solution, and vice versa. Thus, using a similar proof, we can also show the hardness of the offline type-level variant. □

Despite the hardness result, we show in the following *fixed-parameter-tractable* special case that under some natural assumptions, the type-level event suppression problem can be solved optimally.

Proposition 4.1 *Suppose the expected number of matches for each query $Q \in \mathcal{Q}$ and $P \in \mathcal{P}$ over some standard time unit is known. Suppose the total number of public and private queries, $|\mathcal{P}| + |\mathcal{Q}|$, is some fixed constant. Then the problem of utility-maximizing online type-level event suppression can be solved in polynomial time to obtain the optimal solution in expectation.*

This proposition follows from a constructive algorithm to be described in Section 4.5.1. We would like to point out here that in comparison to the offline type-level variant, in the online version the solution is only optimal *in expectation*. The reason is because in the online version, only expectations of query matches are known and events can still arrive in an uncertain manner that deviates from the expectation. As such, one type-level solution may not be the optimal for all event stream instances. Rather it is optimal in the sense that in the long term, when the number of query matches converge to the expectation, the solution is optimal in expectation.

Given that the instance-level variant is intractable while the type-level problem can be optimally solved in theory, we next propose to first tackle the type-level variant (in Section 4.5.1), and then develop the *Hybrid* solution at instance-level by combining the type-level solution with local optimization heuristics (Section 4.5.2).

4.5 Online Suppression Algorithms

In this section, we devise two real-time event suppression strategies that maximize the overall utility. The first algorithm offers optimal suppression decisions at the event-type

level. Due to the fluctuations in the event stream, the optimal type-level decisions may not be the best decision for each event instance. This leads us to design the instance-level algorithm that tweaks type-level decisions based on run-time context, henceforth called *Hybrid* algorithm. The key idea is that while the type-level approach provides a pretty good long-term decisions, the instance-level decisions can *fine-tune* the type-level solution based on the events in the local windows.

Both of our algorithms need to estimate the cardinality of pattern matches as part of their decision making processes. Therefore, we also propose two advanced *cardinality estimation* approaches. The first light-weight estimation approach treats each type of event independent of each other, while the second sophisticated approach takes advantage of periodicities in the stream to produce even more accurate estimations.

4.5.1 Optimal Type-Level Algorithm

Inspired by Proposition 4.1, we first look at the type-level problem variance. We propose an optimal online solution to suppress events based on their types. This solution will be also used as a guidance for further optimizations in our instance-level algorithm presented in Section 4.5.2.

We propose to model the type-level suppression problem as an *integer linear program*. Specifically, let $\Sigma = \{E_i\}$ be the set of all event types. Let the integer $x_i \in \{0, 1\}$ be the decision variables to drop/keep all events of type E_i in order to ensure privacy, with $x_i = 1$ denoting to keep E_i, and $x_i = 0$ to drop E_i. Further let the integer $y_j \in \{0, 1\}$ denote whether or not public pattern $Q_j \in \mathcal{Q}$ will be output, and $z_k \in \{0, 1\}$ denote whether private pattern $P_k \in \mathcal{P}$ will be output.

Let $N_T(Q_j)$ and $N_T(P_k)$ be the expected number of pattern matches for Q_j and P_k produced over a standard time period T respectively. Using historical data, we can obtain statistics about the stream, like the average event arrival rate. These statistics then allow

us to get a rough estimate of $N_T(Q_j)$ and $N_T(P_k)$ (we defer a detailed discussion on the orthogonal issue of cardinality estimation to Section 4.6). Now assume that $N_T(Q_j)$ and $N_T(P_k)$ are computed and treated as known constant values. The objective utility function then becomes:

$$\mathcal{U} = \sum_{Q_j \in \mathcal{Q}} w(Q_j) N_T(Q_j) y_j + \sum_{P_k \in \mathcal{P}} w(P_k) N_T(P_k) z_k \tag{4.5}$$

Recall that $w(\cdot)$ represents the utility weight function. We note that whether or not query Q_j can be reported (y_j is 0 or 1) depends on the values of the x_i's. Let $\sigma(Q_j)$ be the multi-set of event types in Q_j. If one constituent event type E_i of a query pattern Q_j is dropped ($x_i = 0$, for $E_i \in \sigma(Q_j)$), then the query pattern can never be revealed ($y_j = 0$). We express the dependence using the following linear constraint. For all $Q_j \in \mathcal{Q}$

$$0 \leq y_j \leq \frac{1}{|Q_j|} \sum_{E_i \in \sigma(Q_j)} x_i \tag{4.6}$$

Intuitively, this says that in a type-level solution where events are suppressed at event type granularity, Q_j can be reported ($y_j = 1$) if and only if none of its participating event types are dropped.

Similarly, a variable $z_k \in \{0, 1\}$ is subject to the constraint:

$$1 \geq z_k \geq \frac{1}{|P_k|} - 1 + \frac{1}{|P_k|} \sum_{E_i \in \sigma(P_k)} x_i \tag{4.7}$$

This captures the fact that the private pattern will be reported ($z_k = 1$) when all its participating event types are preserved. When at least one constituent event type is absent, z_k can get a value of 0, which would avoid the utility penalty.

The problem of maximizing $\mathcal{U}$ is then an integer linear programming problem subject to the constraints in Equations (4.6) and (4.7). Putting the issue of estimating $N_T(Q_j)$ and

$N_T(P_k)$ aside, this online type-level variance is really no different from offline type-level variance. From Proposition 4.1, we know that it is solvable if the total number of event types or queries is limited.

This is useful, for although the number of event instances may be unbounded in a stream system, the number of queries or event types tends to be limited (System-H, for example, has 6 event types and 14 queries and the corresponding LP can be solved in 20 ms). Furthermore, even when we are dealing with problems that have a large number of queries and events, since the type-level decisions would stay the same for every new arriving event, the LP only needs to be solved once for a long period of time (until new statistics arrive, at which point the decisions need to be re-computed). This approach thus provides a practical way to solve the type-level event suppression problem. We use Example 4.1 to illustrate this algorithm.

Example 4.1 *Suppose we have five event types, $\Sigma = \{A, B, C, D, E\}$. A sample event stream is illustrated in Figure 4.3a. The superscript of an event denotes its time-stamp. As can be seen from the figure, events with different event types arrive with the same arrival rate, namely 1 event instance per 10 time units for all event types, in a simple and recurring pattern (a, b, c, d, e).*

Assume that there are three public queries, namely

$Q_1 = SEQ(B, C, D)$, $Window(Q_1) = 10$, $w(Q_1) = 5$;

$Q_2 = SEQ(A, B)$, $Window(Q_2) = 10$, $w(Q_2) = 20$;

$Q_3 = SEQ(D, E)$, $Window(Q_3) = 10$, $w(Q_3) = 20$;

and one private pattern,

$P_1 = SEQ(A, C, E)$, $Window(P_1) = 10$, $w(P_1) = -10$;

So as an example, the query Q_1 looks for pattern (B, C, D) that appears within window of 10 time units. In Figure 4.3a, (b^2, C^4, d^6) is an instance of match for Q_1, and this match produces an utility gain of 5.

Let $[x_1, x_2, ..x_5]$ be the decision variables of whether events of type A, B, C, D, E can be preserved, $[y_1, y_2, y_3]$ be the variables that indicate whether Q_1, Q_2, Q_3 can be reported, and $[z_1]$ be the variable indicating whether P_1 will be produced. Using Equations (4.5), (4.6) *and* (4.7)*, we obtain a linear program with the objective function:*

$$U = \frac{1}{10} \cdot 5 \cdot y_1 + \frac{1}{10} \cdot 20 \cdot y_2 + \frac{1}{10} \cdot 20 \cdot y_3 - \frac{1}{10} \cdot 10 \cdot z_1$$

where the four $\frac{1}{10}$s are essentially $N_T(Q_j)$ and $N_T(P_k)$, i.e., the expected number of pattern matches over a time period T. Since there is 1 match over 10 time units for all four patterns, the values are all $\frac{1}{10}$.

The utility maximization is subject to constraints:

$$0 \leq y_1 \leq \frac{1}{3}(x_2 + x_3 + x_4)$$

$$0 \leq y_2 \leq \frac{1}{2}(x_1 + x_2)$$

$$0 \leq y_3 \leq \frac{1}{2}(x_4 + x_5)$$

$$1 \geq z_1 \geq \frac{1}{3}(x_1 + x_3 + x_5) - \frac{2}{3}$$

Solving the linear program gives us the optimal type-level solution, $[x_1, x_2, ...x_5] = [1, 1, 0, 1, 1]$. Namely, events of type C will all be suppressed. This solution is intuitive. Dropping event type C ensures that no matches for P_1 and Q_1 will be produced. Since the arrival rate of Q_1 and P_1 are both $\frac{1}{10}$, and the utility weight $w(Q_1) = 5$ while $w(P_1) = -10$, it is profitable to drop event type C.

Once the type-level decisions are computed, at execution time, all events of type C will simply be suppressed, resulting in the event stream in Figure 4.3b.

Discussion of LP-based algorithm. While this LP-based algorithm is practical for

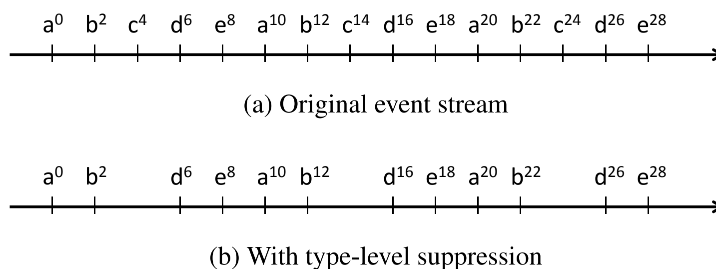


Figure 4.3: Type-level LP-based algorithm

the type-level problem variance, it would be *impractical* for the instance-level variance. Recall that the instance-level variance makes suppression decisions at the granularity of event instances. Hence for an instance-level solution, a decision variable x_i is needed for *each event instance*, and each *pattern match* will be represented as a constraint. Note that both of these two can quickly become impractically large, resulting in too big a linear program to be solved efficiently. Hence, in the next section we propose a hybrid approach that combines the type-level solution with instance-level heuristics.

4.5.2 Hybrid Instance-Level Solution

We now explore solutions to online instance-level event suppression, which can produce better utility than a type-level solution for the following reason. Recall that in the type-level solution, the expected number of pattern matches, $N_T(Q_j)$, is an average statistic produced over a long term. It is possible that over a short period of time event frequencies may deviate from their long term averages. Therefore, if the type-level solution is used as an instance-level solution, it may be sub-optimal in the short-term local windows. This observation is illustrated in the following example.

Example 4.2 *We revisit the event stream in Figure 4.3 while considering the same query patterns. Recall that in Example 4.1, the type-level solution is to suppress event type C based on the long term event arrival expectation.*

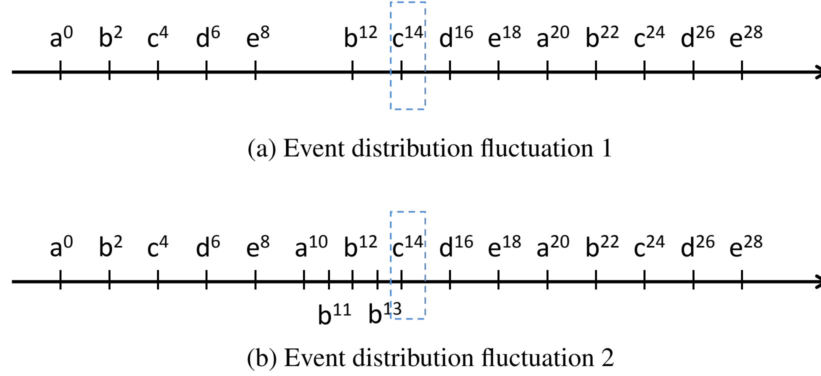
In Figure 4.4a, however, no a^{10} *arrives as expected, which is different from what the long-term statistics predict. Then when* c^{14} *arrives, there is no event of type* A *prior to* c^{14} *within* $Window(P_1) = 10$*. Hence, at time 14, no match for* $P_1 = SEQ(A, C, E)$ *can be produced even if* c^{14} *is kept. In that case, keeping* c^{14} *is essentially "free" (without any utility penalty). An optimal solution should keep* c^{14} *since it bears the potential of producing a match for* $Q_1 = SEQ(B, C, D)$*. The type-level solution in this example fails to make this optimal decision.*

The reasoning in the type-level solution, however, is not without merit. Over the long term, we expect one event of type A *prior to an event of type* C *within* $Window(P_1)$ *on average, which leads to one match of* P_1*. We also expect one match of* Q_1 *in which the same event of type* C *participates. So the type-level solution decides to suppress the event of type* C*, because the expected utility penalty of keeping it is* $w(P_1) = -10$*, which outweighs the expected utility gain* $w(Q_1) = 5$*.*

The particular case of c^{14} *is different because there is no previous event of type* A *in* c^{14}*'s active window. This deviation from the long term statistics renders the type-level solution sub-optimal.*

As another example, in Figure 4.4b, there are many more events of type B *prior to* c^{14} *than expected, namely* b^{11}*,* b^{12} *and* b^{13}*. In this case, at time 14 it may be advisable to keep* c^{14} *because there are three potential matches for* Q_1*. This will result in a utility gain of* $3 \times w(Q_1) = 15$*, which outweighs the potential utility penalty of one match of* P_1*, i.e.,* $w(P_1) = -10$*.*

This example illustrates that due to fluctuations in the event stream, the optimal type-level solution may *not* be the best decision. Instead it would be beneficial to deviate from the type-level decisions. This leads us to design an enhanced algorithm that extends type-level solution by instance-level heuristics that *"tweaks" type-level decisions based on run-time context*. The intuition is that while the type-level approach provides a "pretty



(a) Event distribution fluctuation 1

(b) Event distribution fluctuation 2

Figure 4.4: Motivation: suppressing c^{14} can be sub-optimal

Algorithm 3 A Hybrid Algorithm for Online Instance-Level Suppression

```
Suppress_Hybrid_Instance_Level (history):
Suppress_T[] = Solve_LP(history)
while new_event arrives do
  utl_gain = 0
  for each partial_match of q that new_event participates do
    exp_match ← Est_Match (partial_match, Suppress_T[])
    utl_gain += exp_match * w(q)
  end for
  utl_penalty = 0
  for each partial_match of p that new_event participates do
    exp_match ← Est_Match (partial_match, Suppress_T[])
    utl_penalty += exp_match * w(p)
  end for
  if utl_gain + utl_penalty < 0 then
    suppress new_event
  else
    preserve new_event
  end if
end while
```

good" overall solution, the local decisions can mostly stick to the type-level solution but fine-tune based on the fluctuations of events in the local windows. We describe our Hybrid algorithm in Algorithm 3.

The overall flow of Algorithm 3 is summarized below. First it computes the expected utility gain of keeping the newly arrived event, e_i, by estimating the expected number of public query matches e_i could participate in. Then it computes the expected utility penalty

if e_i is kept by estimating the expected number of private pattern matches e_i could be part of. It then decides to either suppress or keep e_i by comparing the expected utility gain and penalty.

We now describe Algorithm 3 in detail. First, the type-level decision for each event type is computed once in **Solve_LP**. Then, as each new event e_i arrives, we look up those events that have previously arrived for *partial matches* that e_i can participate in. Since each such partial match is only a *"prefix"* of a full match, we then estimate in the subroutine **Est_Match** the number of matches for the *"suffix"* pattern from events that are expected to arrive in the future.

To produce an accurate estimate, two challenges arise: (1) to estimate the expected number of future matches of the "suffix" pattern, and (2) to estimate the event suppression decisions for the future events. We defer the first issue to Section 4.6. The second issue is also hard, because the suppression decision of a future event could affect the decisions about other future events thereafter. Our intuition, however, is that the type-level solution should provide guidance of what events tend to be suppressed in general. After all, instance-level decisions deviate from the type-level solution only when the local event distribution differs substantially from the long term average. It can be expected that suppression decisions should in general converge to the type-level solution. Thus we estimate the number of matches for the "suffix" of the partial match by making the assumption that events in the future will be suppressed in the same manner that the type-level solution dictates.

With that estimate in **Est_Match** we can calculate an expected utility gain for each partial match. Summing all the utility expectations gives us an estimation of the benefit of keeping the newly arrived event. In the next step, we can do a similar estimation of the total utility penalty that we suffer if the new event is kept. In the end, by a simple comparison of total utility gain and utility penalty we can determine, based on the events

that have arrived, whether to suppress the newly arrived event or to keep it. We use the running example in Example 4.3 to illustrate Algorithm 3.

Example 4.3 *In Figure 4.5, we revisit the event stream in Figure 4.4a of Example 4.2 while considering the same patterns. In Figure 4.5, event a^0 arrives first, which participates in $Q_2 = SEQ(A, B)$. According to the type-level decision, we expect events of type B will be kept in the future, with which a^0 is likely to contribute to matches of Q_2. Then the utility gain of keeping a^0 is expected to be $w(Q_2) = 20$. On the other hand, a^0 also forms a partial match for $P_1 = (A, C, E)$. Suppose in the **Est_Match** procedure one match for the remaining suffix (C, E) is expected to arrive. However we reason that events of type C will most likely be suppressed, according to the type-level solution. As a result, the expected utility penalty of keeping a^0 is 0. Hence a^0 is preserved.*

Next arrives b^2, which participates in Q_1 and Q_2 but does not participate in any private pattern. Hence keeping b^2 will not incur any utility penalty. Without further estimation, we keep b^2.

*Now consider c^4. It participates in both $Q_1 = SEQ(B, C, D)$ and $P_1 = SEQ(A, C, E)$. Suppose the **Est_Match** procedure tells that for the partial match (b^2, c^4) of Q_1, one event of type D is expected in the remaining window. Accordingly, the utility gain of keeping c^4 would be $w(Q_1) = 5$. Similarly for P_1, suppose **Est_Match** expects that in the remaining window of the partial match (a^0, c^4), one event of type E will come. That would lead to a utility penalty of $w(P_1) = -10$ if c^4 is kept. Hence c^4 should be suppressed.*

*Following the same logic, d^6, e^8 and b^{12} will be preserved. So far all the suppression decisions have been the same as the type-level solution. Next we consider c^{14}. First, for the partial match (b^{12}, c^{14}) of Q_1 that it participates in, suppose **Est_Match** again predicts that one event of type D will arrive in the remaining window, which amounts to a utility gain of 5. Then for P_1, in which c^{14} could participate, we find that from time 4 to 14, no event of type A exists, namely no partial match of P_1 that c^{14} could join with. Hence,*

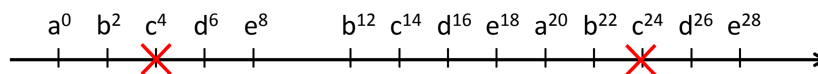


Figure 4.5: Running example of hybrid algorithm

keeping c^{14} will produce a net utility gain with no utility penalty. In this case we will keep it, which is a different decision from the type-level solution.

