International Journal of Hybrid Information Technology
Vol. 7, No. 6 (2014), pp. 71-82
http://dx.doi.org/10.14257/ijhit.2014.7.6.06

LogCEP - Complex Event Processing based on Pushdown Automaton

Jian Cao[1,2], Xing Wei[3], Yaqi Liu[1], Dianhui Mao[1,2] and Qiang Cai[1,2]

[1] *School of Computer and Information Engineering, Beijing Technology and Business University, Beijing, 100048, China*
[2] *Collaborative Innovation Centre for State-owned Assets Administration, Beijing Technology and Business University, Beijing, 100048, China*
[3] *School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China*
caojian@th.btbu.edu.cn, caojian9527@sina.com

Abstract

Complex (or Composite) event processing systems have become more popular in a number of areas. Non-deterministic finite automata (NFA) are frequently used to evaluate CEP queries. However, it is complex or difficult to use the traditional NFA-based method to process patterns with conjunction and negation. In this paper, we proposed a new CEP system LogCEP using pushdown automaton to support efficient processing of conjunction and negation. First, the semantic and query language specification of LogCEP system are presented. Then, an automaton named LogPDA is proposed for query processing in LogCEP system. LogPDA construction method describes how to convert a query to LogPDA automation. The LogPDA execution approach describes how to detect the specified pattern using LogPDA. Meanwhile, most of previous NFA-based optimizations can be employed to improve the evaluation efficiency. Finally, our simulation based experimental results show that our method not only extended the expressibility and processing capability but also didn't lead to efficiency decreasing.

Keywords*: Complex event processing, Pushdown automaton, Query processing, NFA*

1. Introduction

Complex (or Composite) event processing (CEP) systems search sequences of incoming events for occurrences of user specified event patterns [1][2]. They have become more popular in a number of areas, such as detecting trends in stock prices, intrusion detection in network monitoring [3], RFID-based supply-chain monitoring and tracking [4], detecting item stealing in super market.

Most of the recent proposed CEP systems use sequential queries [5][6], which relate events to each other based on temporal orders. However, purely sequential queries are not enough to express many real world patterns, which also involve conjunction (concurrent), disjunction

ISSN: 1738-9968 IJHIT

(choices among many options) and negation, all of which make the matching problem more complex.

Currently, non-deterministic finite automata (NFA), which express patterns as a series of state transitions, are the most commonly used method for evaluating CEP queries. However, previously proposed approaches have two limitations:

***Negation*:** They don't efficiently model negation (events that do not occur) in an NFA when there exist predicates between the negated and non-negated events. Existing NFA-based systems perform negation as a post-NFA filtering step.

***Concurrent events*:** It is hard to support concurrent events, such as conjunction queries (*e.g.*, A and B), in an NFA-based model because of NFA's explicitly order state transitions.

In this paper, a new CEP system called LogCEP is designed and implemented. Our contributions include:

Language. We introduce conjunction and disjunction operators into CEP system and propose a language called Log++, which support these logical operators;

Evaluation model. A Pushdown automaton-based (PDA-based) evaluation model is proposed; new data structures and algorithms are designed to evaluate the PDA-based query processing.

Performance. Our simulator-based evaluations results show that we not only extend the language expressiability but also keep the evaluation efficiency.

The rest of the paper is organized as follows: Section 2 introduces the related work. Our Log++ language is presented in Section 3. Section 4 presents our proposed LogPDA automaton, which is used to evaluate logical operators such as conjunction and disjunction. Section 5 shows evaluation results. Finally, conclusion and future work are presented in Section 6.

2. Related Work

CEP systems first used in active databases to provide active functionalities which are not supported by traditional databases. Petri Nets-based systems such as SAMOS are able to support concurrency, but it is very complex to express and evaluate. Sequence databases [7]-[8] offer SQL extensions for sequence data processing. SEQUIN [8] uses joins to specify sequence operations and thus cannot express Kleene closure. SQL-TS [7] adds new constructs SAL to handle Kleene closure, but restricts to only contiguous tuples in each relevant partition.

