Optimising Complex Event Queries over Business Processes using Behavioural Profiles

Matthias Weidlich[1], Holger Ziekow[2], Jan Mendling[2]

[1] Hasso Plattner Institute, Potsdam, Germany
matthias.weidlich@hpi.uni-potsdam.de
[2] Humboldt-Universität zu Berlin, Germany
holger.ziekow|jan.mendling@wiwi.hu-berlin.de

Abstract. Complex event processing emerged as a technology that promises tight integration of business process management with the flow of products in a supply chain. As part of that, complex event querying is used to monitor and analyse streams of events. The amount of data that needs to be processed along with the distribution of the event-emitting sources impose serious challenges for efficient event querying mechanisms. In this paper, we assume that the business process to which the events relate is defined in terms of a normative process model. Based thereon, we show how this knowledge can be leveraged to optimise complex event queries and their processing. To this end, we use the formal concept of behavioural profiles as a behavioural abstraction of the process model.

Key words: Complex Event Processing, Query Optimization

1 Introduction

Traditional business process management has put a strong emphasis on business process design based on both conceptual and executable models. The latter are used in workflow management systems for process automation, mostly in a rather narrow organizational setting. Recent technological innovations including information systems standards for RFID applications offer the chance of a much tighter coupling of a business process management system with the physical flow of goods in a supply chain, and event-based systems play an important role in tying informational process and object flow closer together.

In order to monitor and analyse the event streams produced by event-based systems, mechanisms to query complex events are of particular importance. In this paper, we consider an information system environment in which events are recorded at different distributed locations. This is a scenario that we frequently encountered in prior case studies in the manufacturing sector. The challenge in such scenarios is to find a strategy to handle querying in an efficient manner. Here, most research centres around the advantages and drawbacks of push and pull strategies in minimizing event propagation traffic.

Our approach for optimisation of complex event querying builds on the assumption that a normative process model is available. Such a process model

captures external knowledge about the potential sequence of events, which can be utilized for optimisation. The contribution of this paper is an approach that leverages such external knowledge based on the formal concept of behavioural profiles. Such a behavioural profile can be efficiently calculated from a process model. It covers constraints about the execution sequence, mutual exclusion, and potential concurrency of activities specified in a business process model. Our approach uses this knowledge to optimise query processing. Therefore, we rely on the accuracy of this knowledge and focus on expected events of the process.

The remainder of the paper is structured accordingly. Section 2 introduces basic terminology and concepts of complex event processing. Section 3 discusses the process knowledge that is leveraged in our approach. Based thereon, Section 4 introduces rules for optimisation based on behavioural profiles of processes. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Complex Event Processing

In this section, we present the background of our work in terms of complex event processing. We introduce some basic terminology and concepts, including an event model and a syntax for a pseudo query language and execution plans that we use throughout this paper.

Complex Event Processing Event processing refers to continuous real-time processing of data items (events) as they enter an IT system [7]. This is in contrast to traditional data bases where queries run on an ad-hoc basis over stored data. Event processing systems store event queries and continuously evaluate them as new events arrive. Application examples are manifold and include monitoring financial stocks, supply chain activities, or production processes [8, 15]. The term *complex event* refers to events that are defined trough more than one input event. For instance, one may define the complex event of a correctly finished process by a sequence of events about corresponding process activities. Complex event processing (CEP) is the process of detecting complex events.

Event model Most event models define an event as a tuple containing a unique ID, a type, a timestamp, and a set of attributes. In the context of business processes, events typically reflect the execution of an activity in a certain process instance. For the discussion in this paper, we build on this simple model and assume that the type denotes the process activity which caused the event. We further assume that each event holds an attribute $Case_{ID}$ that has the same value for all events generated in the same process instance. This basic model is in line with existing models for monitoring business processes, such as the EPCglobal standard for RFID events [6]. Further on, we use capital letters to denote events of a certain type and '.' to denote the attribute of an event (e.g.: A.ID for the ID of an event of type A).