The preservation of event c^{14} by our algorithm matches our intuition — it should not be suppressed in this particular case, due to the event fluctuations in the local window. This example manifests how the Hybrid algorithm can exploit local optimization opportunities to maximize utility.

We skip the explanation of the decisions on remaining events. The upshot is that decisions about remaining events converge to the type-level solutions (keeping all but event C). This is an intuitive justification for our simplifying assumption that events which arrive in the future will most likely be suppressed as predicted by the type-level solution.

4.6 Pattern Match Cardinality Estimation

So far we have treated the computation of $N_T(Q_j)$ in our type-level solution and the cardinality estimation of **Est_Match** in the Hybrid solution as black boxes. Recall that the reason we need an estimate using $N_T(Q_j)$ in the type-level solution is to measure the expected utility Q_j yields on average. The need of **Est_Match** is to weigh the utility gain and penalty of keeping the current event based on existing events in the active windows. The key issues behind both operations are essentially the same, which is to estimate the number of pattern matches over a time range T in the future.

As a matter of fact, in many applications the continuously arriving events tend to ex-

hibit certain regular patterns. If history is of any guidance for the future, using statistics from historical data would allow us to make an educated guess of the pattern match cardinality in the future. The problem then is to use historical data to estimate the cardinality of future pattern matches. Formally, given a pattern query $Q_j = SEQ(E_{j_1}, E_{j_2}, ..)$, and a time variable T, we want to estimate, using event arrival statistics, the number of matches for Q_j produced within T, henceforth referred to as $N_T(Q_j)$.

The time range T is a variable because it depends on how far apart in time the first event and the last event in the partial match are. For example, in Figure 4.5 of Example 4.3, when c^4 arrives, we need to estimate the number of events of type D that may arrive, which along with the existing partial match (b^2, c^4), can form full matches for query $Q_1 = SEQ(B, C, D)$. Because $Window(Q_1) = 10$, and the first event in the partial match is b^2, at current time 4, events of type D that arrive in the next $10 - 4 + 2 = 8$ time range can contribute to this particular match. This, then, is the value of T that we need to compute. However if the first event in the partial match is, say b^1 instead of b^2, then the window over which event of type D should be estimated is $10-4+1 = 7$, instead of 8. In other words the window of the partial suffix match is not fixed but varies, which makes the problem of cardinality estimation challenging (if the time window for cardinality estimation is always some fixed value then the problem can be solved by simply counting the historical data, with no computation involved).

In general, in order to estimate the number of matches in **Est_Match** in the Hybrid solution, we can compute the corresponding value using the function $N_T(Q_j)$. Let $e_f.ts$ be the timestamp of the first event in the partial match, $e_c.ts$ be the timestamp of the current event under consideration, and Q_S be the suffix pattern that remains to be matched. Then estimates can be expressed as $N_{(Window(Q_j)-e_c.ts+e_f.ts)}(Q_S)$.

It is clear that accurate pattern match cardinality estimation, or the computation of $N_T(Q_j)$, provides an important basis for our suppression solutions. Next we devise two

approaches for pattern match cardinality estimation.

4.6.1 Estimation by Arrival Rate

Our first estimation approach assumes that the event arrivals follow a certain statistical model, namely the Poisson process. The Poisson process is a common model used for data arrival in queuing theory in the performance modeling literature [58], as well as in the stream processing literature [18, 55]. In the context of CEP, the Poisson process dictates that each type of event occurs continuously and *independently of each other*. Each type of event, E_i, arrives with an arrival rate λ_i, and the number of events that arrive in a fixed time period follows a Poisson distribution. In practice, when events arrive in a Poisson process, λ_i can be estimated by sampling the arriving events. Considering the fact that events may follow different distributions as time evolves, a moving sample of recently arrived events can be used to estimate the arrival rate [36]. This should give us a good estimate of λ_i.

Given the arrival rate λ_i for event type E_i, we describe how $N_T(Q_j)$ can be estimated. We first compute for each event type in Q_j the expected number of event occurrences in time range T, denoted by l_i. This can be computed as $l_i = \lambda_i T$. Further we denote by $\Gamma(Q_j)$ the multi-set of event types in Q_j, $\sigma(Q_j)$ the set of event types in Q_j, and $|Q_j|$ the pattern length of Q_j (for example, query $Q = SEQ(A, A, B)$ has $\Gamma(Q) = \{A, A, B\}$, $\sigma(Q) = \{A, B\}$, and $|Q| = 3$). Denote by $L_j = \sum_{E_i \in \sigma(Q_j)} l_i$ the expected number of occurrences of events relevant to Q_j in T. We can then estimate the expected number of query matches for Q_j in T as:

$$N_T(Q_j) = \binom{L_j}{|Q_j|} \prod_{E_i \in \Gamma(Q_j)} \frac{\lambda_i}{\sum_{E_k \in \sigma(Q_j)} \lambda_k} \tag{4.8}$$

Equation (4.8) can be explained as follows. Given a total of L_j event occurrences in

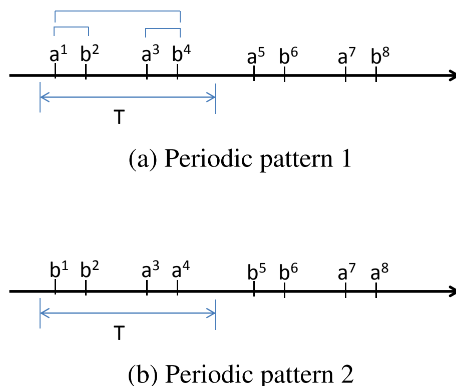
T, we need to pick $|Q_j|$ events to form one query match. Let us pick the first $|Q_j|$ events among the L_j events, and compute the probability that the first $|Q_j|$ events produce a match. Let the event at the first position of Q_j be of type E_{i_1}. The probability that the first event picked is actually of type E_{i_1} is $\frac{\lambda_{i_1}}{\sum_{E_k \in \sigma(Q_j)} \lambda_k}$, which is E_{i_1}'s arrival rate λ_{i_1} divided by the sum of arrival rates of all events in Q_j. For the second event in the sequence, the probability that it matches the event type at the second position of Q_j can be computed similarly. In the end the probability that the first $|Q_j|$ events all match the event types specified in Q_j and thus produces a query match is the cross product of individual terms, $\prod_{E_i \in \Gamma(Q_j)} \frac{\lambda_i}{\sum_{E_k \in \sigma(Q_j)} \lambda_k}$. Given that there are a total of $\binom{L_j}{|Q_j|}$ possible permutations out of L_j events and by symmetricity, the expected count of query matches can be expressed as the product of the two, thus Equation (4.8).

4.6.2 Estimation by Periodicity

Estimation using the event arrival rate is generally applicable in the sense that it does not assume dependencies between events of interest. However, it is often the case that many event streams exhibit certain *periodic patterns* that are highly regular. For example, in the hospital setting, a nurse may sanitize and enter patient rooms at a regular time interval, say every hour, for routine patient check-ups. Such patterns, being periodic and regular, can be leveraged to produce even more accurate pattern match cardinality estimation. We use Example 4.4 to illustrate this observation.

Example 4.4 *Suppose we have two types of events, A and B, as in Figure 4.6. From their respective arrival rates we know there will be two instances of A and B in a time period* T*. Knowing only this information, the best we can do is to use Equation* (4.8) *to produce an average estimate of the number of matches for pattern SEQ(A, B).*

If we can ascertain that events of type A and B follow certain periodic patterns, then we can do a significantly better job. Suppose A and B have the periodic pattern "A, B,



(a) Periodic pattern 1

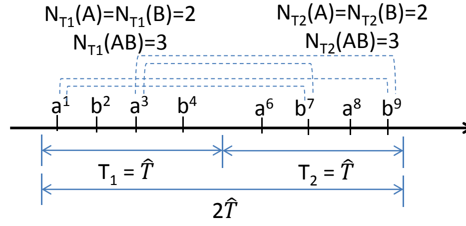
(b) Periodic pattern 2

Figure 4.6: Estimating matches of SEQ(A,B) using periodic pattern

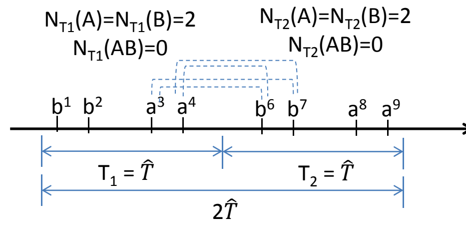
A, B" as illustrated in Figure 4.6a. Then starting at 0 *over a period of time* T *there are a total of three matches, namely* $\{(a^1, b^2), (a^1, b^4), (a^3, b^4)\}$. *On the other hand, if the periodic pattern is of the form "B, B, A, A" as shown in Figure 4.6b, then even if the arrival rate of A and B stays the same over* T*, no match for SEQ(A, B) can be produced.*

This illustrates the importance of discovering periodic patterns embedded in the event data, for if such patterns can be revealed and utilized, a much better pattern match estimate can be produced. The natural problem then is to discover such periodic patterns in a given stream. This problem of periodicity mining has been extensively studied. In this work we adapt the techniques in [43, 67] to discover the periodic patterns.

Once such periodic patterns are discovered, we can compute the number of matches for pattern Q_j in each periodic cycle $\hat{T}$, denoted by $N_{\hat{T}}(Q_j)$. The problem that remains to be solved is to extrapolate the expected pattern match cardinality of an arbitrarily long time period T_X. More specifically, the problem can be stated as given a periodic cycle $\hat{T}$ and $N_{\hat{T}}(Q_j)$, what is the number of pattern matches of Q_j in time period T_X, or $N_{T_X}(Q_j)$? Care must be taken in calculating $N_{T_X}(Q_j)$. While it is tempting to think that if $T_X = 2\hat{T}$, then $N_{2\hat{T}}(Q_j)$ can be simply computed as $N_{2\hat{T}}(Q_j) = 2N_{\hat{T}}(Q_j)$, the analysis presented in Example 4.5 shows the error in this thinking.



(a) Periodic pattern 1



(b) Periodic pattern 2

Figure 4.7: Estimating via periodic pattern for time $2\hat{T}$

Example 4.5 *We revisit Example 4.4 in Figure 4.7. Given the periodic pattern "A, B, A, B" of stream in Figure 4.7a, and supposing we know that in the periodicity $\hat{T}$, there are two instances of A and B, respectively, denoted by $N_{\hat{T}}(A) = N_{\hat{T}}(B) = 2$, and three matches of pattern SEQ(A,B), or $N_{\hat{T}}(AB) = 3$, what is the number of pattern matches for SEQ(A, B) in $2\hat{T}$? Note that $N_{2\hat{T}}(AB) = 2N_{\hat{T}}(AB) = 6$ is problematic, because when the time period is extended from $\hat{T}$ to $2\hat{T}$, As in T_1 and Bs in T_2 also produce matches in a cross-product manner. In this particular case, $N_{2\hat{T}}(AB) = N_{T_1}(AB) + N_{T_2}(AB) + N_{T_1}(A)N_{T_2}(B) = 3 + 3 + 2 * 2 = 10$. This can be verified by enumerating the number of SEQ(A,B) matches in $2\hat{T}$. Similarly, in the second periodic pattern in Figure 4.7b, the number of pattern matches $N_{2\hat{T}}(AB)$ follows the same formula: $N_{2\hat{T}}(AB) = N_{T_1}(AB) + N_{T_2}(AB) + N_{T_1}(A)N_{T_2}(B) = 0 + 0 + 2 * 2 = 4$, which correspond to $\{(a^3, b^5), (a^3, b^6), (a^4, b^5), (a^4, b^6)\}$.*

In general, to compute $N_{n\hat{T}}(Q_j)$ given $N_{\hat{T}}(Q_j)$, where $n \in \mathbb{Z}^+$, Equation (4.9) can be used following the logic derived in Example 4.5. First, define a *continuous i-segmentation*

of Q_j as a segmentation that breaks the event sequence of Q_j into i non-empty segments (for example, 2-segmentations of $Q = SEQ(A, B, C, D)$ include (A/BCD), (AB/CD), and (ABC/D)). Each segment in an i-segmentation is referred to as a *chunk* (for example, A and BCD are two chunks in the segmentation (A/BCD)). Denote by $M_i(Q_j)$ all continuous i-segmentations of Q_j, and $m = (c_1, c_2, ...c_i)$ be one such segmentation in $M_i(Q_j)$, where c_k is the k-th chunk in m. Let $D(m) = \prod_{c_k \in m} N_{\hat{T}}(c_k)$ be the cross-product of the count statistics of chunk c_k. Then the estimation of $N_{n\hat{T}}(Q_j)$ can be written as:

$$N_{n\hat{T}}(Q_j) = \sum_{i=1}^{n} \binom{n}{i} \sum_{m \in M_i(Q_j)} D(m) = \sum_{i=1}^{n} \binom{n}{i} \sum_{m \in M_i(Q)} \prod_{c_k \in m} N_{\hat{T}}(c_k) \tag{4.9}$$

So far we have only considered estimating pattern matches in a time period that is an exact multiple of a periodic cycle $\hat{T}$. Next we extend the logic to the general case. To estimate pattern match cardinality of a time period that is less than a full periodic cycle, we can detect the number of pattern matches within that time period from the known periodic pattern. To handle a time period that is longer than $\hat{T}$ but not an exact multiple of $\hat{T}$, we first estimate the utility within the time of a multiple of $\hat{T}$ using Equation (4.9), and then detect the number of matches within the remaining time period using the periodic pattern. Finally we combine the two parts of estimation to obtain a final estimate.

We implemented the periodicity mining and the periodicity based cardinality estimation approach as an alternative to estimation using the Poisson arrival process. For event streams without such periodicities we will fall back to use the arrival rates (Section 4.6.1) to estimate pattern match cardinality. As we will see in our experiments in Section 5.6, event streams with stronger periodic patterns can significantly benefit from this periodicity based estimation.

4.7 Experimental Evaluation

4.7.1 Experimental Setup

We have implemented our proposed algorithms on the HP CHAOS stream engine [49], and used the OptimJ [4] module as the linear program solver. All experiments were conducted on a machine with an Intel Core 2 Duo 3.0GHz CPU and 3.2GB of RAM running Windows 7 and Java JRE 6.

Our experiments were conducted using both a real-world workload and a synthetic workload. The real-world event stream was collected from a hospital where the a real-time hygiene monitoring system is deployed [93]. Public query patterns were created in reference to the hand hygiene regulations established for US hospitals [25], while private queries were constructed in consultation with domain experts. The weights of pattern queries were assigned by the domain experts as well. The characteristics of real-world pattern queries are summarized in Table 4.1.

The synthetic event stream is produced with a parameterized *degree of periodicity*. Periodicity 1 corresponds to a perfectly periodic stream, whereas periodicity 0 means perfectly random. The synthetic event stream is produced as follows. First we generate a perfectly periodic stream, by randomly generating a short "seed" event sequence, which is then replicated over time in a periodic manner. The event stream so produced has degree of periodicity 1. On the other hand we generate another event stream that using a pure Poisson process. Because no intentional periodicity is embedded in its generation we denote its periodicity as 0. Between these two extremes we can generate a spectrum of event streams with varying degrees of periodicity. Specifically, we randomly sample $\gamma\%$ of events from the perfect periodic stream, and sample $(100 - \gamma\%)$ events from the non-periodic, Poisson stream. These two streams are then combined to produce a stream with $\gamma\%$ of periodicity. For example, in the stream with 0.8 degree of periodicity, 80% of

Parameter	Value range
Number of public patterns	10
Number of private patters	4
Pattern length	2 to 6
Time-based window size	30 seconds to 2 hours
Weight of public patterns	1 to 10
Weight of private patterns	-1 to -1000

Table 4.1: Pattern query characteristics

the events come from the perfectly periodic stream and the rest from the Poisson stream. Synthetic pattern queries are generated by randomly picking pattern lengths, event types and window sizes using value ranges of real-world queries in Table 4.1. Each synthetic experiment is reported using results averaged over 20 runs with randomly generated patterns.

4.7.2 Algorithms Compared

We compare algorithms proposed in this work, namely the type-level LP-based solution (labeled as "Type-lvl"), the instance-level Hybrid solution with independent Poisson arrival estimation, (labeled as "Hybrid-I"), and the Hybrid solution with periodicity estimation (labeled as "Hybrid-P"). We also implemented two alternative event suppression solutions, namely the "Hybrid optimal" algorithm and the "Greedy" algorithm, to compare against our proposed algorithms, in order to shed light on the performance of our algorithms.

Hybrid optimal. The first baseline approach is the so-called Hybrid-optimal. Recall that one key problem that we address in this work is to estimate the cardinality of pattern matches in the future. In this algorithm, we assume the Hybrid-optimal algorithm has perfect statistics, that is, it can magically know the exact number of pattern matches in the future. This can be implemented by looking at the events in the future and counting the number of matches. Comparing our approach with such an "oracle" algorithm that

essentially "cheats" helps us to understand the utility loss due to inaccurate match cardinality estimations. Experimental results for this perfect-statistics solution are labeled as "Hybrid-opt".