NFA are the most commonly used method in CEP. Examples of such work includes SASE [9][10] and Cayuga [11]. Both of them are NFA-based, hence suffers from the limitations of the NFA-based model.

Plan-based method [12] evaluates its event language using event trees, but it arbitrarily constructs a physical tree plan for evaluation rather than searching for an optimal plan. ZStream [13] is a cost-based query processor for adaptively detecting composite events.

However, this cost-based method needs statistics about operator selectivity and data rate. It is difficult to get for some applications.

Others [7][14] try to make use of regular expression and string-based matching algorithm to evaluate CEP. These methods only work efficiently for strictly consecutive patterns, limiting the expressive capability of the pattern matching language.

3. Semantic and Language Specifications

In this section, we first describe the event model for the language and query processing; and then briefly describe the CEP language constructs used in our system and their semantics.

3.1. Event Model

In this model, the input to an event processing system is an infinite sequence of events, which is referred to as an event stream. Our model includes event types and events. Event type, which is similar to types in database system, describes a set of attributes that a class of events must contain. Each event is assigned a timestamp from a discrete ordered time domain. Primitive events are predefined single occurrences of interest that cannot be split into any smaller events. Composite events are events detected by the CEP system from a collection of primitive and/or other events.

3.2. Language Specification

The overall structure of the B++ language is:
SELECT return attributes AS type name
PATTERN complex event expression
WHERE value constraints
WITHIN time constraints

The **SELECT** clause defines the expected output event from the pattern query. This expected output event is a composite event, whose type name is type name and its attributes is the return attributes. In the **PATTERN** clause, the complex event expression describes an event pattern to be matched using different event operators. The **WHERE** clause defines the value constraints on event attributes. The **WITHIN** clause describes the time window during which the matching must occur.

3.3. Operators and Semantics

Sequence (A_1, A_2, ..., A_n; [w]). It specifies a particular order in which the events of interest should occur within a specified time window w. The time window constraint for the sequence operator is optional. (A, B; w) means event B should occur after event A within time window w. The operator is formally defined as:

$$(A_1, A_2, \cdots, A_n\ w) \equiv \exists t_1 < t_2 < \cdots < t_n = t, A_1(t_1) \wedge A_2(t_2) \wedge \cdots \wedge A_n(t_n), t_n - t_1 < w \quad (1)$$

Negation (!). Negation is used to express the non-occurrence of event. This operator is usually used together with other operators. Let's take the combination of sequence operator and negation operation as example, (*A*, !*B*, *C*) means that event *C* follows event *A* without any interleaving occurrence of *B*. It can be formally defined as:

$$(A, !B, C, t) \equiv \exists t_A < t_C = t, A(t_A) \wedge B(t_C) \wedge (\forall t_i \in (t_A, t_C) \neg B(t_i)) \quad (2)$$

Conjunction (A_1&A_2&…&A_n; [*w*]). Conjunction (*A*&*B*) means that both event *A* and *B* occur within a specified time window, and their order doesn't matter. The operator is formally defined as:

$$(A_1\ \&\ A_2\ \&\ \cdots\ \&\ A_n\ w) \equiv \exists t_1, t_2, \cdots, t_n, A_1(t_1) \wedge A_2(t_2) \wedge \cdots \wedge A_n(t_n), t_x - t_y < w,\ x, y \in [1, n] \quad (3)$$

Disjunction (A_1 |A_2 | … | A_n; [*w*]). Disjunction (A | B) means that either event *A* or event *B* occurs within a specified time window. The operator is formally defined as:

$$(A_1 \mid A_2 \mid \cdots \mid A_n; [w]) \equiv \exists t_1, t_2, \cdots, t_n, A_1(t_1) \vee A_2(t_2) \vee \cdots \vee A_n(t_n) \quad (4)$$

Kleene Closure (A^* / A^+ / A^{num}). A^* means that event *A* can occur zero or more times. A^+ means event *A* occurs one or more times. A^{num} means event *A* occurs an exact number of times.