Complex Event Queries Several approaches to phrasing and executing complex event queries exist. Dedicated query languages for complex event processing provide means to correlate, filter, and transform events from several sources. These languages typically use an SQL like syntax, follow a Pattern Condition Action (PCA) structure, or provide support for both. Without loss of generality, we use a simplified PCA based language in this paper.

The patterns in PCA based queries define relations between events that together cause an event of interest. The sequence operator SEQ and the logical operators AND, OR, and NOT are common operators for defining event patterns. $SEQ(A, B)$ matches if event A is followed by event B, $AND(A, B)$ if A and B occur in any order, $OR(A, B)$ if A or B occur, and $NOT(A)$ if A does not occur. Note that events A and B can be events or event patterns. This enables nested constructs such as $SEQ(A, OR(B, C))$ that matches sequences AB or AC as well as queries for long event sequences.

The constraint part in PCA based queries defines conditions on event attribute values that must hold in matching event sequences. These constraints resemble the *where* clause of SQL queries and compare attribute values of different events or attribute values against constants. To monitor events of a certain process instance one must query event sequences with the same value for $Case_{ID}$. In this paper, we assume this condition to always be in place without explicit mentioning. That is, the query $SEQ(A, B)$ matches only event sequences AB where $A.Case_{ID} = B.Case_{ID}$ holds true.

The action part in PCA based queries defines which action the system should trigger if the defined event queries matches. This is often issuing of a notification or triggering of some actuator. We focus on optimisations for detecting complex events and, therefore, do not discuss the action part in more detail, which is not affected by our contribution.

Execution model In this paper, we consider execution of complex event queries based on state machines. We choose this model for the sake of an intuitive illustration and because state machines (or variations of state machines) are widely used in complex event processing [4, 11, 17]. However, our approach to query optimisation may also be applied to other execution models.

The pattern part of PCA based complex event queries intuitively translates to transitions in state machines. The event types in a query define the input alphabet of the machine. A query for a single event A is realized by a transition on A from the start state to the final state. Queries for complex event sequences are built by concatenating state machines for detecting single events.

Our solution builds upon two important extensions of the simple state machine based model. One extensions is constraint evaluation along state transitions as proposed in [17]. In particular we assume that the state machine performs equivalence checks on the $Case_{ID}$ along transitions. That is, in a query $SEQ(A, B)$ the machine transitions on any event A but only transitions further if an event B with $A.Case_{ID} = B.Case_{ID}$ occurs. This optimisation avoids extracting irrelevant events sequences that do not belong to the same processes case (those sequences would have to be filtered out later on).

The other extension that we build upon concerns the employed communication paradigm. Event processing typically builds on push-based communication, often realized with a publish/subscribe mechanism [10]. Thus, events must be processed when they occur. While this scheme is appropriate in many applications of CEP, it is unnecessarily restrictive for monitoring business processes. Many business applications keep records of event data in transaction logs. This allows pulling (some) events from logs and processing them later. Akdere et al. exploit this by proposing plan based execution of complex event queries [2]. The plan based approach combines push-based and pull-based communication.

For illustration consider the query $SEQ(A, B)$. With push based communication the processing machine subscribes for events A and subsequently waits for corresponding events B. In a hybrid model, the processing machine can subscribe to events B and then pulls corresponding events A from the transaction log (with $A.timestamp < B.timesamp$ and $A.Case_{ID} = B.Case_{ID}$). The latter plan saves network traffic if events B occur with a much lower frequency than events A.

In our approach, we use process models to derive optimised hybrid execution plans. Throughout this paper, $A \rightarrow B$ denotes the processing order of events in an execution plan. We assume push based communication unless denoted by the keyword *pull*. Thus, '$A \rightarrow B$' and '$B \rightarrow$ *pull* A' are both execution plans for the query $SEQ(A, B)$. The first plan passively waits for events A and subsequently for corresponding events B. The second plan waits for events B and subsequently actively pulls corresponding events A that happened before B.