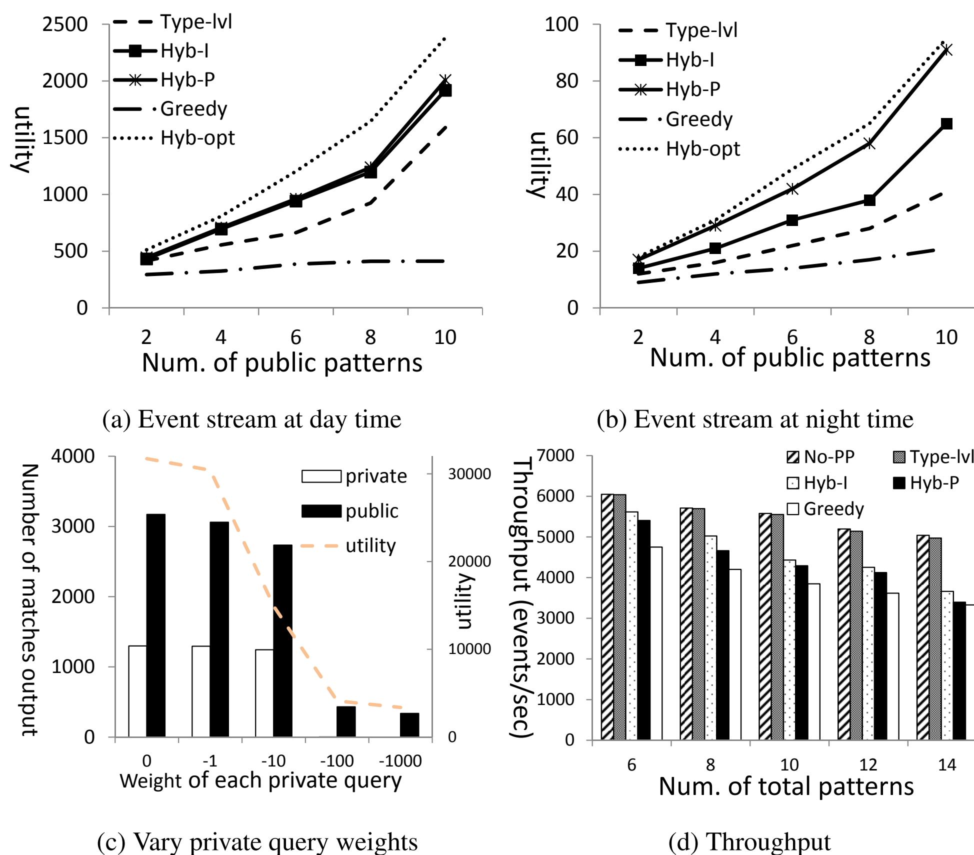
Although it is interesting to compare with an "offline-optimal" solution to see the utility deficit against the optimal instance-level solution, it turns out that solving the optimal offline problem at instance-level is so computationally expensive that we can only finish with 30 event instances in a reasonable amount of time, which is too small a number to produce any meaningful results (our data stream has at least thousands of event instances).

Greedy solution. In addition, we experimented with a greedy algorithm that detects and eliminates breaches in a *detecting-then-removing* fashion [96]. Specifically, for each event e_i, it calculates the utility gain of public query matches that e_i triggers, as well as the utility loss of private query matches that e_i triggers over the stream of events output for public query matches. Both utility gain and loss are calculated using currently detected matches *without* estimating the effect of e_i on future matches. The greedy approach would suppress e_i if the utility loss outweighs the utility gain, and keeps e_i otherwise. This approach will be labeled as "Greedy" in the experiments.

4.7.3 Experiments on Hospital Workload

We observe that events collected from the hospital during the day are usually dominated by random healthcare worker activities, which tend to be chaotic and aperiodic. However, events during the night are more regular with a higher degree of periodicity, due to the fact that at night time nurses on duty in ICU typically only conduct routine check-ups in every room at regular intervals. We thus separate the stream into two sub-streams, one with day-time events and the other with night-time events.

Num. of public patterns vs. utility. In the first experiment, we vary the number of public queries and keep private queries unchanged. Figure 4.8a shows the util-



(a) Event stream at day time

(b) Event stream at night time

(c) Vary private query weights

(d) Throughput

Figure 4.8: Performance on hospital workload

ity gain comparisons over the day-time stream. Clearly, Hybrid-opt (with perfect estimates of matches) has best utility gains. Our Hybrid-P approach (estimating with periodicity) comes second, slightly outperforming Hybrid-I (estimating with independent arrival rates). Both hybrid algorithms consistently outperform the Type-level algorithm by around 30%, and Hybrid-P outperforms the Greedy algorithm by up to 4x. This underlines the effectiveness of our Hybrid algorithms.

Figure 4.8b shows the utility comparisons over the night-time event stream. While the relative performance of the Hybrid-I and Hybrid-P is close in Figure 4.8a, in Figure 4.8b Hybrid-P produces 40% more utility than Hybrid-I. More impressively, it has only 10% less utility than the Hybrid-opt. This is encouraging, because it shows that if the event stream exhibits certain periodic patterns, then our periodicity based cardinality estimation

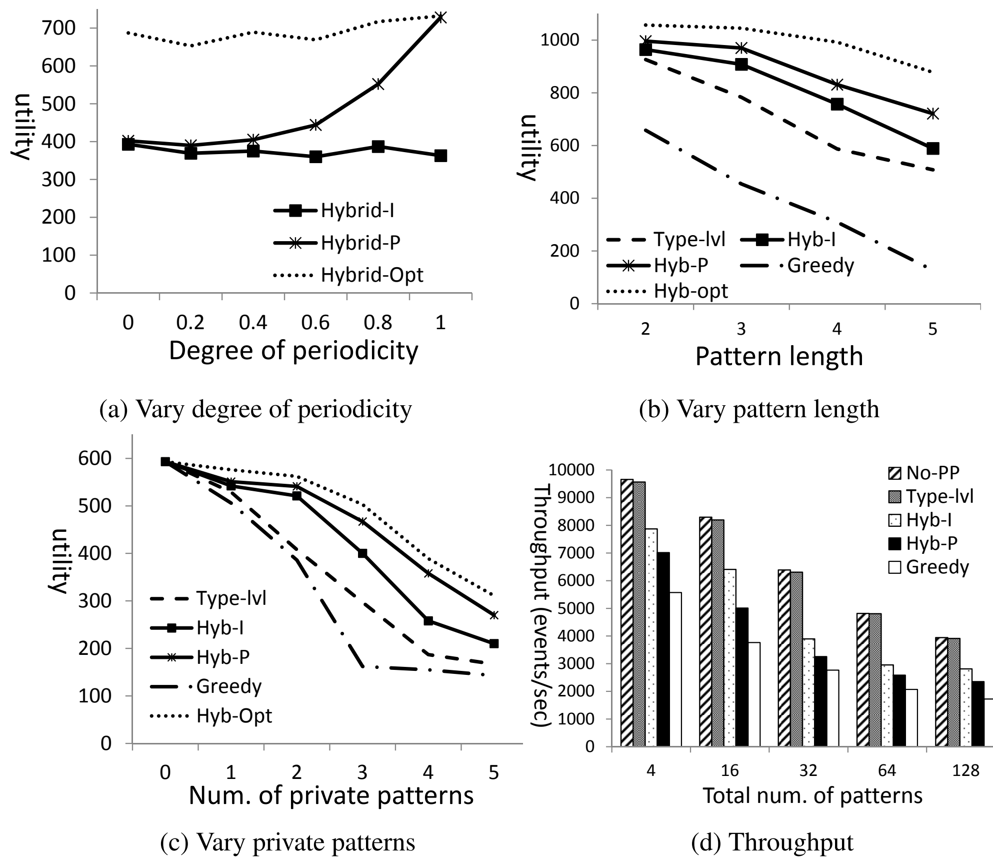
can capture these patterns and produce accurate estimations that ultimately lead to better utility-preserving decisions.

Pattern weight vs. num. of output. In this experiment we vary the weights of private patterns from 0 to -1000. Figure 4.8c shows the number of private matches vs. public matches produced by our Hybrid-P approach, as well as the corresponding utility. As the weights become more negative, the number of private matches output decreases. Observe that when the negative weight has significantly larger absolute value than the positive weight, no private match can be output at all. This also shows that our solutions can be configured with extremely negative weights to solve the hard-constraint problem.

In this experiment we compare the private and public pattern matches produced by the soft-constraint variant vs. hard-constraint variant. For the soft-constraint setting, we vary the weights of private patterns from 0 to -100. For the hard-constraint setting, we set the weights of private patterns to be -1000. Figure 4.8c shows the number of private matches vs. public matches produced by our Hybrid-P approach, as well as the corresponding utility. As the weights become more negative, the number of private matches output decreases. Observe that in the hard-constraint scenario (weight equals to -1000), no private match is output at all. This confirms that our approach can guarantee that there is no privacy leak in the hard-constraint setting.

Num. of patterns vs. throughput. Figure 4.8d depicts the throughputs, in terms of events/second, of the competing approaches. An overall message here is that all of our proposed approaches can successfully achieve near-real-time responsiveness, i.e., at least 3500 events/second on modest hardware. To put this in perspective, in our real-world hospital environment the event arrival rate is around 20 events/sec. The proposed approaches are thus efficient and more than sufficient to deal with real-world applications like HyReminder.

In terms of the performance of each approach, the Type-level solution has almost the



(a) Vary degree of periodicity

(b) Vary pattern length

(c) Vary private patterns

(d) Throughput

Figure 4.9: Performance on synthetic workload

same throughput as the baseline approach without any privacy preservation (denoted as "No-PP"). Because the linear program associated with HyReminder can be solved very efficiently (within 20 ms), and after that Type-level approach only follows the pre-computed policy at run-time with virtually no computational overhead. The Greedy approach has lowest throughput, as it produces private pattern matches – while other methods just estimate future matches. The Hybrid-P approach has slightly lower throughput than the Hybrid-I as it is more computationally expensive. Overall, the computation overhead of hybrid algorithms on this real workload is not significant, which makes them attractive especially considering their substantial utility benefits.

4.7.4 Experiments on Synthetic Workload

Periodicity degree vs. utility. Now we assess the relationship between degree of periodicity in event streams and utility in a more systematic way. Figure 4.9a compares Hybrid-I and Hybrid-P over six synthetic streams with varying degrees of periodicity. With no periodicity Hybrid-P and Hybrid-I yield almost the same utility. As we move from a less periodic stream to a more periodic one, Hybrid-P achieves better utility preservation. The utility gain of Hybrid-P in the case of perfect periodicity (periodicity degree = 1) is almost 2x than that of Hybrid-I. It is also worth noting that even for streams with low degree of periodicity, say 0.2, there is a positive, albeit slight, utility advantage of using periodicity based cardinality estimation. For the rest of experiments in this section we fix the degree of periodicity at 0.4.

Pattern length vs. utility. Next, we vary pattern complexity by changing pattern length from 2 to 5 and compare the corresponding utility of different algorithms. As we can see in Figure 4.9b the utility drops as the pattern length increases. This is because the number of matches decreases when the pattern length increases and becomes more complex. Our proposed Hybrid-P achieves only 12% less utility than the Hybrid-Opt, and outperforms all other approaches in this experiment.

Num. of private patterns vs. utility. In Figure 4.9c we vary the number of private patterns while keeping public patterns the same. When there is no private pattern, each approach produces the same utility, as no event suppression is needed. As the number of private patterns increases, the proposed Hybrid-P and Hybrid-I outperform all other alternatives except for the oracle version Hybrid-Opt.

Num. of patterns vs. throughput. Finally, we conduct a throughput test by increasing the number of synthetic queries. Figure 4.9d shows that the most expensive Hybrid-P algorithm attains at least 55% of the throughput of a vanilla system when a large number of queries are present. Its relatively efficiency and superior utility makes it an appeal-

ing approach for CEP privacy. We also note that the Type-level approach achieves only slightly less throughput than the vanilla CEP (No-PP). Considering the fact that Type-level significantly outperforms the Greedy approach in terms of utility, it may also be an attractive alternative when system resources is a concern.

4.8 Related Work

The problem of sequential pattern hiding was recently investigated in [10,46] for static sequence databases. Their approaches function on a set of sequences which are independent from each other. While in our CEP context, the single input stream contains potentially endless events that are temporal correlated. Therefore their approaches cannot be applied to solve our problem. Technically we have been inspired by several principles from their approaches, such as minimizing side-effects and data distortion [46].

The problem of privacy and security has also been looked at in different contexts for streaming data. Authors in [102] consider k-anonymity in streaming data, where the aim is to generalize clusters of tuples into equivalence classes of size at least k. Recently, authors in [47] consider the problem of filtering sensitive user location data in a stream to preserve privacy. The problem of access control in a stream environment has also been studied [75, 83]. These techniques, while useful for their respective purposes, are not applicable as they do not consider CEP pattern query semantics nor the utility optimization problem.

The mechanism of event suppression has been used in relational stream load shedding [39, 100] to keep up output rate when system resources are limited. However, the notion of privacy preference was not considered in the literature of this area. Besides, loading shedding for the specific semantics of CEP pattern queries has not been studied to date.

The general idea of combining “global” (type-level) prediction and “local” (instance-level) prediction in our Hybrid algorithm has been applied to problems in completely different contexts, such as to predict the movement of mobile users [65], and to predict branches in the computer architecture literature [101].

In comparison to the relative lack of research efforts on privacy in the context of CEP, There is an extensive body of literature focusing on privacy preservation in relational databases. K-anonymity [86] and its successors (e.g., [62, 68]) have been extensively studied, which recode the original database in ways such that the modified database satisfies structural properties with privacy protection. Differential privacy [42] has been proposed more recently as a rigorous alternative. However, both approaches focus on statistical queries on static relational data, which are not directly applicable to CEP which is more concerned with producing pattern matches in real-time.

Differential privacy achieves privacy preserving by injecting additive noise to statistical aggregates (e.g., SUM queries). This model, however, does not extend to CEP, because SEQ queries employed in CEP are more concerned with producing individual sequence matches, for which differential privacy are not suitable.

The problem of PP-CEP also bears some resemblance to online query auditing [57, 73], where the goal is to determine whether answering a SQL query given previous queries would disclose private information. While query auditing is mostly about deciding whether answering a query would compromise privacy, PP-CEP focuses on optimizing utility. In addition the significant difference in the data and query model of these two problems make solutions to query auditing inapplicable to PP-CEP.

4.9 Conclusion

In this chapter we study the problem of utility-maximizing event stream suppression with privacy preference. Our problem formulation enables users to specify relative importance of preserving private leaks and producing useful public pattern matches. This transforms event suppression into the problem of computing the expected gain and loss of reporting a pattern match, with the goal of optimizing the overall utility. We first introduce a taxonomy of the utility-maximizing PP-CEP problem. We then propose a suite of solutions, including a LP-based type-level solution that achieves optimal type-level decisions, and a hybrid instance-level solution that exploits current event distributions. Our experiments using real-world and synthetic data show that our proposed approaches can preserve utility effectively and efficiently. Furthermore, we demonstrate that the periodicity-based approach is needed for accurate cardinality estimation in at least one real world scenario and many more synthetically generated datasets.

Chapter 5

Probabilistic Inference of Uncertain Identities over Event Streams

5.1 Introduction of FISS

Real-time object monitoring systems are becoming increasingly popular in domains ranging from healthcare, inventory management, public transit management, traffic monitoring to home safety care [8, 64, 66, 93, 98]. These systems receive high-volume event streams from sensors installed at locations of interest. These events then are filtered and correlated for complex pattern detection, aggregated on various temporal and geographic scales, and transformed into high-level actionable information.

In object monitoring systems, while sensed events are often attached with the unique identification of the tagged object (called "ID-ed" events), it is equally common that events may not carry their object identification (called "non-ID-ed" events). Consequently input event streams may be composed of *both ID-ed and non-ID-ed* events. Reasons that cause such mixed event streams include:

- Events may come from heterogeneous sources. While some devices are capable to provide events with object identification, e.g., RF sensors [8,28,54], other devices do not capture object identification, e.g., Passive Infrared sensors [34,77,88]. Object monitoring applications may consist of heterogeneous monitoring devices, for example both RF sensors and Infrared sensors. For instance in a hospital inventory management system [8], RF taggers are attached to medical equipments to track their distributions, while Infrared sensors are installed in nurse stations and doctor offices to check the sterilization procedure of the equipment. As another example in the hospital environment, it is a regulation to restrict the number of staff in an Operation Room [25] in order to prevent airborne infections to the patient in operation. In some hospitals, Infrared sensors are thus installed in Operation Rooms to check on the number of staff over time, and to observe the door openings to get an insight into airflow transmission. Simultaneously, RF sensors are equipped to

also monitor the hygiene performance of healthcare workers [93]. As a result, data collected in those applications will have mixed ID-ed and non-ID-ed events.

- Furthermore, some objects being monitored, usually human beings, may advertently choose to conceal their identification for privacy concerns, or inadvertently forget to show their identification. In some settings, the environmental condition detection, like detecting an object's motion and presence, is only loosely coupled with an object's identification detection. For instance, in real-time hospital infection control systems, such as HyReminder [93], a sensor installed in the hospital has two components: the motion reader that detects a healthcare worker's behaviors, such as sanitizing hands and entering a patient's room; and the badge reader, which records a worker's identification only when she actively presents her badge to the reader. So when a worker shows her badge and then washes her hands, a "wash" event with her identification is generated. Otherwise, if the worker chooses not to present her badge or forgets to show her badge, a "wash" event *without* any identification is generated. Consequently event streams observed in such applications also consist of both ID-ed and non-ID-ed events.

Given such a mixed input stream, those non-ID-ed events prevent us from performing object-based analytics, such as object tracing, alerting and pattern matching, which usually is the key service needed in object monitoring applications. For example, the HyReminder system [93] continuously tracks each healthcare worker for hygiene compliance by running a set of pattern queries. An example pattern query is to observe whether a worker cleanses her hands before contacting a patient. These pattern queries are based on events associated with an individual worker. This means we have to know the worker's identification associated with an event before we can correctly utilize the event in the query evaluation process. Therefore, in this paper we address a fundamental data trans-

formation problem for event streams mixed with ID-ed and non-ID-ed events, namely to translate raw streams into queriable, probabilistic event streams with object identification.

Contributions. In this chapter, we propose a novel probabilistic inference framework called Familiar- Stranger System, or **FISS** [1], that efficiently transforms raw streams of ID-ed and non-ID-ed events into queriable streams of events with probabilistic object identifications. Specifically, our technical contributions include:

Modeling. We devise a time-varying graphic model to capture the underlying event stream generation process from the physical world, including the key component−the temporal correlations among events. In contrast to existing work on either solely ID-ed events [28, 32, 54] or solely non-ID-ed events [81, 88], we now embrace ID-ed and non-ID-ed events within a single model. This keeps our model simple yet while concisely expressing both the true objection identifications (which we may not observe) and the sensed events (which we do observe) (Section 5.3).

Efficient Inference. Based on our proposed model, we extend a classical inference approach, the Forward-backward algorithm [74], to infer the object identification of non-ID-ed events (Sec. 5.4). However, our experimental evaluation (Sec. 5.6.4) demonstrates that this approach, though suitable for our inference logic, is not efficient enough to provide near-real-time system responsiveness nor scalable for a high volume event stream.