4. Operation Implementation

In this section, we first present our LogPDA, and then describe the construction of LogPDA given a pattern query. Some evaluation techniques are presented at last.

4.1. Our Pushdown Automaton - LogPDA

Pushdown automaton (PDA) is a finite automaton that can make use of a stack to containing data. It differs from finite state machines in two ways: (*i*) it can use the top of the stack of decide which transition to take; (*ii*) it can manipulate the stack as part of performing a transition.

A PDA is formally defined as a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ where:

Q is a finite set of states

Σ is a finite set which is called the input alphabet

Γ is a finite set which is called the stack alphabet

δ is a finite set which is called the transition function, mapping $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ into finite subset of $Q \times \Gamma^*$

$q_0 \in Q$ is the start state

Z is the initial stack symbol

$F \subseteq Q$ is the set of accepting states

In our system, the input alphabet $\sum$ is the set of input event, such as the price of a stock in finance service. The logic pushdown automaton (LogPDA) extends the general PDA as follows:

The stack alphabet Gamma is a set of Boolean expression, which includes Boolean constant (True and False), Boolean variable and Boolean expression connected by Boolean operators such as and (&), or(|), not (!). For example, the Boolean expression *A*&*B*, where *A* and *B* are Boolean variables which can be assigned with Boolean values.

We put a Boolean expression to the stack, a transition can be followed if and only if the current state satisfies the formularies on the transition but also the evaluation result of the Boolean expression on the top of the stack should be true.

In our system, as the content of the top cell of the stack is a Boolean expression, we allow set the values for a specific variable of the Boolean expression.

We employ an additional buffer to compute and store complete matches and to support a wider range of queries such as aggregation. When returns the query result, this buffer is used to construct or generate the result.

4.2. Automaton Construction

We first develop a basic compilation algorithm that constructs a LogPDA automaton that is faithful to the original query.

Step 1. Main structure: Similar to a NFA, LogPDA also has a main structure, which defines the states and transition typology among states. As shown in Figure 1, the **PATTERN** clause of a query uniquely determines the structure of its main structure and the edges of each state.

We use the query-defined event sequence to define the main structure of the pushdown automaton. Each logical operator is considered as a whole, which is called a logical block. A useful approach has been to adopt Non-deterministic Finite Automata (NFA) to represent the structure of an event sequence [15]. For example, Figure 1 shows an main structure for query (*A*, (*B* & *C* & *D*), (*E* & *F*), *G*), where state 0 is the start state, state 1 is for the recognition of an A event, state 2 is for the recognition of a conjunction event (*B* & *C* & *D*) after that, and likewise state 3 is for the recognition of another conjunction event (*E* & *F*). State 4, denoted using two concentric circles, is the accepting state of the automaton after the recognition of *F* event.

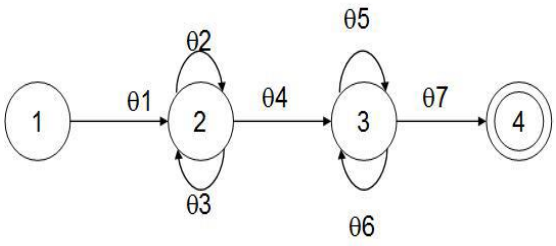


Figure 1. Main Structure of PDA for Query (*A*, (*B* & *C* & *D*), (*E* & *F*), *G*)

For sequence pattern without logical operators, the processing method is the same as in SASE [5][9].

Step 2. Logical operators:

Disjunction. For example, Figure 2 shows the automaton for query (*A*, (*B* | *C* | *D*), *E*), where state 0 is the start state, state 1 is for the recognition of an *A* event, state 2 is for the recognition of either event *B* or *C* or *D* after that, and likewise state 3 is for the recognition of a *E* event. State 2, denoted using two concentric circles, is the accepting state of the automaton. This NFA structure is simple, so the formulas are shown on the transition line directly.