3 Process Knowledge

This section introduces the process knowledge that we use for query optimisation. It combines event querying with behavioural profiles, a technique for deriving behavioural constraints from a process model (see Fig. 1). We start our discussion from the perspective of the process model, which we will use to derive a behavioural profile and its relations. Information about relations between events can directly be derived from process models [16]. Most of our optimisation rules solely build upon these relations. However, some rules need information about the absolute or relative frequencies of events.

Process Models Process models are extensively used in companies for describing business operations and technical workflows. In many cases, they are directly used as a template for execution by a process engine. Then, the process model plays a normative role and is explicitly enforced by the engine. If, for instance, the process model depicted in Fig. 2 is used by a process engine, it is only possible to execute A and B, potentially repeatedly, followed either by C and E, or D and E, or none of the two before F is executed towards completion of the process.

Behavioural Profiles In our event query optimisation, we will exploit the fact that behavioural constraints can be defined in a normative process model. In particular, we use the notion of a behavioural profile [16]. Such a profile describes

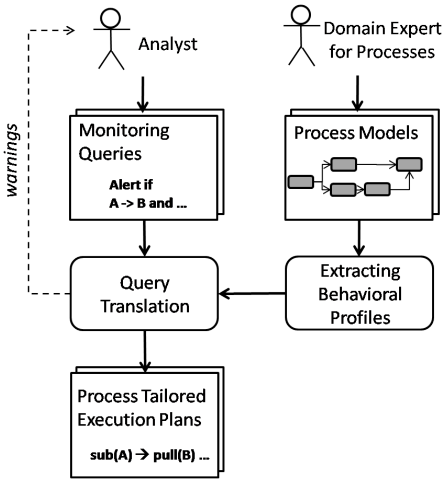

Fig. 1. Automatic tailoring of queries to processes

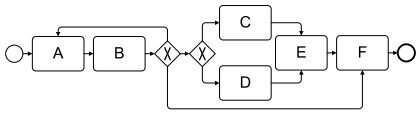

Fig. 2. Example of a process model in BPMN notation

behavioural relations on the level of activity pairs. A behavioural profile consists of three relations that partition the Cartesian product of all activities, such that two activities are either in strict order, exclusive to each other, or in interleaving order. Behavioural profiles can be efficiently calculated for process models as the one shown in Fig. 2. Semantics of the relations of the behavioural profile are defined on the possible execution sequences, alias traces, of the process model.

- The *strict order* relation, denoted by $\rightsquigarrow$, holds between two activities x and y, if x might happen before y, but not vice versa. In other words, x will be before y in all traces that contain both activities.
- The *exclusiveness* relation, denoted by $+$, holds for two activities, if they never occur together in any process trace.
- The *interleaving order* relation, denoted by $||$, holds for two activities x and y, if x might happen before y and y might also happen before x. Thus, interleaving order might be interpreted as the absence of any specific order between two activities.

Further on, the causal behavioural profile defines an additional *co-occurrence* relation, denoted by $\gg$, between activities. Two activities are co-occurring, if any trace (from the initial to the final state of the process) that contains the first activity contains also the second activity. For the process model in Fig. 2, for instance, the behavioural profile states that B is always preceding E, both activities are in strict order, and that C is exclusive to D. The co-occurrence relation states that every trace containing C also contains B, but not vice versa.

Event Frequencies As stated above, some optimisation rules require knowledge about event frequencies. Such information could be provided by domain experts or derived through analysis of event logs. In the remainder of this paper, knowledge of the frequency of an event A will be referred to by λ_A. Still, information on the relative order between frequencies of different events is sufficient for our purposes.

4 Process Tailored Query Optimisation

Process tailored query optimisation follows the idea of using process knowledge to enhance the execution of complex event queries regarding an optimisation goal. This approach complements existing query optimisation techniques. A feature of our solution is that it allows abstracting from details of process instances during query formulation but considers these details in query execution (see Fig. 1).