Our second contribution is to devise a suite of strategies for optimizing the performance of the Forward-backward inference. Our key insight is that the Forward-backward algorithm conducts a large number of unnecessary computations during the backward smoothing. We aim to avoid such waste by only computing the "affected" events, i.e., events whose distributions should be revised in the backward smoothing. Our first strategy is to *prune random variables* that can be shown to be unaffected by exploiting the

[1] A familiar stranger is an individual who we repeatedly observe and yet do not know directly. Our system is given this name because it aims to identify those continuously observed non-ID-ed events.

features of ID-ed events. The second optimization, called *finish-flags* mechanism, enables early termination of the backward computation yet without sacrificing inference precision. Lastly, we propose to represent temporal conditional dependencies using Complex Event Processing (CEP) *pattern queries* to capture temporal correlations of events in a large volume stream [13,70,98]. And then chasing down "affected" events can be transformed into a pattern matching. Meanwhile, we devise an advanced data structure customized for streaming uncertain events to speed up the optimized backward probability computation. These strategies together lead to a solution that keeps up with high-volume streams while offering high-precision inference results (Section 5.5).

System and Evaluation. Our third contribution is the implementation and thorough performance evaluation of FISS over event streams of healthcare object monitoring. The experimental results demonstrate that our proposed model achieves better inference precision compared to the MHT model [1, 81]. Moreover, our optimization techniques for the Forward-backward algorithm make it work 15 times faster than the basic implementation [2] (Section 5.6).

5.2 Problem Statement

Physical world. FISS targets environments with well-bounded sub-spaces, such as an Intensive Care Unit (ICU) with dozens of separated patient rooms, where sensors are installed *within* each room. In this setting, the surveillance areas of sensors *do not overlap*. So an object will be detected by *at most one* sensor at a time. This is a distinct difference from the existing RFID data cleaning literature, where an object can have many redundant readings at a time [28, 54, 56, 89], due to assuming sensors with overlapped surveillance areas.

For ease of exposition, the rest of this chapter assumes the environment is an ICU, as

depicted in Figure 5.2a. Such layout is typical in clinics and hospitals. In this chapter, we use the HyReminder [93] system, deployed at University of Massachusetts Memorial Hospital, as a representative application. However our techniques are general and can equally be applied to other applications that work with streams mixed with ID-ed and non-ID-ed sensor readings. In summary, the physical world being monitored is an ICU composed by a set of separate rooms $\mathcal{R} = \{R_1, ..., R_{|\mathcal{R}|}\}$ and a set of healthcare workers being monitored, i.e., objects $\mathcal{O} = \{O_1, ..., O_{|\mathcal{O}|}\}$.

To simplify our discussion, we assume the physical world is closed, namely, the number of rooms and number of objects remain constant. We also assume that each object is independent of each other, which is the "disjoint tracks constraint" commonly adopted in the PDA literature [20, 81, 88].

Input event stream. Each sensor in the ICU interrogates objects' movements in its range and immediately returns its reading to the server. The server collects raw readings from all sensors and merges them into a single input stream. We abstract each sensor reading as an *event*. Each event in the stream, denoted by a lowercase e, corresponds to an instantaneous and atomic occurrence of interest [98]. Events are conceptually grouped into *event types*. Every event type is distinguished by its event type name. The event type of e is derived based on the physical information about the sensor. For example, a motion sensor installed over a patient room's door will generate events of type `Exit-patient-room` and `Enter-patient-room`, whereas a sensor installed at a sanitizer will generate events of type `Sanitize`. The integrated input stream is mixed, meaning events may or may not have object identifications. Such stream is commonly seen in object monitoring applications for various reasons provided in Section 4.1.

Each event type has associated attributes as defined by the schema. We assume an event e in the input stream has the schema: `event-type (nonce,ts,room#,OID)`. Here `nonce` is a unique number attached by the server distinguishable for any events.

`room#` $\in \mathcal{R}$ represents the room of the sensor that generates e, i.e., e's location. As described previously, our target environment is well-bounded and closed, so `room#` is assumed to be accurate. `OID` is the object identification associated with e. Since the object identification of e may not be available, we allow `OID` to be a concrete $O_i \in \mathcal{O}$, or `OID` $= null$. In the rest of this chapter, we denote an ID-ed event as $\hat{e}$, and a non-ID-ed event as $\widetilde{e}$.

Output event stream. The output of FISS is a probabilistic event stream, where each event has an associated probabilistic object identification. As in most probabilistic data management models [9,22], we represent the probabilistic object identification using the possible-worlds semantics. Namely, the `OID` of an output event consists of a set of mutually-exclusive *alternatives* with associated *confidence* values. Intuitively, the object identification takes the value of one of its alternatives, and the probability of taking a particular alternative is given by its confidence value. For example, consider an output event `enter-patient-room(127,12/01/28 17:30:00,R1,<`O_1`:0.6,`O_2`:0.3,`O_3`:0.1>)`. This event is most likely associated with object O_1 (60% chance), but may also be with O_2 (30%) or O_3 (10%).

Intuitively, the uncertainty of `OID` can change over time when additional information is obtained from a newly arrived event. Then the previously inferred object association could be revised to gain a more accurate probability. This revision task is called *smoothing* in probabilistic inference algorithms [82]. Therefore, FISS is designed to also output a special event called *revision*, which represents the modification over a previously inferred `OID` of an event. A revision has the format: `Rev(nonce-of-previous-event, new-OID)`. Whether to output revisions is an option chosen by the user. In this chapter FISS supports three commonly-used output strategies: (i) report any change of identification association obtained during the smoothing; (ii) report when the revision results in 100% confidence; or (iii) report when the revised probability is more than a threshold,

say 50% different from the previous probability of that event.

In summary, FISS outputs a stream consisting of probabilistic events and optionally revision events. Such output stream can then be fed into an event management system for object-based analytics. For example, the output stream from FISS for the HyReminder application will be used for hand hygiene pattern detection [93]. Many stream systems are capable to process various queries over such probabilistic streams, including relational queries [56], complex event queries [80] and aggregate queries [35]. Also, several event stream processing systems [13, 21, 69] support revisions that amend previously arrived events.

The inference problem. Finally, the problem we solve in this chapter is: given an input stream of raw sensor reading events, where an event may or may not have an associated object identification, we derive a queriable, probabilistic event stream where each event is associated with probabilistic object identification. We aim to infer the identification association as accurately as possible and to do it in an efficient manner so to achieve near real-time responsiveness.

5.3 Proposed Graphical Model

In this section we present our proposed probabilistic model that captures the correspondences among sensed events and monitored objects. In contrast to existing work on either solely ID-ed events [28,32,54] or solely non-ID-ed events [81,88], our solution embraces ID-ed and non-ID-ed events within a single model. This keeps our model simple yet while concisely describing the physical world.

$\hat{e}$	an ID-ed event
$\widetilde{e}$	a non-ID-ed event
s_i	room state variable for Room R_i
ψ_j	object identification association variable for $\widetilde{e_j}$
ϵ_j	information variable for event e_j

Table 5.1: Summary of symbols

5.3.1 Components of the Model

Our model describes the world using random variables that present both true object identification associations, which we may not observe, and the input events, which we directly observe.

Given numerous event types abstracted in the representative HyReminder application, to simplify our discussion, we focus on two important event types `Enter-patient-room` and `Exit-patient-room`, or `Enter` and `Exit` in short respectively. These two types of events are most critical because they tell us an object's movements throughout the ICU, which serve as the basic knowledge from which we can estimate object identification.

Time and space. Same as most data stream management systems [15, 98], we divide time into a sequence of discrete epochs of, for example, one second in duration. All sensor readings that occur in the same epoch are treated as simultaneous. Each event e can thus be attached with a timestamp from the discrete epoch domain, denoted by `e.ts`. In our representative application, assuming the epoch granularity is one second, a healthcare worker can conduct at most one action in a single epoch. That is, one object will trigger *at most one* `Enter` or `Exit` event at an epoch. As for space, given the well-bounded physical layout we target, it suffices to model the space as a discrete set of rooms, while each room connecting to the hallway, as shown in Figure 5.2a.

Room state variables. In our problem setting (Section 5.2), a non-ID-ed event does not carry an objection identification, but does carry its accurate location, in terms of `Room#`, and timestamp `ts`. Intuitively, we can utilize the `Room#` and timestamp as a

starting point to estimate the `OID` of the event. Casually speaking, suppose a non-ID-ed `Exit` event $\widetilde{e_j}$ occurs at the room R_i, we can figure that those who were in room R_i previously have the possibility to exit the room now. So we will consider those objects as candidates to associate with $\widetilde{e_j}$. Similarly, for a non-ID-ed `Enter` event, we will look for those objects who were outside the room previously, as they are possible to enter the room at this time. This intuitive observation suggests us to maintain the state of each room, i.e., which objects are in the room at a certain time, so that later we can refer to this information for object identification association.

Therefore, we define a *room state variable* for each room. Namely, for each $R_i \in \mathcal{R}$, let random variable s_i^t present which objects are in R_i at a given time t. Ideally we wish to express a room state variable s_i^t as a set of objects. But because of the uncertainty of `OID` of an event, a room state could be uncertain as well. In this case each object in the room state will be associated with a confidence value. For example, s_1^{12} = ¡O_1:0.3, O_2:0.7¿ represents that for Room R1 at time 12, object O_1 has 30% possibility to be in that room while O_2 has 70% possibility. A room state could also be an empty set when no healthcare worker is spotted in the room. In addition, in our representative ICU layout (shown in Figure 5.2a), every patient room has a door connecting to the hallway. In our model we treat the hallway as a special kind of room, with a corresponding random variable called *hallway state*, denoted as s_H.

Object association variables. Since our goal is to infer object identification associations for non-ID-ed events, we next define an *object association variable* for each non-ID-ed event $\widetilde{e_j}$, denoted as ψ_j. ψ_j expresses the probability of an object being associated with $\widetilde{e_j}$. It is important to note that the association can be changed over time, because when obtaining additional information from new events, we may revise the association to a more accurate probability. Therefore, ψ_j is a *temporal* random variable, meaning its value evolves over time. Specifically, ψ_j^t represents the object identification association

for event $\widetilde{e_j}$ at time t, where t is different from and regardless of $\widetilde{e_j}$'s timestamp $\widetilde{e_j}.ts$. For example, given a non-ID-ed event `enter(122,12,R1,?)`, where 122 is the nonce, 12 is its timestamp and "?" means its `OID` is unknown, then an association variable is first created at time 12, ψ_{122}^{12}. Assume ψ_{122}^{12}= $\{O_1$:0.5,O_2:0.5$\}$, representing that at time 12 we reckon objects O_1 and O_2 have equal chances to be associated with this event. Later at time 14, suppose we are able to improve this association, we set ψ_{122}^{14} = $\{O_1$:0.9,O_2:0.1$\}$, meaning at time 14 we reckon O_1 has 90% probability to associate with this event while O_2 has only 10%.

Event information variables. The input event stream is the source of run-time information that we use to infer object associations. We thus abstract the information conveyed by an input event, e_i, as an *event information variable*, denoted as ϵ_i. Specifically, given an ID-ed event, its information variable presents an object's action (via its event type, `Enter` or `Exit`) and location (via its `Room#` attribute) at time t. In comparison, given a non-ID-ed event, we need to first infer its probable `OID` and then extract its information variable. Consequently, a non-ID-ed event with inferred object association presents one or more objects' probable actions and locations. To discriminate between these two scenarios, let $\hat{\epsilon_k}$ denote an information variable for an ID-ed event $\hat{e_k}$, and $\widetilde{\epsilon_j}$ denote an information variable for a non-ID-ed event $\widetilde{e_j}$.

5.3.2 Dependencies for Object Association

Next we introduce the conditional dependencies that describe how each component in our model interacts with the other ones, which is used later to compute the probabilities of the object associations. Graphically, the random variables stated previously are represented using *nodes*, and the conditional dependencies presented in this section are represented using *edges*, as depicted in Figure 5.1.

Room state evolution. This conditional dependency describes how a room state vari-

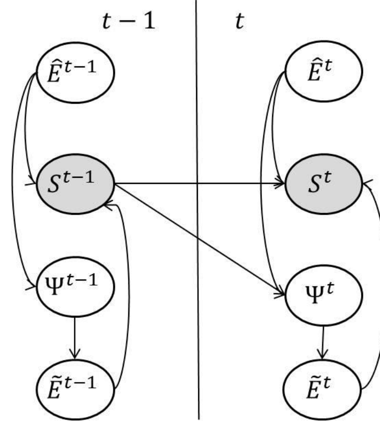


Figure 5.1: Graphical model for FISS

able, s_i, evolves as new events continuously arrive. Intuitively, at each epoch, the new state is the old state plus objects that are just entering the room, and minus objects that are just exiting the room. In other words, a room state at time t *depends on* its previous state of time t-1 and all objects' movements at time t. Given that objects' movements are provided by newly arrived events, which are modeled as event information variables, it is intuitive that s_i^t conditionally depends on s_i^{t-1} and all event information variables at t. Formally, let $\hat{E}^t$ = $\{\hat{\epsilon_k}|\forall\hat{\epsilon_k}.ts = t\}$ be the vector of all information variables for ID-ed events at t, $\widetilde{E^t}$ = $\{\widetilde{\epsilon_k}|\forall\widetilde{\epsilon_k}.ts = t\}$ be the vector of all information variables for non-ID-ed events at t, then this conditional dependency can be expressed as $p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E^t})$. Note that the special hallway state s_H is maintained as follows: an `Enter` event at any ordinary room will be considered as an `Exit` event for the hallway, and vice versa.

Object association for non-ID-ed events. Suppose a new non-ID-ed event $\widetilde{e_j}$ occurs at room R_i. Intuitively we know that if $\widetilde{e_j}$ is an `Exit` event, then those who were in room R_i at time t-1, i.e., objects expressed in s_i^{t-1}, will be alternative objects to be associated with $\widetilde{e_j}$. In addition, if there are other ID-ed events simultaneously occurring at t, we can safely exclude those objects from being alternatives of $\widetilde{e_j}$ (because we have assumed that one object can conduct at most one action at an epoch). Formally, this conditional dependency can be expressed as $p(\psi_j^t|s_i^{t-1}, \hat{E}^t)$. Symmetrically, if $\widetilde{e_j}$ is an `Enter` event,

then those who were outside R_i at time t-1, i.e., in the hallway, should be the alternate objects associated with $\widetilde{e_j}$. Such objects are listed in the hallway state s_H^{t-1}. So this conditional dependency can be represented as $p(\psi_j^t|s_H^{t-1}, \hat{E}^t)$.

Formal joint model. Now we are ready to state the formal description of our model. To make the notations compact, let $S^t = \{s_i^t|\forall R_i \in \mathcal{R}\}$ be the vector of all the room states at time t, $\psi^t = \{a_j|\forall \widetilde{e_j}.ts = t\}$ be the vector of all object identification association variables at time t. Assume the initial room states S^0 are known. We combine the above conditional dependencies to define a joint model over the entire domain:

$$p(\boldsymbol{\Psi}, \boldsymbol{S}, \widetilde{\boldsymbol{E}}|\hat{\boldsymbol{E}}, \widetilde{\boldsymbol{E}}) = p(S^0) \prod_t \prod_{\widetilde{e_j^t}} p(\psi_j^t|S^{t-1}, \hat{E}^t)$$
$$\prod_{i\in\mathcal{R}} p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E}^t)$$

where $\boldsymbol{\Psi}$ is the vector of all object identification associations over all times, similarly $\boldsymbol{S}$ is the vector for all room states over all times; $\hat{\boldsymbol{E}}$ is the vector for information of ID-ed events over all times and $\widetilde{\boldsymbol{E}}$ is the vector for the information of non-ID-ed events over all times. Our model can be viewed as a particular case of a Dynamic Bayesian Network (DBN) [74] but with conditional probability functions specially designed for our problem, which is depicted graphically in Figure 5.1.

5.4 Inferring IDs Over Streams: Initial Effort

As the raw stream is mixed with ID-ed and non-ID-ed events, the task of translating it into a probabilistic stream is treated as an inference process in our work. Inference is essentially to *estimate the true object association* for non-ID-ed events. Formally speaking, our inference task is, from the joint distribution $p(\boldsymbol{\Psi}, \boldsymbol{S}, \widetilde{\boldsymbol{E}}|\hat{\boldsymbol{E}}, \widetilde{\boldsymbol{E}})$, to compute the posterior distribution of all object association variables, given a sequence of information $\hat{E}^{1:T} = \{\hat{E}^1, ..., \hat{E}^T\}$ and $\widetilde{E}^{1:T} = \{\widetilde{E}^1, ..., \widetilde{E}^T\}$. Namely, to compute, for each association

variable $\psi_j^t \in \{\Psi^1, ..., \Psi^T\}$, the distribution $p(\psi_j^t | \hat{E}^{1:T}, \widetilde{E}^{1:T})$.

5.4.1 Inference using FB Algorithm

We first set out to adapt the classical *Forward-backward* (FB) inference algorithm [74,82]. In this section, we describe the main intuition of how and why the FB inference algorithm tackles our problem. This provides a technical context for our later optimizations in Section 5.5.

To find the posterior distributions, i.e., the most likely states for random variables, the FB algorithm involves three steps [74, 82]: (1) computing a set of forward probabilities, (2) computing a set of backward probabilities, and (3) computing smoothed values. Steps (2) and (3) are typically performed simultaneously, called "*backward and smoothing*", which look backward at any past distributions while accounting for the current information, and then obtain more accurate results when possible.

Extension for FB algorithm. The FB algorithm has originally been designed for static data set, we now *extend it to the streaming context*: all arrived events are stored; whenever a new event arrives, the above three steps are performed over all events received so far; the new event is output with (inferred) object identification. Updated values during smoothing may be optionally output as revisions if the user chooses to. In the rest of this paper, we use the term "FB algorithm" to refer to this extended version of FB.

Given that our target applications are real-time systems, typically only the relatively recent events are of interest. Though a revision of a historical event can increase the overall inference correctness, it may not be practically useful at the current time. For example, the HyReminder system would monitor HCW's hygiene behaviors in the last one hour [93]. So a revision for an event more than one hour ago would be assumed not to affect the monitoring results. Therefore, FISS allows user to set an application-specified temporal boundary, call *smoothing window* [54], that restricts the scope of backward revi-

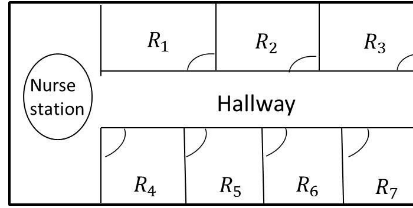
sion. The FB algorithm then will only smooth events within this temporal sliding window.

