Complex Event Processing Methods for Real-Time Event Streams (Postprint)

Authors: Qiu Tao, Xie Peiliang, Deng Guopeng, Xi Hongmei,

Zheng Zhi

, Xia Xiufeng

Date: 2022-05-11T10:48:42+00:00

Abstract

Complex event processing is a technique for analyzing event streams in dynamic environments. Typically implemented based on finite state automata, complex event processing techniques generate a large number of overlapping partial matches on the event stream during the matching process, requiring the finite state automata to maintain numerous duplicate matching states. This leads to redundant computation issues in methods based on this technique. To improve matching efficiency in complex event processing, a method utilizing complex event instance coverage technology is proposed. By designing a temporary matching chain-based partitioned storage structure and a matching algorithm built upon this structure, the approach leverages complex event instance coverage to reduce redundant computation, thereby achieving improved matching efficiency. Experimental testing and analysis were performed on both synthetic and real datasets, with comparisons made against two commonly used complex event processing techniques. The experimental results demonstrate that the proposed method can effectively reduce redundant computation during the matching process while ensuring matching correctness, thus enhancing overall matching efficiency.

Full Text

Preamble

Vol. 39 No. 9
Application Research of Computers
ChinaXiv Cooperative Journal

Complex Event Processing Method over Real-Time Event Streams

Qiu Tao[1], Xie Peiliang[1]†, Deng Guopeng[2], Xi Hongmei[2], Zheng Zhi[2], Xia Xiufeng[1]
(1. School of Computer Science, Shenyang Aerospace University, Shenyang 110136, China;
2. Flight Test Station/Flight Test Laboratory, Shenyang Aircraft Industry (Group) Co. LTD, Shenyang 110034, China)

Abstract: Complex event processing is a technology for analyzing event streams in dynamic environments. Complex event processing technology is usually implemented based on finite state automata. During the matching process, a large number of overlapping partial matches are generated on the event stream, and the finite state automaton needs to maintain numerous repeated matching states, leading to redundant computation in methods based on this technology. To improve the matching efficiency of complex event processing, this paper proposes a method using complex event instance coverage technology. By designing a temporary matching chain partition storage structure and matching algorithms based on this structure, redundant computation can be reduced through complex event instance coverage, thereby achieving improved matching efficiency. Experiments were conducted on simulated and real datasets, comparing with two commonly used complex event processing technologies. The experimental results show that the proposed method can effectively reduce redundant computation during the matching process while ensuring matching correctness, and improve overall matching efficiency.

Key words: complex event processing; query optimization; nondeterministic finite automaton; partition storage

0 Introduction

With the further development of the information society, an increasing number of industries are adopting Complex Event Processing (CEP) technology to perform real-time analysis on massive event streams. CEP analyzes relationships among events through techniques such as correlation, aggregation, and filtering, and formulates query rules based on temporal and aggregation relationships between events to continuously extract event sequences that meet requirements from the event stream. This technology has found widespread applications in financial transaction analysis [1, 2], sensor networks [3], Internet of Things [4-6], and transportation [7].

Currently, the processing model based on Nondeterministic Finite Automaton (NFA) is the most popular implementation approach for CEP technology, exemplified by systems such as SASE [8, 9], Cayuga [10, 11], and Siddhi [12]. NFA-based CEP implementations generate temporary matches during the matching process on event streams. These temporary matches can be used by subsequent events to generate new temporary matches and final matching results. Conse-

quently, the matching process produces a large number of overlapping partial matches on the event stream, and the NFA must maintain numerous repeated matching states, resulting in redundant computation. This problem becomes particularly severe when the time window span of complex event queries is large, imposing significant additional overhead on hardware resources such as processors and memory.

To reduce redundant computation in NFA-based CEP and improve matching efficiency, this paper utilizes a chain partition storage structure to manage temporary matches and employs complex event instance coverage to reduce redundant generation and copying of temporary matches, thereby enhancing CEP matching efficiency. In summary, the main contributions of this paper are:

1) We propose the concept of complex event instance coverage, which establishes correlation relationships between temporary matches to reduce redundant generation and copying of temporary matches.

2) We design a temporary matching chain partition storage structure that avoids centralized storage and usage of temporary matches while serving as a carrier for complex event instance coverage to build relationships between temporary matches.

3) We propose the CoverMatch and CombineMatch algorithms for complex event matching based on the temporary matching chain partition storage structure. These algorithms ensure correctness and completeness of CEP matching results while reducing the number and copying of temporary matches.

4) Through comparative experiments and analysis on simulated and real datasets, we validate the effectiveness and performance of the proposed methods and algorithms.

1 Related Work

Complex event processing originates from event-driven business, where each data record generated by a system is regarded as an event. Real-time input data streams constitute real-time event streams, and the CEP engine performs judgment, filtering, and correlation operations on the event stream according to predefined complex event description rules, then outputs a series of higher-level composite events to users. Complex event description rules generally contain event semantics of interest to users or established standards and specifications in specific domains. In other words, CEP can identify user-defined composite events in real-time event streams and provide feedback on the recognition results.