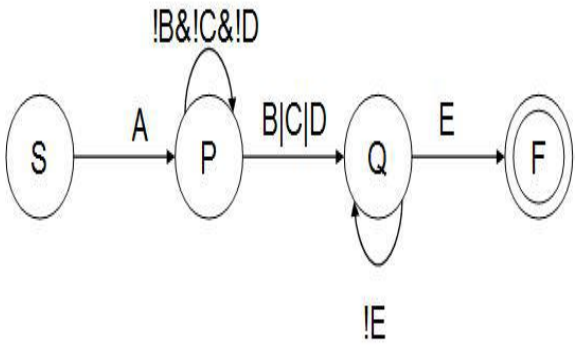


Figure 2. PDA Structure for Query (*A*, (*B* | *C* | *D*), *E*)

Negation. Queries with negation operators can be trivially translated into a NFA.

Conjunction. For each conjunction block, the corresponding node has an ignore edge, a take edge and a proceed edge. If the automaton run arrives state corresponding to a logical block, the Boolean expression of this block is pushed into the stack. As shown in Figure 3(a), the Boolean expression is built up and the value of the Boolean variables is initialized as false, which is also the default value of all the variables.

For the **take** edge, the take edge will be active when the input event is related to any of the event in the Boolean expression. For each taken event, the corresponding Boolean variable is set as true and also the element is stored in the buffer. As shown in Figure 3(b) and (c), if event *C* arrives, the Boolean variable for event *C* is set as TRUE, and then the arriving of event *D* results in the variable for event *D* is set as TRUE.

For the **proceed** edge, it will be active only when the current state satisfies the formulas on the edge and also the top element of the stack or the evaluation result of the top element is TRUE. If the proceed edge is taken, the top element of the stack is popped out, meanwhile, the Boolean expression of the next arrival state is pushed to stack. As shown in Figure 3(d), the arriving of event *B* results in the evaluation result of the Boolean expression to be TRUE, so the proceed transition is followed, which leads the popping out of the top stack element and a new stack element (Boolean expression corresponding to logical block *E* & *F*) is pushed to the stack as shown in Figure 3(e).

The details of **ignore** edge processing will be described in Step 4.

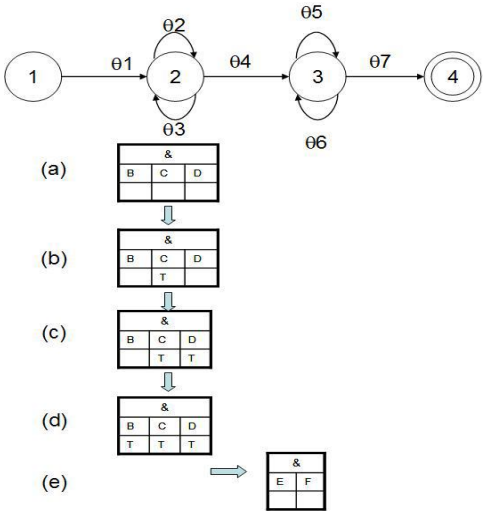


Figure 3. LogPDA Example for Query (*A*, (*B* & *C* & *D*), (*E* & *F*), *G*)

Step 3. Predicates: The algorithm start with the **WHERE** clause and uses the predicates to set formulas of begin, take, and proceed edges as shown in **SASE** project [5].

Step 4. Event selection strategy: The formulas on the ignore edges depend on the event selection strategy in use. Our algorithm uses determines the formula of an ignore edge at a state $q, \theta_{q,ignore}$ as in **SASE** project [5].

Step 5. Time window: Based on our Log++ language, not only the entire pattern has a time window, but also each logical block may have a time window.

For the *pattern level* time constraint, a time window test formula for the entire pattern is added on the begin, proceed edge to the final state. For optimization of the evaluation, the time window test is pushed early, it is copied to all the take, ignore, and begin edges for each state. This allows some unnecessary runs to be pruned as soon as they fail to satisfy the window constraint.