Estimating object associations. Given the conditional dependencies of object association specified in Section 5.3.2, for a newly arrived event that is non-ID-ed, say $\widetilde{e_j}$, the FB algorithm performs the following in the "forward" step: compute the conditional distribution $p(\psi_j^t|s_i^{t-1}, \hat{E}^t)$ (for an `Exit` event) or $p(\psi_j^t|s_H^{t-1}, \hat{E}^t)$ (for an `Enter` event). Then the resulting distribution is the estimate of `OID` for $\widetilde{e_j}$. Once all non-ID-ed events are associated with probable `OID`s at time t, the FB algorithm keeps room states up to date by computing the conditional distribution $p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E^t})$.

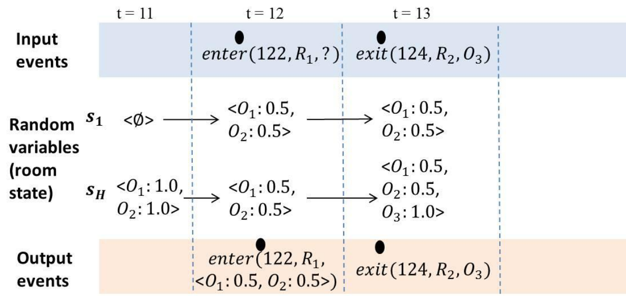
Then in the "backward and smoothing" step, the FB algorithm revisits past distributions while accounting for the fresh information gained at the current time t. Still considering the conditional dependency $p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E^t})$, but now FB interprets the dependencies in a retrospective manner: if an object, say O_x, exits Room R_i at time t, then O_x would have to have stayed in Room R_i at time t-1. Suppose previously the room state s_i^{t-1} was uncertain about O_x's staying, then the FB algorithm is now able to ensure that O_x must have been in s_i^{t-1} with 100% confidence. As a result, FB amends the previous value of s_i^{t-1}. And then this updated information is carried by FB to compute backward probabilities for other older variables. Since the object association depends on room states, as expressed by $p(\psi_j^t|s_i^{t-1}, \hat{E}^t)$, the revision of a room state may trigger the revision of an object association in turn. Suppose a historic `Exit` event $\widetilde{e}$ at time t depended on the room state s_i^{t-1}, and now s_i^{t-1} is revised. Then FB will revise the `OID` of $\widetilde{e}$ by computing the backward probability of $p(\psi_j^t|s_i^{t-1}, \hat{E}^t)$. The backward and smoothing computation continues going back, one epoch at a time, until reaches the smoothing window boundary.

Next in Example 5.1, we illustrate how the FB algorithm infers object identifications for non-ID-ed events using the conditional dependencies defined in Section 5.3.2.

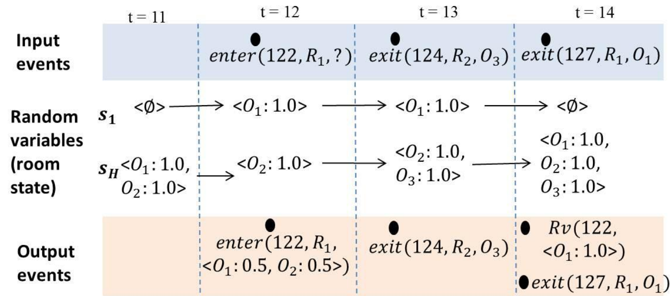
Example 5.1 *We make use of a representative layout of an ICU, as shown in Figure 5.2a.*



(a) Example ICU floor plan



(b) Inference results at time 12 and 13



(c) Inference results at time 14

Figure 5.2: Forward-Backward inference example

Figures 5.2b and 5.2c depict Room R1's state s_1 *and Hallway's state* s_H*, while Room R2 to R7's states are not shown due to space constraints. We assume there are only three healthcare workers in the ICU, i.e.,* O_1*,* O_2 *and* O_3*.*

Figure 5.2b depicts Room R1's state s_1 *and the hallway's state* s_H*' starting from time t=11. Namely, at time 11, there was no one in Room R1, and there were two workers,* O_1 *and* O_2*, in the hallway.*

Suppose at time 12, a non-ID-ed event `enter(122,R1,?)` *arrives, where 122 is the nonce, R1 is the room number and "?" means the* `OID` *is unknown. The FB*

algorithm first performs the forward inference to assign possible object identifications for this event. Based on conditional dependency $p(\psi_j^t|s_H^{t-1}, \hat{E}^t)$, FB checks the hallway's status at time 11. Intuitively, those in the hallway at t=11 have the possibility to enter Room R1 at t=12. So O_1 and O_2 both are alternatives for this event. To simplify our discussion, we assume O_1 and O_2 have equal chances of entering R1. So FB assigns an equal confidence for them, i.e., 50% vs. 50%. Consequently, a probabilistic event enter(122,R1,<O_1:0.5,O_2:0.5>) *is output. And then FB computes room states s_1^{12} and s_H^{12} based on the conditional probability $p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E}^t)$. Since the* enter *event is uncertain, room states s_1 and s_H at time 12 are uncertain too. Namely, s_1^{12} is set to ¡O_1:0.5,O_2:0.5¿, saying that either worker O_1 or O_2 is at Room R1 at t=12, each with 50% probability. Symmetrically, the hallway state s_H^{12} is set to ¡O_1:0.5,O_2:0.5¿. And then FB goes back in time, namely from t=12 to the beginning of the smoothing window (say t=1), to compute backward probabilities for all random variables (not shown in Fig. 5.2).*

Next at time 13, event exit(124,R_2,O_3) *arrives, expressing that worker O_3 exits Room R2 at t=13. This is an ID-ed event, so no identification association is performed. The FB algorithm then computes Room R3's state and hallway's state at t=13. Next FB performs backward smoothing over all random variables. Note that Room R1's historic states are computed for backward probability, however, none of them is changed given* exit(124,R_2,O_3). *Actually, after the backward probabilistic computation, s_1^{12} is left as it was. That is, we are still not sure which one of O_1 and O_2 was in Room R1 at t=12 and t=13.*

Next at time 14, as shown in Figure 5.2c, event exit(127,R_1,O_1) *arrives. This ID-ed event conveys that worker O_1 exited room R_1 at t=14. Intuitively we now figure that O_1 must have entered Room R1 previously. By computing the backward probability of $p(\psi_j^t|s_H^{t-1}, \hat{E}^t)$, FB is able to amend Room R1's historic states (s_1^{13} and s_1^{12}) and the*

hallway's historic states (s_H^{13} and s_H^{12}) to reflect this finding. Namely, s_1^{12} and s_1^{13} are both changed to < O_1:1.0>, meaning it is assured that worker O_1 was in room R_1 at time 12 and 13. Symmetrically, s_H^{12} and s_H^{13} are revised to present that worker O_2 were in the hallway at time 12 to 13. If the user sets to output revision events, then a revision `Rev(122,` $< O_1 : 1.0 >$`)` *is produced. This revision event conveys that the historical event with nonce 122 should be associated with O_1 with probability 1.0.*

5.4.2 Discussion of Deficiencies of FB Algorithm

The conventional implementation of the FB algorithm described in Section 5.4.1 is straightforward and easy accessible (by extending an off-the-shelf AI software). However, it is not able to achieve near-real-time responsiveness, nor to scale to a large volume stream, as we will explain below and also demonstrate by experiments in Section 5.6.4.

In the conventional implementation of FB, the backward and smoothing step is very computation-intensive. This is because this step computes backward probabilities for every random variable one epoch at a time. Suppose a smoothing window contains *n* epochs, then for *each* random variable, the smoothing step needs to check relevant conditional probabilities *n* times [74].

In fact, we observe that many of these computations are unnecessary. For example, in Figure 5.2b, the event `exit(124,`R_2`,`O_3`)` occurring at time 13 will not change room state s_1 nor the object association for `enter(122,`R_1`,?)`. But unfortunately the FB algorithm would nonetheless run the backward computation for all random variables. Our intuition is that if the distribution of a historic event's `OID` is not affected by the newly added event, the smoothing will produce an "empty" computational result. If we are able to *skip* probabilistic computations for those "unaffected" events, then the cost of FB can be reduced − henceforth its efficiency can be improved.

5.5 Inference Speedup: Optimization Strategies

To overcome the above deficiencies suffered by the FB algorithm, we now devise three advanced techniques, namely, pruning unaffected variables, early termination of smoothing and selective smoothing, to optimize the backward and smoothing step. These techniques lead to a solution that keeps up with high-volume streams while offering the equally high precision of inference as the classical FB algorithm.

Our intuition is, in order to chase down the random variables that need to be computed during the backward and smoothing, we wish to find out which kind of random variables could ever be affected by the newly updated information.

Definition 5.1 ***Affected random variable**: a random variable that will be set a different value from its current value by the backward probability computation during the inference.*

5.5.1 Pruning Unaffected Variables

We first show an important property of random variables in our model, which tells us which random variables are definitely *not* affected. The merit of knowing that a random variable is not affected is that the backward computation can thus safely skip this random variable while guaranteeing to offer the same inference result.

Theorem 5.1 *If an alternative of a random variable is associated with 100% confidence, then this alternative's confidence will never be changed by the FB inference in future.*

Proof: In our model, an alternative with 100% confidence of a random variable can be produced by two ways: first, it is directly given as input by an ID-ed event; and second, it is inferred by the FB algorithm. So we prove this theorem case by case. In the first case, the information brought by an ID-ed event is considered as a ground truth which

does not depend on any other random variable. Henceforth it will not be changed during the inference process. In the second case, we prove by contradiction. If an alternative's confidence will be changed later, then that means it is still uncertain now. In that case, it would not have had 100% confidence.□

Lemma 5.1 *If all alternatives of a random variable are associated with 100% confidence, then this random variable will not be an affected random variable.*

Proof: This lemma is derived from Theorem 5.1 and Definition 5.1.□

Lemma 5.1 is intuitive. Let us consider the random variables in Figure 5.2b for example. At the beginning, the hallway state at time 11, i.e., s_H^{11}, is assured to be $< O_1$:1.0,O_2:1.0$>$. This implies s_H^{11} will never be changed by any smoothing afterwards, because we are already 100% sure who were in the hallway at time 11.

Therefore the first optimization we propose is to skip the backward computation for those "unaffected" random variables, i.e., those whose alternatives are associated with 100% confidence. Though simple, this optimization is especially appealing in our target environment where the input stream is mixed with ID-ed and non-ID-ed events. Based on our model, an ID-ed event provides relevant random variables with accurate states. Thus there are potentially a large number of random variables with accurate values in our system. Skipping backward computations for those variables will henceforth significantly reduce processing cost .

5.5.2 Early Termination Using Finish-Flags

Next, we introduce a second optimization strategy that improves performance by early terminating unnecessary computations in the backward and smoothing process, yet without sacrificing inference precision. We start by presenting the intuition of this mechanism using the following example.

Example 5.2 *Let us consider the random variables in Figure 5.2b again. Based on Lemma 5.1, we know that the hallway state variable s_H^{11} is unaffected. We further observe that all hallway states earlier than s_H^{11} will also never be changed. The intuition is, since the state at time 11 is assured, the smoothing performed at time 11 must have completed all necessary computations for s_H with the accurate information already. Thus any smoothing process after time 11 will never change those historical states afterwards. Consequently, it is safe to skip backward computations for all hallway states earlier than time 11.*

Example 5.2 suggests that for a room state variable, the backward step should be able to skip the computations for all historic states before an accurate state. The justification for this strategy is given in Lemma 5.2.

Lemma 5.2 *For a room state random variable s_i, if its state at time t, i.e., s_i^t, has all alternatives associated with a 100% confidence, then previous states of s_i that are earlier than t will not be affected by any inference occurring later than t.*

Proof: This feature is due to the Markov property of our model and the construction of the Forward-backward algorithm [74, 82]. When computing backward probabilities for s_i upon the arrival of an event e_j^t, the information carried by e_j^t, i.e., ϵ_j, is passed through a sequence of states of s_i, from the current time t to the beginning of the smoothing window, say t_0. The passing is from one epoch to its contiguously earlier epoch. The backward distribution $p(s_i^{t_k}|\epsilon_j, s_i^{t_k+1}, ..., s_i^t)$ computed at every epoch t_k provides the probability of being in the state of $s_i^{t_k}$ given event e_j^t and all future states. Noting that our model (defined in Section 5.3) exhibits the Markov property, namely the conditional probability distribution of future states of the process depends only upon the present state, not on the sequence of events that preceded it. So the backward distribution is equal to

$p(s_i^{t_k}|\epsilon_j, s_i^{t_k+1})$. If the value of ϵ_j and $s_i^{t_k+1}$ does not change, $p(s_i^{t_k})$ will not change neither. Now that s_i^t is assumed to be unaffected, its contiguously previous state s_i^{t-1} will be unaffected too. Then it can be deduced that all previous states of s_i^t will be unaffected. □

Lemma 5.2 gives a basis for our **early termination** mechanism, namely to stop the backward and smoothing when a room state with 100% probability is reached. To efficiently implement this mechanism, a mark for room state random variables, called *finish-flag*, is created. A finish-flag is attached to a state when the state is certain. The smoothing process for this random variable stops once a finish-flag is encountered. Suppose there are thousands of historical states before the finish flag, which is realistic in our streaming environment, using the finish-flag then can save thousands of backward probability computations. An example of placing and utilizing finish-flags is described below.

Example 5.3 *Figure 5.3 shows our optimization techniques applied to the same event stream as in Example 5.1. Room R1's states and hallway's states are listed at the bottom of Fig. 5.3. From Example 5.1 we know that s_1^{11} and s_H^{11} have all alternatives with 100% confidences. Hence two finish-flags are created for s_1^{11} and s_H^{11} respectively at time 11. Therefore the backward and smoothing computation for s_1 or s_H will not check any states earlier than time 11. In comparison, in Example 5.1, the backward computation goes all the way back until the starting point of the smoothing window, which potentially involves hundreds of historic states.*

Furthermore, when creating finish-flags, we can also dynamically purge the states of random variables by removing those states prior to finish-flags, because those states will never be affected afterwards. **Purging by finish-flags** is a complementary policy to purging by the smoothing window in FISS. Purging by finish-flags is more aggressive and henceforth saves more space. This is important in stream processing where runtime data structures need to be purged to avoid memory depletion [15, 98].

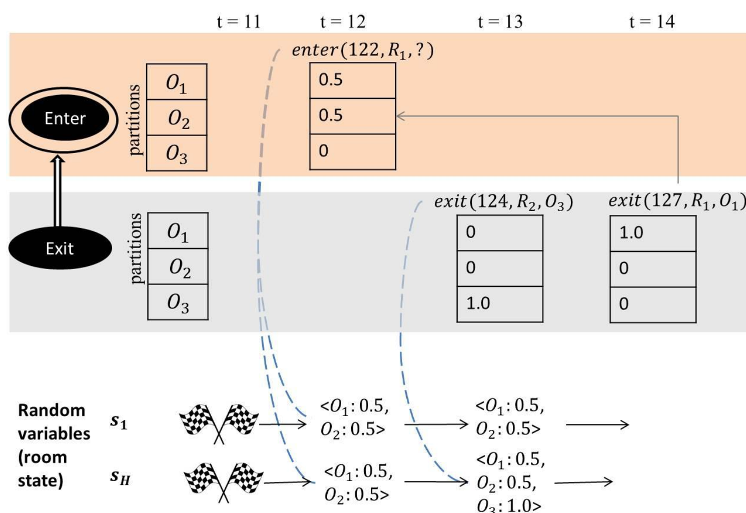


Figure 5.3: Example of optimized smoothing when *exit(127,*R_1*,*O_1*)* just arrives. *Upper*: partitions of probabilistic events. *Bottom*: random variables with finishing-flags. *Dashed lines*: hyper-links from events to random variables.

5.5.3 Selective Smoothing via Pattern Matching

Our proposed pruning (Section 5.5.1) and early termination (Section 5.5.2) strategies sift out random variables that are definitely unaffected, which serves as a preliminary round of seeking affected random variables. In this section, we propose a method that declaratively detects affected events.

Motivation of pattern matching based method

We first introduce the notion of affected events, and then present an important observation regarding affected events in our model.

Definition 5.2 ***Affected event*** *is an event whose object association will be changed by the backward probability computation during the inference.*

Lemma 5.3 *An affected random variable depends on at least one affected event.*

Proof: Prove by contradiction. If a random variable depends only on events that are unaffected, then based on the definition of DBN [74], this random variable will not be

affected either. This is a contradiction of the assumption that the variable is affected. □

Motivated by Lemma 5.3, we propose to detect affected events and then follow the conditional dependencies to locate affected random variables, instead of searching through all random variables. There are two main benefits of such event detection based strategy: First, the total number of events in our system is much less than the number of random variables, as can be seen from our graphic model defined in Section 5.3 and depicted in Figure 5.1; Second, detecting affected events is more intuitive than detecting random variables, because events are observed by human beings, while random variables are abstractions.

Therefore we now rephrase conditional dependencies $p(\psi_j^t|s_H^{t-1}, \hat{E})$ and $p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E^t})$ from events' perspective. We found that these dependencies essentially specify the correlations between events, as described below.

Observation 5.1 *The backward probability of $p(s_i^t|s_i^{t-1}, \hat{E}^t, \widetilde{E^t})$ tells that if there is an* `Exit` *event for object O_x at room R_i, then there must be a previous* `Enter` *event for O_x at room R_i.* [2]

This observation raises an interesting challenge: can we declaratively specify which event will be affected by a given event? The answer is yes. Intuitively, it appears that an affected event should satisfy the temporal constraint, the event type constraint, as well as the value constraints on `OID`, as presented in Observation 5.1. Specifically, our proposed solution is to declaratively specify affected events by using *Complex Event Processing (CEP) pattern queries*.

The reason of choosing CEP pattern queries is two-fold. First, for the sake of *expressiveness*, a CEP pattern query specifies how individual events are filtered and multiple

[2]Symmetrically, the backward probability also tells that if there is an `Enter` event for object O_i at room R_j, then there must be a previous `Exit` event for O_i at a certain room. However, in our model, entering a room is equal to exiting the hallway, and vice versa. So it is unnecessary to repeat the symmetrical logic in Observation 5.1.

events are correlated via time-based and value-based constraints. This fits our specific problem well. Second, for the sake of *performance*, CEP engines are known for their sophisticated capabilities for detecting temporal correlated patterns of events in huge volume event streams [6, 13, 70, 98]. Thus if we can specify affected events via CEP pattern queries, then the CEP technology can help FISS to detect affected events effectively and efficiently.