We provide a set of rules that consider different information about processes and serve different optimisation goals. (1) One goal is reducing the number of required event messages for query processing. This is relevant in scenarios where the network is a bottleneck or communication uses scarce resources. For instance, battery powered sensor devices aim to reduce energy intensive communication in order to maximize battery lifetime. (2) Another optimisation goal is reducing required memory for intermediate query results. This is relevant if resource constrained devices run the query and/or if intermediate results grow large. The latter is the case if queries cover a large time span or events occur at high frequencies. (3) Furthermore, reducing delay between event occurrence and event detection can be a goal for optimisation. This is relevant whenever systems have real-time constraints, i.e., an immediate response to the event is required. These constraints can be found in automated production processes where events trigger production tasks.

We consider optimisation at three stages. First, we provide transformation rules that target complex event queries on the language level (see Section 4.1). Such rules change the semantics of the original query without changing the result set. Second, we provide rules that address the generation of query execution plans (see Section 4.2). A complier can generate several candidate execution plans for the same query. Our rules help selecting the most efficient plan. Third, we provide rules that transform execution plans (see Section 4.3). Transformed plans use events that are outside the scope of the original query but provide information that allow for more efficient query execution.

We use the notation '$A \Rightarrow B, \text{if } X$' for describing rules. Here, This denote that A translates to B if condition X applies. A and B can be parts of query expressions in a high level language or parts of query execution plans. X is a logical expression that involves processes knowledge. For each rule we discuss the effect on optimisation goals and provide some intuitive discussion how the optimisation is achieved. In addition, we also depict small process model fragments that illustrate the applicability of certain rules.

4.1 Query Transformation

Rules for query transformation operate on a high level language level. They change the semantics of the query but - given the considered processes - they do not change the result set of detected events. Below, we list such optimisation rules.

Rule 1: $and(A, B) \Rightarrow seq(A, B)$
if $A \rightsquigarrow B$

Effect of the rule: The rule reduces memory consumption and event messages.
Intuition behind the rule: Any B that matches in a query has a preceding A, while B cannot be observed before A. The rule avoids receiving and storing events B that will have no matching A.
Required knowledge: $\rightsquigarrow$

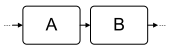

Fig. 3. Process where rule 1 applies

Rule 2: $and(...and(E_1, E_2), ...), E_i) \Rightarrow false$
if $\exists\ E_x, E_y \in \{E_1, ..., E_i\} : (E_x, E_y) \in +$

Effect of the rule: The rule reduces memory consumption and event messages.
Intuition behind the rule: The rule avoids querying for event combinations that cannot occur because involved events are mutually exclusive.
Required knowledge: $+$

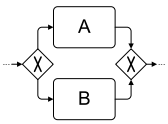

Fig. 4. Process where rule 2 and 3 apply

Rule 3: $seq(...seq(E_1, E_2), ...), E_i) \Rightarrow false$
if $\exists\ E_x, E_y \in \{E_1, ..., E_i\} : (E_x, E_y) \in +$

Effect of the rule: The rule reduces memory consumption and event messages.
Intuition behind the rule: Similar to rule 2, this rule avoids querying for event combinations that cannot occur because involved events are mutually exclusive.
Required knowledge: $+$

Rule 4: $seq(...seq(E_1, E_2), ...), E_i) \Rightarrow false$
if $\exists\ E_x, E_y \in \{E_1, ..., E_i\} : y > x \wedge E_y \rightsquigarrow E_x$

Effect of the rule: The rule reduces memory consumption and event messages.
Intuition behind the rule: The rule avoids querying for combinations that cannot occur because events cannot happen in the queried order.
Required knowledge: $\rightsquigarrow$

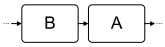

Fig. 5. Process where rule 4 applies

Rule 5: $and(...and(E_1, E_2), ...), E_i) \Rightarrow false$
if $\exists\ false \in \{E_1, ..., E_i\}$

Effect of the rule: The rule reduces memory consumption and event messages.
Intuition behind the rule: The rule propagates rules 2, 3, and 4 through query hierarchies in complex events. If the query includes a complex event E_i that was falsified by any of these rules the queried combination cannot occur.
Required knowledge: + *and/or* $\rightsquigarrow$