For the *logical block level* time constraint, a time window test formula for this block is added on the proceed edge to the next state. Actually, for the optimization, the time window test can also be copied to the take and ignore edges of this block corresponding state.

4.3. Automaton Execution

Order of formula evaluation. The introduction of logic operators, such as concurrency and not, leads to the out-of-order of formula evaluation. As described in literature [13], it is not straightforward to efficiently model negation (event that do not occur) in an NFA when there exist predicates between the negated and non-negated events. For example, if there is a predicate involving *B* and *C*, such as the pattern "*A* followed by *C* such that there is no interleaving *B* with *B*.price > *C*.price". It is difficult to decide a *B* event transition before the arrival of event *C*. The similar problem exists in queries with conjunction operators.

In our evaluation approach, the predicate will not be evaluated until all the related events arrive. A pending formula list is used to store the formulas which are delayed for evaluation because they are related to non-arrival events. For each new arrival event, we not only check the predicate for the possible transitions and also check the pending list. Either predicate check failure will lead to the termination of the current automaton instance.

Optimizations. We introduce all of the **SASE** [5] optimizations to our system. They include shared version of matched buffer, the merging runs and backtrack algorithm.

5. Performance Evaluation

5.1. Experiment Setup

We have implemented all the query evaluation techniques described in the previous section in a Java-based prototype system. Throughput, which is the number of events processed per second, is used as the performance metric in all our experiments.

To test our system, we implemented an event generator that dynamically creates time series data. In our experiment, we consider 20 events types and 6 attributed for each event type including the timestamp. For each attribute, the domain size was chosen from 10 to 10,000. We also created a query generator based on using the parameters shown in Table 1. The following template is used to generate the query:

PATTERN (E_1, E_2, …, !E_k, (E_m & … & E_{m+x}), …, (E_p | … | E_{p+y}), …, E_L)

WHERE [$attr_1$] **AND** ... **AND** [parameterized predicate]

WITHIN window

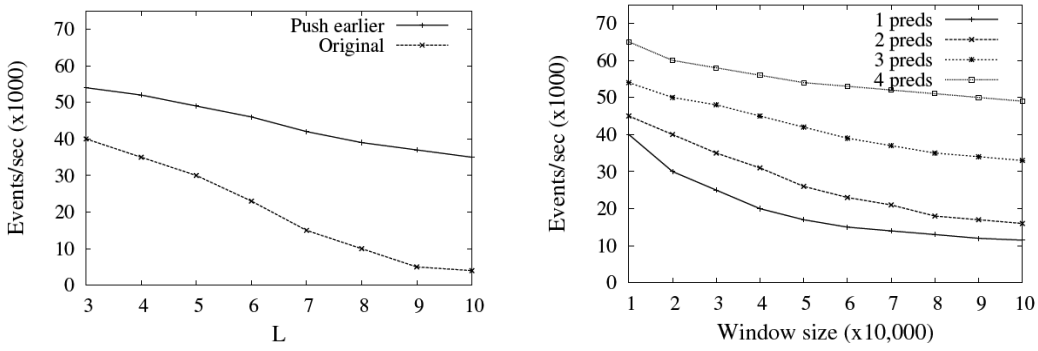
Table 1. Parameters for Query Generation

Parameter	Description	Values
L	Num. of events in each query	20
theta	Zipf distribution of occurrences of event types	0
EP	Num. of equivalence tests per query	1-2
IP	Num. of other parameterized predicates per query	0- 10
SP	Num. of simple predicates per query	0-1
x	Num. of conjunction events per query	1-10
y	Num. of disjunction events per query	1-5
N	Num. of negations in the sequence	0-2
W	Window size	10K-100K

5.2. Basic Evaluation

In this set of experiments, we show the basic properties of our query processing algorithm by varying some parameters of the query and show the effect of time constraints.