Conditional dependency as pattern queries

Next we describe how to declaratively specify affected events using CEP pattern queries based on the conditional dependencies in our model. As a preparation, we describe the properties of an affected event in further depth by extending Observation 5.1.

Lemma 5.4 *Given an event e, suppose event e' is affected by e, then e' should satisfy the following conditions: (1) $e'.ts$ ¡ $e.ts$; (2) $e'.\mathtt{OID}$ is uncertain; (3) $e'.\mathtt{OID}$ and $e.\mathtt{OID}$ contain at least one common alternative; (4) if e's event type is `Exit`, then e''s event type is `Enter`, and vice versa.*

Proof: Conditions (1) and (2) can be directly derived from Theorem 5.1. We prove condition (3) by contradiction. Suppose the alternative objects for e are $\{O_{j_1},...O_{j_n}\}$. If none of them is an alternative object of e', then that means we are 100% sure e' is not associated with any of $\{O_{j_1},...O_{j_n}\}$. Based on our assumption of "disjoint tracks constraint" presented in Section 5.2, then we can figure that the association of e does not affect e'. This contradicts the definition of our affected event (Def. 5.2). For condition (4), this is derived from Observation 5.1.□

Our **key idea** is to compose a pattern query, using the commonly used CEP pattern query syntax, by imposing every conditions in Lemma 5.4 with proper clauses.

```
CREATE QUERY Q1
PATTERN SEQ(Enter x, Exit y) ON event-history
WHERE skip_till_next_match(x, y) {
            x.OID ≅ y.OID         AND
            x.Room# = y.Room#       }
WITHIN 1 hour
```

Figure 5.4: Pattern query Q1 specifying affected event

Since the conditions in Lemma 5.4 require comparisons of probabilistic `OID`s, we need to extend the CEP pattern query semantics to support equivalence tests on a probabilistic attribute.

Definition 5.3 *The **equivalence test over two `OID`s**, denoted as ≅, returns true if the two `OID` contain at least one common alternative object, otherwise returns false.*

For example, <A:0.5, B:0.5, C:0> ≅ <A:0.7, B:0, C:0.3> returns true, while <A:0.5, B:0.5, C:0> ≅ <A:0, B:0, C:1.0> returns false.

Next we create the pattern query named Q1 (shown in Figure 5.4) to express the conditions of affected events as specified in Lemma 5.4. Q1 exhibits three unique features.

- Q1 utilizes the ***sequence pattern SEQ*** to specify the temporal order in which the events must occur. Namely, when a new `Exit` event arrives we search for an `Enter` that occurs previously and satisfies all the predicates in Q1.

- Q1 is defined using the appropriate *event selection strategy* that addresses how to select the relevant events from a large volume of events. Namely, Q1 uses the ***skip-till-next-match selection strategy*** to impose that irrelevant events are skipped until an `Enter` event matching all constraints is encountered. If multiple events in the event history can match the constraints, only the first one, i.e., the most recent one, is considered.

- In the `WHERE` clause Q1 specifies predicates on the probabilistic `OID` attribute, which requires the two events to contain at least one common object alternative (Def. 5.3).

We have thus shown the successful declarative specification of affected events. This is a transformation from the formal conditional dependencies to practical CEP pattern queries. It is worth noting that *not* all conditional dependencies can be transformed into pattern queries. CEP pattern queries put great emphasis on specifying sequence patterns of events. We thus take this feature to represent temporal conditional dependencies in DBNs, i.e., *temporal arcs* crossing different epochs [74]. For other conditional dependencies that are not temporal, we will use the conventional presentation.

Runtime affected variable detection

Next we describe our proposed algorithm for efficiently detecting affected events and random variables by leveraging CEP pattern matching technology. Specifically, we devise an advanced data structure called *hyper-linked Queue*, or *hyQ* in short, to manage events and random variables in an integrated manner. *hyQ* maintains *partitioned queues* of probabilistic events in order to speed up event pattern matching. It also builds *hyper-links* between events and relevant random variables to help efficiently trace the conditional dependencies.

Initialization. At the initialization stage, suppose all pattern queries created for the model (using the technique presented in Sec. 5.5.3) are registered in FISS. The set of queries is denoted as $\mathcal{Q}$. Following the well-established NFA (Non-deterministic Finite Automata) based event pattern detection mechanism [70, 98], an NFA is created for each pattern query $Q \in \mathcal{Q}$ to represent the sequence of event types. For example, the NFA for Q1 contains event types `Enter` and `Exit`, as depicted in Figure 5.3 by two black ovals. The arrow from `Enter` to `Exit` imposes the temporal order between them, namely an `Enter` event should be ahead of an `Exit` event, as specified in Q1.

A hyper-linked queue, i.e., *hyQ*, is created for each event type to store events of the same type in time order. In order to expedite the equivalent test on `OID`s of events,

every *hyQ* is partitioned on the probabilistic `OID` attribute. The conventional partitioning mechanism in CEP engines [64, 98] dispatches events based on the discrete values of an attribute. However, in our context, `OID`s are uncertain. We thus propose to build *partitions based on the alternatives* of an uncertain `OID`, while recording the *confidence* of each alternative in the partitioned event. This special partitioning mechanism allows an uncertain event to belong to multiple partitions (when its `OID` has more than one alternative). Figure 5.3 shows two *hyQ*s for query Q1, namely, the gray block is the *hyQ* for event type `Exit`, while the pink block is the *hyQ* for event type `Enter`. Since we assume there are totally three objects in the example application, events in *hyQ*s are partitioned on these three alternatives. Note that event enter(122, R_1, ?) falls into two partitions, O_1 and O_2, because it could be associated with either of these two objects.

For a room state variable, in order to provide efficient ordered access, we index its states by timestamp. Figure 5.3 illustrates such an arrangement. In the bottom of Figure 5.3, for example, the state variable for room R1, s_1, has its states organized in time order. Also note that the finish-flags have been established for all historic states where applicable.

Furthermore, to keep track of the probabilistic dependencies, a *hyper-link* is maintained for each event, connecting to all random variables that *depend on* this event. As illustrated in Figure 5.3, the hyper-link for event enter(122, R_1, ?) points to room R1's state s_1^{12} and the hallway's state s_H^{12}, because the distributions of s_1^{12} and s_H^{12} depend on this event.

Selectively Backward Smoothing. Next we describe the customized backward and smoothing process of FISS. When a new event e arrives, FISS first performs forward inference, same as Step 1 of the classical FB algorithm (Section 5.4.1). However, FISS then performs the backward computation in a dramatically different way. Specifically, FISS first evaluates all pattern queries in $\mathcal{Q}$ to locate affected events. Suppose the inferred

OID of e is ¡$O_{x_1} : p_1, ...O_{x_k} : p_k$¿, where p_k denotes the confidence, and the event type of e is Exit. During the query evaluation, FISS only checks those Enter events that fall into at least one partition of $\{O_{x_1}$, ..., $O_{x_k}\}$. This evaluation makes heavy use of partitioned hyQs. Namely, for a pattern query Q, its NFA helps to restrict the event type and the temporal order between events. Also its partitioned hyQs enable the equivalence test on probabilistic OID to be efficient. The query evaluation returns historic events that are potentially affected by the new event e, while other events will not be considered for further backward computation (because they are guaranteed to be unaffected by our technology). Example 5.4 illustrates such query evaluation process.

Example 5.4 *In Figure 5.3, at time t=14, event exit(127,R_1, O_1) arrives. FISS follows the NFA to search Enter events that occurred before time 14. As we can see, all historic Enter events are stored in hyQs and partitioned by objects. So FISS only needs to check those events falling into the partition of object O_1, because exit(127,R_1,O_1) is associated with O_1. Event enter(122, R_1,?) is quickly detected, which is then determined to be an affected event. And then the evaluation for this particular query stops, because the "skip-till-next-match" clause of Q1 imposes that only the first matched event is returned, as explained in Sec. 5.5.3.*

Next, given an affected event, FISS utilizes the hyper-links to locate affected random variables immediately. Moreover, the pruning (Sec. 5.5.1) and finish-flag (Sec. 5.5.2) strategies are *incorporated* during the selection of affected random variables, so that those random variables that are definitely unaffected will be sifted out on the way. Finally, the backward probability computation will account for those affected random variables *only*.

Example 5.5 *In Figure 5.3, the backward and smoothing at time t=14 only considers Room R1's state s_1 and hallway state s_H, because they are affected variables of exit(127,R_1,O_1). All other room state variables are not ever accounted for backward computation.*

In summary, our proposed selective smoothing strategy achieves *scalability* by restricting the backward probability computations to a small scope of affected random variables. Most importantly, we do so in a timely fashion by leveraging the CEP technology. This enables FISS to offer the *most likely association* for non-ID-events in *near real-time*, even over a large volume stream.

5.6 Experimental Evaluation

We have implemented all proposed inference techniques in FISS using Java and we take the Active CEP framework [92] as our back-end CEP engine. In this section, we present a detailed evaluation of FISS using event streams modeled based on the real-world healthcare system HyReminder [93]. All measurements were obtained from a 1.3Ghz Intel Due-core processor with 4GB RAM running JRE 1.6.

Our experimental results demonstrate that our system (1) produces a probabilistic event stream with probabilistic object identifications for all events; (2) offers significant error reduction over the MHT [81] model, a state-of-the-art alternative model; (3) responds to high-volume streaming events within near-real-time on a moderate hardware platform; (4) provides on average 15 times faster processing time than the basic FB algorithm.

5.6.1 Experimental Setup

Motion tracing event streams. We make use of the data simulator for a hospital ICU scenario that produces sensor reading streams according to the real-world observations obtained in the HyReminder system [93]. Specifically, the simulator has a subroutine that generates a *single* healthcare worker's trace in the ICU, which consists of a sequence of pairs of `Enter` and `Exit` events. Based on our analysis of the real data from the

HyReminder system, the time interval of a worker staying at a patient room follows a Gaussian distribution. This subroutine is first executed for each worker separately. Then the simulator *merges* all workers' traces into one stream ordered in time.

Also, to obtain insight into key factors on accuracy and performance, we control several properties of the event stream. One crucial property is the "*non-ID-ed ratio*" of the input stream, meaning the percentile of events that do not have associated object identities. We implement this by randomly selecting a number of events in the stream, for which their `OIDs` are hidden. E.g., if the non-ID-ed ratio is set to 20%, then 20% events in the stream will have their `OID` missing. In this way, FISS cannot see the real `OIDs` of non-ID-ed events, while the simulator keeps a copy of all `OIDs` for later inference precision evaluation.

When we use the simulator to generate healthcare workers' traces, the experimental results below are the average over 20 runs of the stream per each particular setting.

Metrics. Typically, the accuracy of inference results is measured using the *precision* metric, i.e., the ratio of inferred values over the ground truth [20, 54, 89]. In our context, the precision of our inference algorithms can be measured as the ratio of the inferred `OID` over the real `OID` of a non-ID-ed event. For example, suppose the inferred `OID` is $<$ O_1=0.75, O_2=0.25$>$, and the ground truth is $<$ O_1=1.0$>$, then the precision is 0.75/1.0 = 75%. Note that we do not calculate precisions for ID-ed events, simply because no object association inference is done for ID-ed events.

The performance metrics are the processing time and the throughput of our system. The throughput is measured as the average number of events that our system takes to process in one unit time. Namely, given a batch of input events of size *numIn* (*numIn* is set to be much larger than the maximum window size of all queries), suppose the system time span taken to process the batch is *Tproc*, then the throughput = $numIn/Tproc$.

5.6.2 Alternative Approaches Compared

MHT. In order to demonstrate our proposed time-varying graphical model provides the desired inference precision, we compare with one of the most widely used approaches for the data association problem, namely the multiple hypothesis tracking (MHT) model [81]. MHT model maintains multiple possible tracks for each object, and updates every possible existing track when processing a new event. Over time, a track can branch into many possible directions. MHT calculates the probability of each potential track. Typically it only reports the most probable tracks because reporting all is too prohibitive [37]. The main difference between MHT and our proposed model is that MHT does not revise the previously inferred object associations in a track.

In the experiments we adopt an open-source Java implementation [1] of MHT based on the Murty best-k assignment algorithm [37]. The real-world constraints specified in the MHT implementation (in the form of ambiguity matrix and observed features [1]) are equivalent to the conditional probabilities aforementioned in Section 5.3.2. The MHT implementation is based on the best-k assignment which returns the k most probable tracks for an object at each epoch [37]. We have enumerated k from 100 to 5 and found that a reasonable k in our setting is k=12, which ensures to return results within an affordable time.

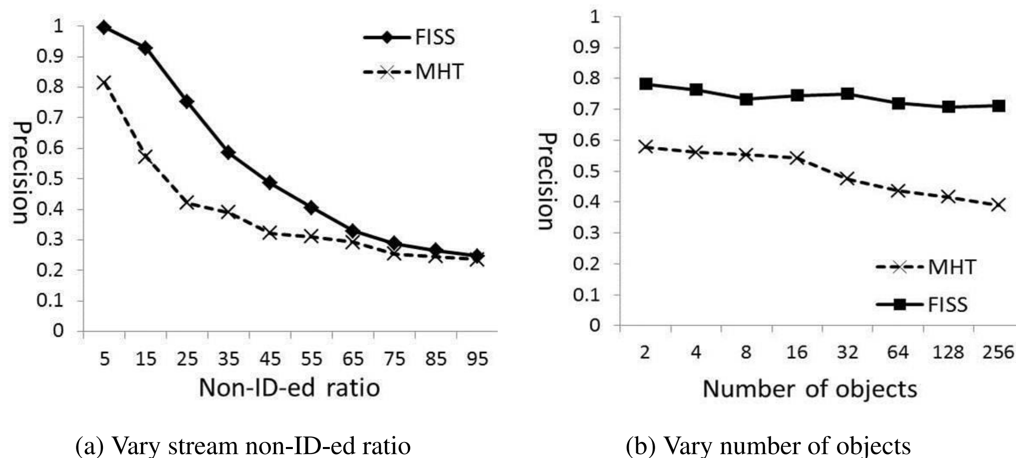
Basic FB Algorithm. We compare the performance of our proposed optimization techniques with the off-the-shelf Forward-backward algorithm. We make use of an open-source implementation of FB algorithm in Java [2], and extend the implementation to enable it to work with streaming data, as described in Section 5.4.1.

5.6.3 Experiments on Inference Accuracy

We first evaluate the accuracy of our inference method. Since our optimized inference algorithm and the basic FB algorithm implement the same model, i.e., use the same conditional dependencies specified in Section 5.3.2, they offer the same inference precision. Thus we only compare our proposed inference technique (marked as "FISS") versus MHT (marked as "MHT") below.

Non-ID-ed ratio vs. precision. We first test the sensitivity to the non-ID-ed ratio of the input stream. As Figure 5.5a shows, while both FISS and MHT produce worse inference accuracy as the non-ID-ed ratio increases, FISS outperforms MHT by 45% on average. Especially when the non-ID-ed ratio is low, i.e., 5% to 45%, FISS achieves 60% higher precision than MHT. This is mainly because the MHT model does not conduct smoothing over historical object identification associations, even when more information could be gained as new events arrive (especially ID-ed events).

Number of objects vs. precision. Next we vary the number of objects contained in the simulated stream for a fixed non-ID-ed ratio of 25%. Figure 5.5b reports the average precision results as the object count increases exponentially. We can see that neither of the models degrades significantly. The primary reason is the conditional probabilities specified for object association treat each object independent of each other, which is known as the "disjoint tracks constraint" commonly adopted in the PDA problem [20]. However, MHT scales less gracefully than our FISS approach, especially when the object number is larger than 32. Recall that the MHT implementation is an approximation based on best-k assignment [37], so when the number of objects increases, an non-ID-ed event naturally has more alternate objects that can be assigned to it, yet in MHT only a subset (with the fixed k size) of all possible assignments will be obtained. Such pruning consequently reduces the inference precision of MHT.



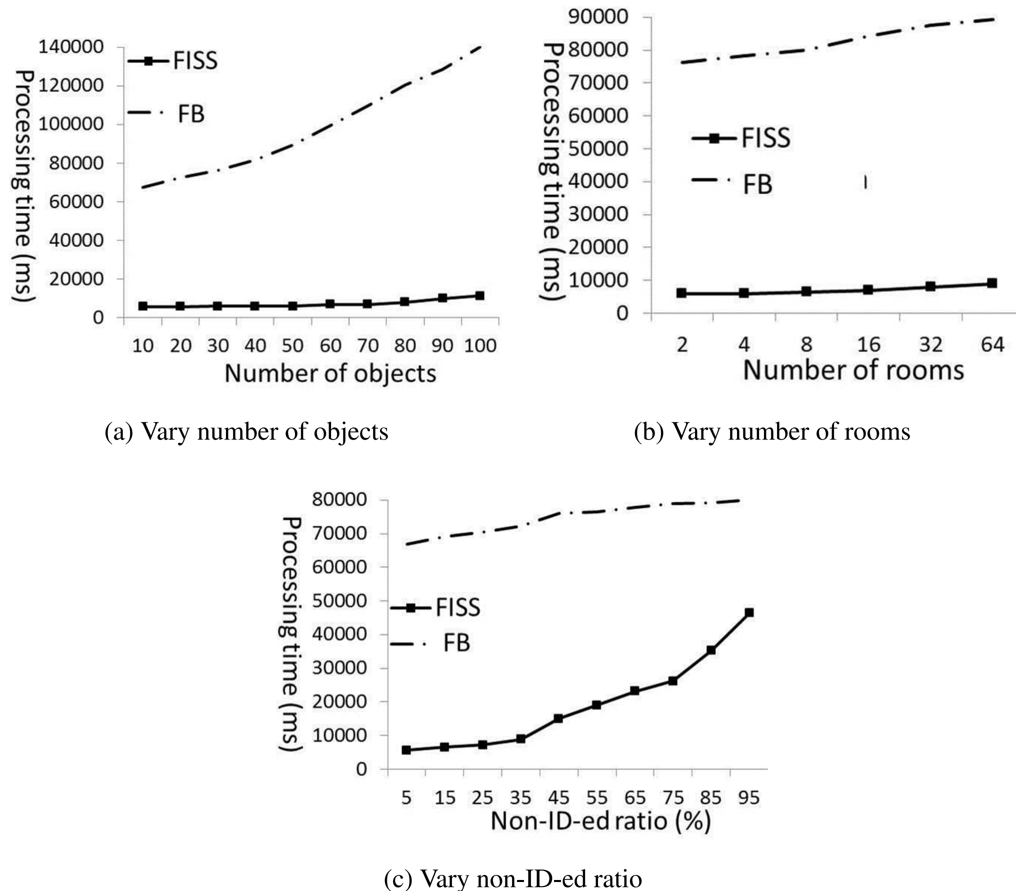
(a) Vary stream non-ID-ed ratio (b) Vary number of objects

Figure 5.5: Comparison on Inference Accuracy

5.6.4 Experiments on Inference Efficiency

Though our optimized inference algorithm and the basic FB algorithm provide the same inference precision, their performances are significantly different. Next we compare the processing time of these two algorithms for consuming the same chunk of the input stream (of 2000 events). The following three experiments (Figures 5.6 (a), (b) and (c)) convey that our optimized inference algorithm (denoted by "FISS") results in a dramatically less processing time than the basic FB algorithm (denoted by "FB") – while FISS only needs several *thousands* of milliseconds to process, the basic FB approach requires several *millions*, i.e., 15-fold faster.