Rule 6: $seq(...seq(E_1, E_2), ...), E_i)) \Rightarrow false$
if $\exists\ false \in \{E_1, ..., E_i\}$

Effect of the rule: The rule reduces memory consumption and event messages.
Intuition behind the rule: Similar to rule 5, this rule propagates rule 2, 3 and 4 thought queries hierarchies. If the query includes a complex event E_i that was falsified by any of these rules the queried combination cannot occur.
Required knowledge: + *and/or* $\rightsquigarrow$

4.2 Plan Selection

Rules for plan selection apply in the process of query plan generation. The rules help picking the most efficient execution plan from a set of candidate plans. It is important to note that these rules allow optimal plan selection without knowledge about event frequencies and solely use information of standard process models. Below, we list rules that illustrate optimisation based on plan selection:

Rule 7: $seq(A, B) \Rightarrow B \rightarrow\ pull\ A$
if $A \not\gg B \wedge B \gg A$

Effect of the rule: The rule reduces messages and memory consumption in the event processor.
Intuition behind the rule: We derive from the behavioural profile that each B matches an A but not vice versa. Thus, A happens more often than B and pulling As (instead of pushing) avoids processing irrelevant As.
Required knowledge: $\gg$

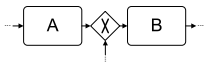

Fig. 6. Process where rule 7 applies

Rule 8: $seq(A, B) \Rightarrow A \rightarrow B$
if $A \gg B \wedge B \not\gg A$

Effect of the rule: The rule reduces messages but (compared to the alternative plan '$B \rightarrow$ *pull* A') increases memory consumption in the event processor.
Intuition behind the rule: We derive from the behavioural profile that each A matches a B but not vice versa. Thus, B occurs more often than A. Pushing A allows to efficiently filter out Bs (those with no preceding A). However, the execution plan requires to keep events A in a buffer until corresponding Bs arrive. It is therefore more memory consuming than the alternative plan '$B \rightarrow$ *pull* A'.
Required knowledge: $\gg$

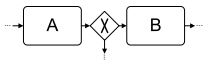

Fig. 7. Process where rule 8 applies

4.3 Plan Transformation

Rules for plan transformation apply after generation of an initial execution plan. The rules add additional events to the execution plan to facilitate more efficient execution. Below, we list rules that illustrate this kind of optimisation:

Rule 9: $A \rightarrow B \Rightarrow A \rightarrow \neg C \rightarrow B$
if $A \rightsquigarrow C \wedge (B, C) \in +$

Effect of the rule: The rule reduces memory consumption in the rule engine but increases the number of event messages.
Intuition behind the rule: The occurrence of C indicates that $A \rightarrow B$ will never match. Thus, A can be dropped from the memory on the occurrence of a corresponding C. The rule is applicable if saving memory is more crucial than reducing event messages.
Required knowledge: $+$ and $\rightsquigarrow$

Rule 10: $A \rightarrow B \Rightarrow C \rightarrow A \rightarrow B$
if $A \not\gg C \wedge C \gg B \wedge C \gg A \wedge A \rightsquigarrow B \wedge \lambda_C << \lambda_A \wedge \lambda_C << \lambda_B$

Effect of the rule: The rule minimizes memory consumption in the rule engine and reduces event messages.

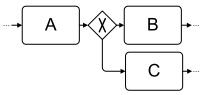

Fig. 8. Process where rule 9 applies

Intuition behind the rule: The rule is beneficial in processes where activities A and B often occur independently but are rare in combination. If an event C indicates that the combination A and B will occur, we can use this C to trigger the processing of queries for combinations of A and B.
Required knowledge: $\gg, \rightsquigarrow$ *and event frequencies* λ

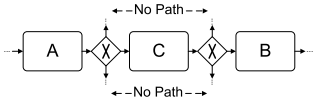

Fig. 9. Process where rule 10 applies

Rule 11: $A \rightarrow B \Rightarrow C \rightarrow A \rightarrow B$
if $A \gg C \wedge C \gg B \wedge A \rightsquigarrow C \wedge C \rightsquigarrow B$