Experiment 1: Effect of pushing up time constraint test. In the first experiment, we fixed the number of values allowed for each attribute at [10, 1000], and varied the number of events in each query L from 3 to 10. We fixed the query with one equivalence test, one conjunction test or one disjunction test. The other predicates are all parameterized predicates. The results are shown in Figure 4(a). It shows that pushdown of the time constraint test will increase the throughput significantly. Another observation is that this optimization plays more significant role in long pattern query. This is because longer pattern introduces more active automaton instances; however, this optimization will reduce some instances earlier.



(a) Effect of Push Earlier Window Constraint

(b) Effect of Window Size and Number of Predicates

Figure 4. Experimental Results for Basic Evaluation

Experiment 2: Effect of window size and number of predicates. In this experiment, we fixed the number of values allowed for each attribute at 10,000 and *L* at 5, and varied *W* from 1,000 up to 100,000. We also varied the number of predicates from 1 to 10. We fixed the query with one equivalence test. At most one conjunction test or one disjunction test is used randomly. The other predicates are all parameterized predicates. The results are shown in Figure 4(b). As *W* increases, the throughput decreases, but not linear decreasing. It is because with larger window size, more active instances (runs) are existed in the system, which will introduce higher evaluation cost. Another observation from this experiment is that, with the increasing of the number of predicates, the throughput increases quickly. This is because more predicates means stricter constraints for state transitions. This will also lead to a smaller number of active automaton instances.

5.3. Comparison to SASE

In this section, we compare LogCEP to a NFA-based complex event processing engine, **SASE**, developed at University of Massachusetts. As **SASE** does not support logic operators such as conjunction and disjunction, we first compare to it using simple sequence pattern query, experiment results show that they have almost the same performance.

We then extend the **SASE** engine by enumerate all possible permutation of the events in logical operators, let's call it **SASE** + **ENU**, and then compare it with our LogPDA engine. The results are shown in Figure 5. It shows that LogPDA can improve the performance significantly in condition of conjunction operators in a pattern, especially for longer patterns. The reason is similar to the previous set of experiment. Enumeration of all possible permutation for a conjunction operator leads to a very huge automaton. Because of simultaneous runs, this will increase the amount of active instances.

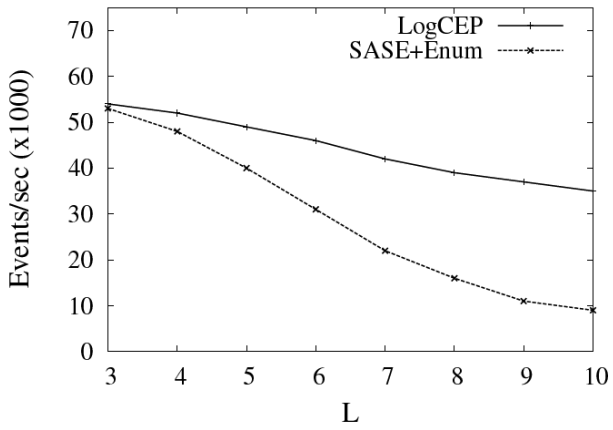


Figure 5. Comparison to SASE

6. Conclusion

In this paper, we proposed and implemented LogCEP, a complex event processing system that can efficiently process logical complex event over event stream. We first proposed a complex event language that allows to express conjunction and disjunction. We then presented a PDA-based query processing approach. Experimental results show that LogCEP can process in stream speed. In the future, we first plan to enhance our system to support distributed complex event processing.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (No.71201004, No.51175033), Funding Project for Innovation on Science, Technology and Graduate Education in Institutions of Higher Learning Under the Jurisdiction of Beijing Municipality (PXM2013_014213_000030_00042300), Collaborative Innovation Centre for State-owned Assets Administration of Beijing Technology and Business University(approval numbers: GZ20131102).