Number of objects vs. processing time. In this experiment we vary the number of objects observed in the input stream. The non-ID-ed ratio of the input stream is 25% and number of rooms is 10, with a smoothing window of 60 minutes. As can be seen from Figure 5.6a, FISS offers a significantly better processing time than the basic FB. Namely, FISS processes the input stream on average 15 times faster. Moreover, we observe that FISS is not very sensitive to the number of objects. The main reason is FISS partitions events on object ID. Even though the total number of objects increases, only the related



(a) Vary number of objects (b) Vary number of rooms

(c) Vary non-ID-ed ratio

Figure 5.6: Processing Time on Stream of 2000 Events

objects to an event, which typically is fairly stable within a given time period, need to be considered for computation. On the other hand, as explained in Section 5.4.2, the basic FB algorithm is linear in the number of possible states of a random variable [74], which in our context is exponential in the number of objects.

Number of rooms vs. processing time. Next we observe the processing time of FISS and basic FB while varying the number of rooms in the environment. Figure 5.6b demonstrates that FISS achieves a stable performance when the room number increases. The key reason is FISS only computes backward probability for *affected* random variables, as stated in Lemma 5.4. This limits the scope of random variables that must be accounted for computation, which usually relates to one particular room and the hallway. On the other

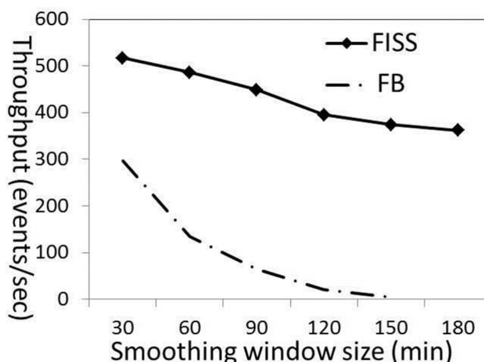


Figure 5.7: Smoothing window size vs. throughput

hand, the basic FB algorithm is linear in the room number, as it always revisits every random variables during the backward step and more rooms means more variables.

Non-ID-ed ratio vs. processing time. Figure 5.6c shows the processing time for increasing non-ID-ratios. FISS needs more processing time when the number of non-ID-ed events in the stream increases for three reasons: First, more non-ID-ed events ask for more forward inference computations; Second, an uncertain event will lead to more affected historical events compared to an ID-ed event; Three, as the number of accurate events decreases, fewer finish-flags can be placed. In contrast, the basic FB algorithm does not perform selective smoothing nor early termination. Its processing time is just slightly affected by the non-ID-ed ratio simply because it performs more forward inference computations to estimate missing `OID`s.

Smoothing window vs. throughput. Next we investigate how FISS and basic FB perform under various smoothing window sizes. In this experiment, the smoothing window varies from 30 minutes to 180 minutes. We set the non-ID-ed ratio to be 25%, the number of objects to be 32, which is realistic for an ICU. Figure 5.7 demonstrates that both FISS and basic FB degrade in terms of throughput as the smoothing window becomes larger. Because a larger smoothing window means more events must be processed during backward smoothing. But FISS is capable to scale even when the window size is

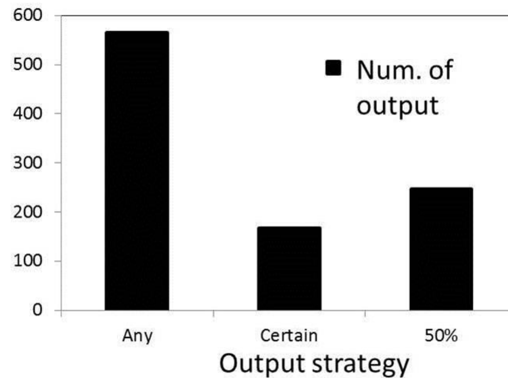


Figure 5.8: Output strategy vs. num. of output events

180 minutes, mainly thanks to the finish-flag technique. As explained in Section 5.5.2, the optimized backward computation terminates when either a finish-flag is reached or in the worst case, when events are out of the window. This experiment shows that even when the non-ID-ed ratio is relatively large, 25% in this experiment, many backward computations take advantage of finish-flags, thus resulting in much less processing time. In contrast, the FB algorithm conducts each backward revision until the window threshold. That is, its time complexity is polynomial in the number of events [74].

Output strategy vs. number of output events. In this experiment we test three commonly-used output strategies introduced in Section 5.2, namely reporting any change (denoted as "Any"), reporting when the object association is certain (denoted as "Certain") and reporting when the revision differs by more than 50% (denoted as "50%"). The input stream in this experiment contains 2000 events with 32 objects and the non-ID ratio is 25%. Figure 5.8 depicts the number of output events, including both ordinary events and revisions, for the three strategies. As we can see, among all changes made during smoothing, shown by the bar of "Any", only a small portion (namely 30%) of them are significant, shown by the bar of "Certain". We also observe that an input event could be modified more than once, resulting in multiple revisions.

5.7 Related Work

Probabilistic data association (PDA). Techniques for PDA determine the correspondence between measured observations and objects [20] when the association between them is uncertain. In typical PDA applications, like radar based object tracking and person tracking in videos, the input observations *never* carry any object identification. This is the key difference from our target application, where the observations are mixed with ID-ed and non-ID-ed events. Consequently, models and methods from this research area do not work very well for our problem. Our experiments demonstrate that if we adapt the widely-used PDA approach, MHT [81], to our problem, the association results are less accurate than those produced using our proposed FISS solution. Besides, existing work [20, 81, 88] of PDA largely focused on modeling, while the efficiency of processing has been overlooked - which is a key objective of our work.

RFID stream processing. Recent research has addressed the RFID location inference [28, 32, 89] and RFID data cleaning problems [45, 54]. In these problem settings, object identities are reliably given by RFID readings, i.e., no inference on object identification is needed. Instead, they focus on challenges like cleaning redundant readings and inferring objects' precise locations. Therefore they fundamentally tackle a different problem from us.

Furthermore, in their context, observations of an object are likely redundant, lossy or erroneous. Hence they chose *approximate* inference methods such as particle filtering [28, 32, 45, 54, 89], which maintains weighted samples about the true location of each object. In contrast, in our problem setting (described in Section 5.2), the information brought by input events are precise, in the sense that even though the identification of an event may be missing, all attributes (including time and location) of an event are accurate. Namely we do not need to handle redundant, lossy or erroneous event streams. We thus chose

an *exact* inference method, the Forward-backward algorithm, that fits our problem well. In spite of different models and inference methods, technically we have been inspired by several ideas of efficient inference over streaming data from their techniques, like limiting the scope of smoothing [54, 89] and customizing data structures to speed up the inference process [28, 89].

5.8 Conclusion

In this chapter we present a probabilistic approach to translate a stream consisting of ID-ed and non-ID-ed sensor readings into a probabilistic event stream, thus enabling object-based event analytics in real-time. We design a set of optimization strategies for inferring streaming events using the extended Forward-backward algorithm. The proposed strategies scale the backward probability computation while offering the most likely object association. Our experiments show that our proposed solution offers on average 45% better precision over the state-of-the-art PDA approach. Our optimized inference strategies achieve on average 15 times higher throughput than the basic Forward-backward implementation.

We are the first to adopt CEP techniques for probabilistic inference over event streams. We not only provide the methodology of transforming temporal dependencies involved in the inference problem into pattern queries (Section 5.5.3), but also experimentally show the dramatic performance gain offered by our CEP-aided optimizations (Section 5.6.4).

Our proposed optimization techniques, namely leveraging pattern queries for selectively smoothing and using finish-flags to early terminate backward computations, are general mechanisms for the Forward-backward algorithm. Even though these techniques were demonstrated in the context of our object identification inference problem, they could be applied to other problems wherever the Forward- backward inference is adopted.

Improving the performance of probabilistic inference algorithms with database principles is of independent interest to the probabilistic database community [26, 56, 78]. The generality of our techniques strengthens the contributions and warrants future studies.

Chapter 6

Conclusions and Future Work

6.1 Summary

The main goal of this dissertation is to introduce advanced services of Complex Event Processing (CEP) systems that are becoming increasingly important for many emerging stream-based applications. Specifically, we tackle the problems of the transactional processing and the privacy preservation inside CEP, and the problem of probabilistic inference powered by the CEP technology. The three highlights of this dissertation can be summarized as follows.

First, we identify the problem of concurrency control in stream execution, and our innovation of stream-transactions is the first attempt of introducing transactional concepts into stream environments. We further design three stream-transaction scheduling algorithms that achieve high responsiveness without compromising correctness. Our work has been implemented within a commercial CEP engine and utilized to support a real-world health care application. Based on our stream-transaction model, we also develop failure recovery strategies for transactional stream systems, which dynamically combines stream data backup and transaction logging.

Second, we are the first to study the problem of utility-maximizing event stream suppression with privacy preference. We formally prove that this problem is in general NP-Hard. We then design the Hybrid online event suppression algorithm, which tweaks the event type level decisions based on run-time context of the event stream. Our experiments using real-world and synthetic data show that our proposed approaches can preserve utility effectively and efficiently. We also develop novel algorithms to address the subproblem of pattern match cardinality estimation. We demonstrate that the periodicity-based approach is needed for accurate cardinality estimation in real world sensor reading data sets.

Third, we have built the FISS framework that efficiently transforms the raw stream of ID-ed and non-ID-ed events into a probabilistic stream of events with inferred object identifications. The key technical contribution of FISS is a suite of strategies for optimizing the performance of the Forward-backward inference algorithm. Most notable, we adopt CEP techniques for probabilistic inference over event streams. We not only provide the methodology of transforming temporal dependencies involved in the inference problem into pattern queries, but also experimentally show the significant performance gain offered by our CEP-aided optimizations.

6.1.1 Interrelationship of Proposed Techniques

Now let us consider the inter-relationships among the three proposed techniques. As depicted in Figure 6.1, we are capable to deal with two kinds of input: the ordinary event stream and the uncertain event stream (with non-ID-ed events). All of our techniques are implemented as optional components to a CEP engine.

We first consider the scenario of an ordinary event stream, and analyze the interrelationship between ACEP and Privacy-Preserving suppression. Recall that ACEP and the associated stream transaction technology functions directly in the kernel of a CEP query

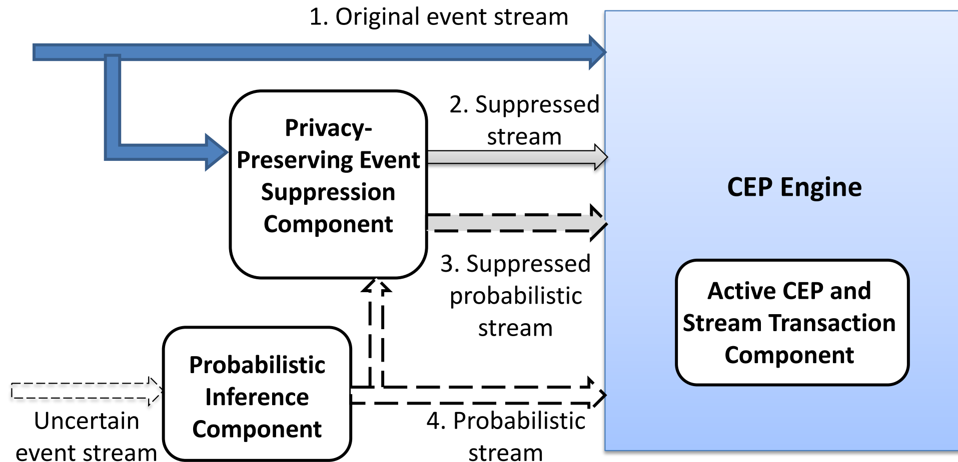


Figure 6.1: Interrelationship of Proposed Techniques

engine. If our Privacy-Preserving suppression technique is utilized, it first drops events and then the suppressed stream will be fed to the CEP engine. In other words, the suppression algorithm takes action before the event stream is evaluated by any pattern query. Hence the event suppression is transparent to ACEP. Even though ACEP may receive a suppressed stream if the privacy-preserving service is turned on, our proposed active rule and stream transaction techniques do not need to change at all to process a suppressed stream. In summary, our ACEP and privacy-preserving techniques can co-exist in a single CEP engine without further extension.

Next we analyze the scenario of an uncertain event stream. In this scenario we have to first use the Probabilistic Inference technique to transform the uncertain stream into a probabilistic stream. After gaining the probabilistic stream, we have three options depending on how we turn on/off each component: (1) consuming the probabilistic stream directly for CEP pattern matching; (2) feeding the probabilistic stream to ACEP; (3) using the probabilistic stream for privacy-preserving suppression and then feeding the suppressed probabilistic stream to ACEP. These alternative input streams for a CEP system is also illustrated in Figure 6.1. Therefore, we want to figure out the following questions.

First, how an ACEP system deals with a probabilistic event stream? Second how to apply the privacy-preserving technique to a probabilistic event stream?

For the first question, when the ACEP technique works on a probabilistic event stream, we expect the following changes are needed: First, the shared store, i.e., a static relation, will contain uncertain tuples. Nowadays uncertain/probabilistic databases have been widely studied in the community [38]. We would then adopt these well-established techniques to maintain an uncertain table and support relational operations on it. Second, pattern queries will produce probabilistic pattern matches. Namely, a pattern match is associated with a confidence value to denote the probability of its occurrence. Third, the rule triggering condition will take probabilistic pattern matches into account. Specifically, the rule triggering condition would also specify the threshold of the confidence of a pattern match. In practice, if an active rule's action will be visible by the outside word, we would choose to set the threshold in a conservative fashion. For example, in the HyReminder application, the active rule that may change a HCW's status to "warning/violation" should have a high confidence threshold. Because we want to avoid a false alarm which certainly annoys the user. In summary, the above three changes essentially accord with the probabilistic event model. But our notion of stream-transaction, execution correctness and concurrency control algorithms will equally work.

For the second question, our utility-based suppression decision-making process does not need to change, while the pattern match cardinality estimation does need to revise. In particular, our arrival rates based estimation needs to collect the arrival rates on an event-type basis, and our periodicity based estimation requires to mine the periodical patterns over time. The new challenge is basically to gather the statistics on a probabilistic event stream. This in general is a hard problem, which I would explore in future work. But a quick-fix would be to use a hard-coded threshold to pre-process the sampled stream. Namely, a probabilistic event with a confidence above the threshold will be treated as

a regular event, while the one with a confidence below the threshold will be discarded. Apparently further research is needed to cope the stream statistics with a “possible world” semantics, and then our cardinality estimation formulas should take such probabilistic statistics into account and render most-likely estimates.

6.2 Future Research Directions

Next we describe the possible directions for future research based on the concepts presented in this dissertation.

6.2.1 On-line Sensor Anomaly Detection

During the construction of the HyReminder system at Univ. of Massachusetts Memorial Hospital, we found that besides the requirements toward CEP’s functionalities that we have addressed in this dissertation, the quality of sensor event data is also a critical issue that needs to be carefully handled. For example, we observed that a sensor may generate a large number of garbage readings when it is low in battery. In this case, if we directly took these readings for hygiene compliance monitoring, the monitoring results would be wrong or unreliable. Therefore, we have to detect and deal with sensor anomalies before we feed the sensor data for real-time analytics.

Anomaly or outlier detection has been an active area of computer science research (see [29, 76] for a recent survey). Popular anomaly detection approaches include classification based detection (using neural networks and SVM [31, 50, 60]), clustering based detection (using distance based clustering [53]) and statistical detection (using stochastic models [14]). Most of these machine learning based algorithms are computation-intensive and function off-line. But our goal is to discover abnormal sensor behaviors over increasingly high-volume event streams within an actionable time.

A natural idea is to resort to CEP for performance boosting, like we successfully practiced in Chapter 5. Namely, we wish to improve the efficiency of the above outlier detection method(s) with streaming data customization and on-line pattern matching techniques. Specifically, we expect to peruse the following directions.

- Early filtering. Similar to the "finish-flag" technique proposed in Section 5.5, it is beneficial to quickly filter out some sensors which are definitely abnormal or definitely normal by applying proper intuitive judgments. One can think about such heuristic filtering as a "weak leaner", which represents an intuitive classification rule that people acquire from the real-world application. For example, if a sensor does not generate heartbeat events periodically, then this sensor is likely to have some problems. We can report this sensor as an abnormal one even before conducting any complicated outlier detection algorithms as described above.
- Partition and parallel processing. Borrowing the idea of Partitioned Active Instance Stack (PAIS) [98] from the modern CEP engines, we can partition sensor readings on the qualification attribute of events. For example, we may want to partition on sensor ID, physical location, sensor type, etc. And then the computation within each partition can be executed in parallel.
- Exploring temporal relationship of events. In online anomaly detection tasks, a common underlying logic is to explore the temporal correlations among events. For example, once a door sensor generates an `Enter-room` event of a doctor, this must be followed by an `Exit-room` event. We can also learn the distribution of the time-span that a person spends in the room, so that if there is not a matched `Exit-room` event is observed within a reasonable time, or there is more than one such `Exit-room` event, then we can trigger an anomaly alarm. As we show in Section 5.5, the CEP technology is good at tracking such temporal correlations

among events. Therefore we could leverage the CEP pattern matching technology to speed up such anomaly detection tasks.