Effect of the rule: The rule reduces buffer sizes and results in shorter delay than B $\rightarrow$ A. However, it increases the number of event messages.
Intuition behind the rule: The data for A can already be pulled if C indicates that B is going to happen. This helps to have A available if the matching B occurs and reduces the delay for detecting the combination of A and B.
Required knowledge: $\gg, \rightsquigarrow$*, and knowledge about timing between events is useful*

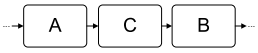

Fig. 10. Process where rule 11 applies

5 Related Work

Technologies for complex event processing and event stream processing are receiving continuously growing attention in the research community. Several research projects addressed different aspects of event processing technologies [1, 3, 5, 9]. A significant proportion of research on query optimisation addresses the application domain of wireless sensor networks (e.g. [9, 18]). Solutions for this domain mainly aim at reducing network load by pushing query operators close to the event sources. Other work presents general purpose approaches to

query optimisation. Srivastava et al. optimise query plans under consideration of differences in the capabilities of available devices [14]. Moreover, network delays can be considered in finding optimal operator placements and corresponding query plans [13]. Query plans can also be rewritten to reuse query operators and minimize resource consumption [12]. Other work optimises the evaluation of query constraints in order to reduce intermediate result sets in query processing [17]. Akdere et al. present an approach that combines push- and pull-based combination to optimise query plans [2].

No approach to our knowledge uses process models to optimise event processing. By extracting behavioural profiles from process models our approach enables optimisations that go beyond general purpose approaches. We foresee potential that solutions presented in existing work can be applied in combination with our work. Still, attention has to be paid to possible interferences between optimization strategies.

6 Conclusion

In this paper, we addressed the challenge of realising complex event processing in an efficient manner. Under the assumption of a normative process model, we showed how the behavioural profile of this process model can be exploited to optimise complex event queries. That is, information of the behavioural profile is used to rewrite queries, select execution plans, or rewrite execution plans. This also enables analysts to abstract from details of process instances during query formulation but still to exploit specifics of process instances in query execution.

While our approach highlights the potential of process model-based query optimisation, we also have to reflect on some limitation. Our approach works solely for expected events representing the accurate behaviour of the process or foreseen exceptional cases, and assumes accurate process models. Therefore, our approach should be applied in a setting where a technical workflow model is directly used for process enactment (e.g. in some manufacturing applications).

In future work, we want to investigate the usage of further information contained in process models for query optimisation. In particular, casual data dependencies between process model activities might be exploited similar to the control flow dependencies that are used in this paper. Moreover, we aim at applying our approach in an industrial case study.

References

1. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Intl. Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.
2. Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, 2008.

3. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical report, Stanford University, 2004.
4. L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.
5. S. Chandrasekaran, O. Cooper, A.Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Intl. Conf. on Innovative Data Systems Research (CIDR)*, 2003.
6. EPCglobal. EPC Information Services (EPCIS) Version 1.01 Specification, September 2007.
7. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, 2001.
8. David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical Report CSL-TR-98-754, 1998.
9. S. R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM TODS*, 30(1):122–173, 2005.
10. G. Muehl, L. Fiege, and P. R. Pietzuch. *Distributed Event-based Systems*. Springer Verlag, Berlin/Heidelberg/New York, 2006.
11. P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1):44–55, 2004.
12. Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.
13. Jeffrey Shneidman, Peter Pietzuch, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. A cost-space approach to distributed query optimization in stream based overlays. In *ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops*, page 1182, Washington, DC, USA, 2005. IEEE CS.
14. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. of the ACM symposium on Principles of Database Systems (PODS)*, New York, NY, USA, 2005. ACM.
15. Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *EDBT*, pages 588–607, 2006.
16. M. Weidlich, A. Polyvyanyy, J. Mendling, and M. Weske. Efficient computation of causal behavioural profiles using structural decomposition. In *Proceedings of Petri Nets 2010*, LNCS. Springer-Verlag, 2010.
17. Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the International Conference on Management of Data*, pages 407–418, New York, NY, USA, 2006. ACM.
18. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, 2002.