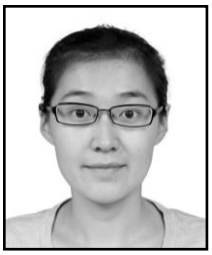
References

[1] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream reasoning and complex event processing in etalis", Semantic Web, vol. 3, no. 4, **(2012)**.
[2] G. Cugola and A. Margara, "Complex event processing with t-rex", Journal of Systems and Software, vol. 85, no. 8, **(2012)**.
[3] T. Takahashi, H. Yamamoto, N. Fukumoto, S. Ano, and K. Yamazaki, "Complex Event Processing to detect congestions in mobile network", IEEE 16th International Conference on Advanced Communication Technology (ICACT), Republic of Korea, Pyeongchang, **(2014)** February 16-19.
[4] V. Govindasamy and P. Thambidurai, "RFID Probabilistic Complex Event Processing in a Real-Time Product Manufacturing System", nature, vol. 2, no. 10, **(2013)**.
[5] J. Agrawal, Y. L. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams. Proceedings of the 2008 ACM SIGMOD international conference on Management of data, USA, New York, **(2008)** June 9–12.
[6] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing", ACM Computing Surveys (CSUR), vol. 44, no. 3, **(2012)**.
[7] R. Sadri, C. Zaniolo, A. Zarkesh and J. Adibi, "Expressing and optimizing sequence queries in database systems", ACM Transactions on Database Systems (TODS), vol. 29, no. 2, **(2004)**.

[8] P. Seshadri, M. Livny and R. Ramakrishnan, "The Design and Implementation of a Sequence Database System", Proceedings of the 22th International Conference on Very Large Data Bases, India, Mumbai (Bombay), **(1996)** September 3-6.
[9] E. Wu, Y. Diao and S. Rizvi, "High-performance complex event processing over streams", Proceedings of the 2006 ACM SIGMOD international conference on Management of data, USA: Chicago, **(2006)** June 27-29.
[10] J. Agrawal, Y. Diao, D. Gyllstrom and N. Immerman, "Efficient pattern matching over event streams", Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Canada: British Columbia, **(2008)** June 9-12.
[11] 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", Proceedings of the 2007 ACM SIGMOD international conference on Management of data, China, Beijing, **(2007)** June 11-14.
[12] Y. S. Chakravarthy, V. Krishnaprasad, E. Anwar and S. K. Kim, "Composite events for active databases: Semantics, contexts and detection", Proceedings of 20th International Conference on Very Large Data Bases, Chile, Santiago, **(1994)** September 12-15.
[13] Y. Mei and S. Madden, "Zstream: a cost-based query processor for adaptively detecting composite events", Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, Rhode Island, Providence, **(2009)** June 29-July 2.
[14] A. Majumder, R. Rastogi and S. Vanama, "Scalable regular expression matching on data streams", Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Canada: British Columbia, **(2008)** June 10-12.
[15] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang and P. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering", ACM Transactions on Database Systems (TODS), vol. 28, no. 4, **(2003)**.

Authors

Jian Cao, male, born in 1982. He received the Ph.D degree from Beijing Institute of Technology, Beijing, China, in 2010. He is currently an associate professor at the College of Computer and Information Engineering, Beijing Technology and Business University. His research interests include image processing and pattern recognition.



Xing Wei, female, born in 1987. She received the MS degree from Beijing Institute of Technology, Beijing, China, in 2010. She is currently a laboratory assistant in University of Science and Technology Beijing. Her research interests include image processing and pattern recognition.

Yaqi Liu, male, born in 1991. He is working toward the master degree in Beijing Technology and Business University. His main interests include image segmentation and content-based image retrieval.

Dianhui Mao, male, born in 1979. He received the Ph.D degree from Huazhong University of Science and Technology, Wuhan, China. He is currently a lecturer at the College of Computer and Information Engineering, Beijing Technology and Business University. His research interests include Location privacy protection and cloud computing.

Qiang Cai, male, born in 1969. He received the Ph.D degree from Beijing University of Aeronautics and Astronautics, Beijing, China. He is currently a professor at the College of Computer and Information Engineering, Beijing Technology and Business University. His research interests include computer graphics, computation geometry, scientific visualization and intelligent information processing.