6.2.2 Revisiting Alternative Stream Transaction Models

Besides of our ACEP model, two alternative models for transaction support in stream management systems are also available. First, many stream management systems use feedback streams (also called internal streams) and punctuations to support the interaction between queries [27, 30, 71]. We found that such mechanisms can be customized for stream transaction execution, if we transform relations into internal streams. We henceforth call this transactional stream model as *Stream-Oriented Model*. Second, researchers in [24] proposed the *Unified Transaction Model* as an alternative to our ACEP model for stream transaction management. Below we briefly summarize these two alternative models using the HyReminder application as a running example (Section 3.3). We will explain how each model works in terms of *data model*, *query model* and *transaction management*.

(1) Stream-Oriented Model.

Data Model. Stream-Oriented model transforms a relation into a stream. According to the classical definition [15], a stream is an unbounded set of tuples ordered by time, on which the only possible update operation is `APPEND`.

For illustration, in the HyReminder example (Section 3.3), we could define an internal stream to represent the relation of HCWs' hygiene compliance status, named `S_HCWstatus`. The schema of this stream is `(HCW-ID, Timestamp, Status)`, which represents a HCW's status at a certain timestamp. Compared to the original relation, this stream has an extra `Timestamp` attribute. The reason is that there is no relational "update" operation on a stream. Whenever a HCW's status changes, a new event will be appended to this internal stream. Hence we need to maintain the application timestamp for every event in

the stream. In summary, this is a straightforward transformation from the static relation to a potentially infinite stream.

Query Model. In order to model that multiple queries will concurrently update the internal stream, we extend a stream processing system with the following functionality: allow multiple queries to publish their results into one single internal stream.

We now re-define query Q1 (specified in Section 3.3). In the Stream-Oriented model, Q1 takes two input streams: the original input stream as well as the internal stream `S_HCWstatus`. Semantically, for every input event e_i with a given HCW_ID, Q1 checks whether the HCW is currently in green status. This checking is performed on the internal stream `S_HCWstatus`. Specifically, the latest status (i.e., with a largest timestamp) of the HCW is extracted from `S_HCWstatus`. If the checking returns "true", Q1 keeps e_i for further evaluation – to detect the pattern `SEQ(Exit, !Wash, Enter)`. If eventually a match for a certain HCW_ID is detected by Q1, this HCW's status should be updated to yellow. That is, Q1 will append a new event, $(HCW_ID, timestamp, Yellow)$, into the internal stream `S_HCWstatus`.

Transaction Management. Special concurrency control has to be enforced to accesses and updates on the internal stream, otherwise read-too-late and write-too-late anomalies may arise. For example, when Q1 tries to read `S_HCWstatus` for checking HCW007's status at time 15, the event $(HCW007, 12, Green)$ should have been already inserted by Q2. However, Q2 in fact inserted such event after Q1's checking. In this case, Q1 should have read the information that "HCW007 was on green status at time 15" but read some other value instead.

For illustration, we assume that the correctness of execution can be interpreted as all `READ` and `APPEND` operations are ordered according to the triggering events' application timestamp. To ensure the correctness, in the Stream-Oriented model, a concurrency control mechanism would be to utilize special *punctuations* [30] to synchronize the stream

processing. Basically a punctuation specifies that all operations corresponding to the events with smaller timestamp than the punctuation have finished. In other words, if Q1 wishes to check HCW007's status at time 15, it has to wait till the punctuation on S_HCWstatus notifies that the current time of S_HCWstatus has advanced to 15.

(2) UTM Model.

Data Model. The UTM model treats all the data sources as sets of data items on which READ and WRITE are executed. As a result, a relation is simply an unordered set of tuples sharing the same schema. A stream is a possibly infinite partially ordered set of events, where an event can be interpreted as a tuple with a special ordering on time attribute. Each event arrival is converted to a WRITE operation on the input stream. In our running example, the input stream is represented as a relation named T_InputEvents.

Query Model. UTM has one-time queries only. Therefore a continuous query is represented as an infinite sequence of one-time queries which are fired as new event arrives. For example, when an event $Enter^{15}(HCW007, RoomI)$ triggers the execution of Q1, Q1 reads all tuples in the relation T_InputEvents within the time window, processes the sequence pattern matching and then possibly updates the relation S_HygieneStatus with a new state for HCW007.

Transaction Management. The traditional definitions of transaction, conflict and serializability can be reused in UTM without change [23]. Then SS2PL (Strong Strict 2-Phase Locking) protocol can be adopted for run-time transaction scheduling. Specifically, *locks* are added to T_InputEvents. For example, when Q1 intends to read(write) T_InputEvents, it needs to request a read(write) lock first. When a transaction ends, the locks held by this transaction are released.

Comparing Alternative Models.

A key selling point of the Stream-Oriented model and UTM is that they treat all data and operations uniformly. The Stream-Oriented model can conduct pure streaming non-

blocking process, which has been demonstrated to be fast and suitable for applications that need near real-time responsiveness. On the other hand, UTM utilizes the sole relational model so that all conventional transaction management techniques established for RDBMS can be applied without any change. Also, when defining a transaction, keeping a single data model with corresponding operations can straightforwardly resort to the classical transaction definition – a partially ordered sequence of homogeneous operations.

However, such uniform processing sacrifices the nature of either a push-based or a pull-based execution paradigm. Specifically, in the Stream-Oriented model, after transforming a relation into a stream, the user cannot use the well-known SQL to access or update the relation any more. Moreover, the punctuation based synchronization is more ad-hoc compared to the well-established lock based concurrency control protocols in RDBMS. On the other hand, in UTM a continuous query has to be transformed to a set of one-time queries. This mitigates the key benefits of a continuous query: non-stop monitoring and efficient one-pass analysis. Executing a huge number of one-time queries inevitably increase the processing time.

In contrast, our ACEP model tries to keep the nature of push-based and pull-based query processing by embracing both relations and streams into a single model. In ACEP, a transaction may be composed by heterogeneous operations, i.e., an operation can be either a `READ`/`WRITE` on a relation or a `READ`/`APPEND` on a stream. This sounds more complicated than the classical definition of a transaction. But in fact, once we group all operations triggered by an input event into such a transaction, many traditional concurrency control algorithms (e.g., SS2PL) can be re-used. Another benefit of the ACEP model is that a relation or a stream can keep using its original query and execution model, which is just right for the particular data. In order words, there is no need to convert a relation to a stream or transform a stream to a relation. Even better, the user can use CEP language or CQL to define a continuous query, while freely embedding SQL sub-queries

to access static relations.

Furthermore, the Low-water-mark transaction scheduling algorithm in ACEP can be viewed as a hybrid of using locks and punctuations. Specifically, the locks keep track of the access on a relation, while the low-water-mark (which can be viewed as a punctuation) maintains the time increment of the table and then grants locks.

In summary, it would be interesting and promising to conduct an experimental performance comparison for these three alternative models. We hope the experimental evaluation can confirm our analytic results above. Eventually, future work in this direction will provides a useful guidance of choosing the stream transaction model for real-world applications.

Bibliography

[1] Implementation of the murty algorithm to obtain the best k assignments. http://code.google.com/p/java-k-best/.

[2] Java implementation of algorithms from ”artificial intelligence - a modern approach 3rd edition”. http://code.google.com/p/aima-java/.

[3] Microsoft StreamInsight: http://www.microsoft.com/sqlserver/2008/en/us/r2-complex-event.aspx.

[4] OptimJ, a java-based language for optimization: ateji.com/optimj.

[5] Streambase cep product. `http://www.streambase.com/`.

[6] Sybase coral 8 engine. `http://www.sybase.com/products`.

[7] U.s. centers for disease control and prevention. `http://www.cdc.gov/ncidod/dhqp/hai.html`.

[8] Wavemark system: Clinical inventory management solution www.wavemark.net/healthcare.action.

[9] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. SIGMOD, pages 34–48, 1987.

[10] O. Abul, F. Bonchi, and F. Giannotti. Hiding sequential and spatiotemporal patterns. In *IEEE Trans. Knowl. Data Eng.*, volume 22, pages 1709–1723, 2010.

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

[12] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. *SIGMOD Rec.*, pages 59–68, 1992.

[13] M. Ali, C. Gerea, B. S. Raman, and et al. Microsoft cep server and online behavioral targeting. *VLDB Demo*, pages 1558–1561, 2008.

[14] F. Anscombe and I. Guttman. Rejection of outliers. *Technometrics*, (2):123–147.

[15] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. In *IEEE Data Engineering Bulletin*. Springer, 2003.

[16] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. SIGMOD, pages 261–272, 2000.

[17] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, 2002.

[18] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of SIGMOD*, 2003.

[19] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30:109–120, 2001.

[20] Y. Bar-Shalom, editor. *Tracking and Data Association.* Academic Press Professional, Inc., 1987.

[21] R. Barga, J. Goldstein, M. Ali, , and M. Hong. Consistent streaming through time: A vision for event stream processing. *CIDR*, pages 363–374, 2007.

[22] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. *VLDB*, 2006.

[23] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Elsevier, second edition, 2009.

[24] I. Botan, P. Fischer, D. Kossmann, and N. Tatbul. Transactional stream processing. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 204–215. ACM, 2012.

[25] J. M. Boyce and D. Pittet. Guideline for hand hygiene in healthcare settings. *MMWR Recomm Rep.*, 51:1–45, 2002.

[26] H. C. Bravo and R. Ramakrishnan. Optimizing mpf queries: Decision support and probabilistic inference. *SIGMOD*, 2007.

[27] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *ACM SIGMOD international conference on Management of data*, pages 1100–1102, 2007.

[28] Z. Cao, C. Sutton, Y. Diao, and P. J. Shenoy. Distributed inference and query processing for rfid tracking and monitoring. *PVLDB*, 4:326–337, 2011.

[29] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection. *ACM Computing Surveys*, 41(3):1–58, 2009.

[30] B. Chandramouli, J.Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *VLDB*, pages 241–252, 2009.

[31] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:1–27, 2011.

[32] H. Chen, W.-S. Ku, H. Wang, and M.-T. Sun. Leveraging spatio-temporal redundancy for rfid data cleansing. SIGMOD, pages 51–62, 2010.

[33] M. Cherniack and et al. Scalable distributed stream processing. *CIDR*, 2003.

[34] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein. Entirely declarative sensor network systems. In *VLDB*, 2006.

[35] G. Cormode and M. N. Garofalakis. Sketching probabilistic data streams. *SIGMOD*, pages 281–292, 2007.

[36] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Optimal sampling from distributed streams. In *Proceedings of PODS*, 2010.

[37] I. Cox and S. Hingorani. An efficient implementation of reid's multiple hypothesis tracking algorithm and its evaluation for the purpose of visual tracking. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 18(2), 1996.

[38] N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[39] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proceedings of SIGMOD*, 2003.

[40] U. Dayal, B. T. Blaustein, A. P. Buchmann, and et al. The hipac project: Combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.

[41] N. Dindar, B. Guc, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: declarative pattern matching over live and archived streams of events. *SIGMOD*, pages 1023–1026, 2009.

[42] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Lecture Notes in Computer Science*, 2006.

[43] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid. Stagger: Periodicity mining of data streams using expanding sliding windows. In *Proceedings of ICDM*, 2006.

[44] M. L. et al. E-cube: Multi-dimentional event sequence processing using concept and pattern hierarchies. *ICDE Demo*, pages 1097–1100, 2010.

[45] M. J. Franklin, S. R. Jeffery, and et al. Design considerations for high fan-in systems: The hifi approach. *CIDR*, pages 290–304, 2005.

[46] A. Gkoulalas-Divanis and G. Loukides. Revisiting sequential pattern hiding to enhance utility. In *ACM SIGKDD*, pages 1316–1324, 2011.

[47] M. Goetz, S. Nath, and J. Gehrke. Maskit: Privately releasing user context streams for personalized mobile applications. In *Proceedings of SIGMOD*, 2012.

[48] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[49] C. Gupta, S. Wang, I. Ari, M. Hao, U. Dayal, A. Mehta, M. Marwah, and R. Sharma. Chaos: A data stream analysis architecture for enterprise applications. *CEC*, pages 33–40, 2009.

[50] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier Detection Using Replicator Neural Networks. *Neural Networks*, pages 170–180, 2002.

[51] Y. He, S. Barman, and J. Naughton. Preventing equivalence attacks in updated, anonymized data. In *Proceedings of ICDE*, 2011.

[52] Y. He, S. Barman, D. Wang, and J. Naughton. On the complexity of privacy-preserving complex event processing. In *Proceedings of PODS*, 2011.

[53] Z. He, X. Xu, and S. Deng. Discovering cluster-based local outliers. *Pattern Recognition Letters*, 24(9-10):1641–1650, 2003.

[54] S. R. Jeffery, M. Franklin, and et al. An adaptive rfid middleware for supporting metaphysical data independence. *VLDB Journal*, 2007.

[55] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *Proceedings of CIKM*, 2003.

[56] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. *ICDE*, 2008.

[57] K. Kenthapadi, N. Mishra, and K. Nissim. Simulatable auditing. *Proceedings of PODS*, 2005.

[58] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.

[59] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Incognito: efficient full-domain k-anonymity. In *Proceedings of SIGMOD*, 2005.

[60] K.-l. Li, H.-k. Huang, S.-f. Tian, I. Technology, C. Science, and I. Detection. Improving one-class SVM for anomaly detection. *Machine Learning*, (November):2–5, 2003.

[61] M. Li, M. Mani, E. A. Rundensteiner, D. Wang, and T. Lin. Interval event stream processing. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 35:1–35:2, New York, NY, USA, 2009.

[62] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Proceedings of ICDE*, 2007.

[63] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. *ICDE*, pages 784–795, 2009.

[64] 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 Conference*, pages 889–900, 2011.

[65] T. Liu, P. Bahl, and I. Chlamtac. Mobility modeling, location tracking, and trajectory prediction in wireless atm networks. *IEEE Journal on Selected Areas in Communications*, 1998.

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

[67] H. J. Loether and D. G. McTavish. *Descriptive and inferential statistics, an introduction*. 1993.

[68] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *Proceedings of ICDE*, 2006.

[69] A. S. Maskey and M. Cherniack. Replay-based approaches to revision processing in stream query engines. In *Scalable Stream Processing System*, pages 3–12, 2008.

[70] Y. Mei and S. Madden. Zstream: A cost-based query processor for adaptively detecting composite events. *SIGMOD*, pages 193–206, 2009.

[71] R. F. Moctezuma, K. Tufte, and J. Li. Inter-operator feedback in data stream management systems via punctuation. *CIDR*, 2009.

[72] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17, 1992.

[73] R. Motwani, S. U. Nabar, and D. Thomas. Auditing SQL queries. In *Proceedings of ICDE*, 2008.

[74] K. P. Murphy. Dynamic bayesian networks: Representation, inference and learning. *Doctor of Philosophy Dissertation*, 2002.

[75] R. Nehme, E. Bertino, and E. A. Rundensteiner. A security punctuation framework for enforcing access control on streaming data. In *Proceedings of ICDE*, 2008.

[76] K. Ni, N. Ramanathan, M. Chehade, L. Balzano, S. Nair, S. Zahedi, G. Pottie, M. Hansen, and M. Srivastava. Sensor network data fault types. *Transactions on Sensor Networks*, 5(3), 2009.

[77] D. Nicklas, M. Grossmann, and T. Schwarz. Nexusscout: An advanced location-based application on a distributed, open mediation platform. In *VLDB*, 2003.

[78] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *VLDB*, pages 373–384, 2011.

[79] N. W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[80] C. Ré, J. Letchner, M. Balazinksa, and D. Suciu. Event queries on correlated probabilistic streams. *SIGMOD*, pages 715–728, 2008.

[81] D. B. Reid. An algorithm for tracking multiple targets. *IEEE Trans. on Automatic Control*, pages 843–854, 1979.

[82] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[83] J. Singh, P. Brandao, and J. Bacon. Context-aware disclosure of health sensor data. In *Proceedings of Pervasive Computing Technologies for Healthcare*, 2010.

[84] U. Srivastava and J. Widom. Flexible time management in data stream systems. *SIGMOD*, pages 263–274, 2004.

[85] R. E. Steuer. *Multiple Criteria Optimization: Theory, Computations, and Application*. 1986.

[86] L. Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10, 2002.

[87] J. Tan. Inapproximability of maximum weighted edge biclique and its applications. In *Proceedings of TAMC*, 2008.

[88] M. A. Tinati and T. Y. Rezaii. Multi-target tracking in wireless sensor networks using distributed joint probabilistic data association and average consensus filter. In *International Conference on Advanced Computer Control*, pages 51–56, 2009.

[89] T. Tran, C. Sutton, R. Cocci, Y. Nie, Y. Diao, and P. Shenoy. Probabilistic inference over rfid streams in mobile environments. *ICDE*, pages 1096–1107, 2009.

[90] D. Wang, Y. He, E. Rundensteiner, and J. Naughton. Utility-maximizing event stream suppression. *SIGMOD*, 2013.

[91] D. Wang, E. Rundensteiner, R. Ellison, and H. Wang. Active complex event processing infrastructure: Monitoring and reacting to event streams. *2011 IEEE 27th International Conference on Data Engineering (ICDE) Workshops*, pages 249–254, 2011.

[92] D. Wang, E. Rundensteiner, and R. Ellison III. Active complex event processing over event streams. *Proceedings of the VLDB Endowment*, 4(10):634–645, 2011.

[93] D. Wang, E. Rundensteiner, H. Wang, and R. Ellison III. Active complex event processing: applications in real-time health care. *Proceedings of the VLDB Endowment*, 3(1-2):1545–1548, 2010.

[94] D. Wang, E. A. Rundensteiner, R. T. Ellison, and H. Wang. Probabilistic inference of object identifications for event stream analytics. *EDBT*, 2013.

[95] F. Wang, S. Liu, and P. Liu. Complex rfid event processing. *VLDB Journal*, pages 913–931, 2009.

[96] T. Wang and L. Liu. Butterfly: Protecting output privacy in stream mining. In *Proceedings of ICDE*, 2008.

[97] J. Widom. The starburst rule system: Language design, implementation and applications. *IEEE DE Bull.*, 15(4), 1992.

[98] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over stream. *SIGMOD*, pages 401–418, 2006.

[99] X. Xiao and Y. Tao. m-invariance: Towards privacy preserving re-publication of dynamic datasets. In *Proceedings of SIGMOD*, 2007.

[100] Y. Xing, J.-H. Hwang, U. etintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. *Proceedings of VLDB*, 2006.

[101] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of MICRO*, 1991.

[102] B. Zhou, Y. Han, J. Pei, B. Jiang, Y. Tao, and Y. Jia. Continuous privacy preserving publishing of data streams. In *Proceedings of EDBT*, 2009.