Currently, CEP technology has yielded numerous research achievements, generally employing variants of the nondeterministic finite automaton model for complex event recognition.

Diao et al. proposed SASE, a complex event processing engine, along with an event description language CEL [13, 14] capable of defining composite events. This language features a high-level structure similar to SQL and can define event sequences, matching strategies, event constraints, and time window constraints. The SASE engine transforms composite events defined in the event description language into NFAs, thereby enabling event acquisition and computation on event streams. Diao et al. also extended SASE in SASE+ [15] by introducing support for Kleene closure, negation, and aggregation operations. The main limitation of SASE/SASE+ is that during NFA matching, temporary matches for matching results must be generated. To ensure result accuracy, these temporary matches cannot be discarded within the time window constraint, leading to accumulation of temporary matches and affecting automaton processing efficiency.

Cayuga, developed by Cornell University, also uses NFA as its computational model for event recognition but has relatively weak event description capabilities. Cayuga supports publish-subscribe technology, provides good scalability, and employs query optimization techniques to process multiple events with equivalent states and identical timestamps simultaneously. However, since its core is single-threaded, it does not effectively benefit from these optimization techniques.

FlinkCEP [16] is conceptually similar to SASE and also uses event constraints as conditions for NFA state transitions. From the perspective of event description languages, the main difference between FlinkCEP and SASE is that FlinkCEP does not support a language for defining composite events. Instead of an event description language, FlinkCEP requires users to write event descriptions in Java or Scala, which is less readable and error-prone.

In addition to NFA-based CEP implementations, another approach using trees as the computational model has been extensively studied and applied.

Mei et al. proposed ZStream [17], a typical tree-based CEP implementation. ZStream' s event description language is very similar to SASE, following most of the same syntax. ZStream stores events in leaf nodes, with internal nodes corresponding to operators. During event stream processing, it does not immediately evaluate constraint conditions for arriving events but instead collects events in batches for processing. The combination of tree structure and batch processing allows ZStream to perform various CEP tasks based on expected cost and conditional constraints. For example, for a given complex event sequence `<A, B>`, SASE would create a new temporary match for every occurrence of event A, even if event B has a low probability of occurrence. In contrast, ZStream can follow an alternative matching rule, waiting for event B to arrive before batch-checking previously arrived A events. However, ZStream still cannot avoid the accumulation of a large number of unprocessed events, which is essentially the same as the temporary match accumulation problem in NFA-based approaches.

Based on the above research, whether using NFA-based or tree-based ap-

proaches, ensuring correctness and completeness of complex event matching results requires storing temporary matches within at least one time window range. When the time window span is large, complex event processing imposes significant load on processor computing power and memory resources.

2 Preliminary Work

Complex event processing is a query analysis technology oriented toward event streams, aiming to identify higher-level complex events that satisfy complex event description semantics from event streams composed of numerous basic events. This section details the preliminary knowledge involved in complex event processing.

Definition 1 (Event Stream). An event stream $S(s_1, s_2, \dots, s_n)$ consists of a series of basic event instances, where s_i is an event instance containing information such as event type, event attributes, and timestamp when the event occurred.

Event streams are often composed of data from multiple data sources. In many research and application domains using massive data, such as weather forecasting [18], maritime navigation [19], and transportation data research [20], data sources can be collection devices or sensors. Therefore, data fusion techniques [21, 22] must first be applied to obtain event streams containing multiple event types.

Figure 1 shows an example of an event stream generated by a vessel during navigation. The event stream contains three event types: A, B, and C, where A represents low-speed startup, B represents a 90° left turn of the bow, and C represents low-speed docking. Each event instance includes a timestamp and attribute values, represented here by corresponding lowercase English letters. For example, b_1 is the first event instance of type B with timestamp 2, and also includes attributes such as travel speed, direction, and tilt angle.

By observing the event stream shown in Figure 1, we can see that within the time window 1–12, the vessel first starts up at low speed, then makes a turn during navigation, and finally docks at low speed.

A complex event is a composite event composed of several event instances on an event stream $S(s_1, s_2, \dots, s_n)$, represented as $R(r_1, r_2, \dots, r_n)$. A complex event represents an objectively existing specific event occurring on the event stream, and its semantics are typically expressed through queries defined by complex event description languages.

Definition 2 (Complex Event Query). A complex event query Q consists of a set of constraints defined on basic events to define and represent the attribute characteristics of higher-level complex events.

Current research has proposed various forms of complex event description languages to define complex event queries, among which SASE proposed the most

representative one. It features concise syntax rules and flexible expressive power, so this paper uses the SASE complex event description language to define complex event queries. The SASE event description language is a declarative language.

By default, queries read events arriving in real-time from the event stream, perform complex event processing, and finally feed back successfully matched complex events to users.

To explain the meaning of the SASE event description language structure, we use an example constructed based on a road traffic scenario. In this example, the event type `TrafficInfo` represents traffic data reports collected at road locations, with each report corresponding to one event instance. Assume the report content includes the location position, as well as traffic flow and average vehicle speed at a certain moment. The constructed query is shown as Q_1.

The `PATTERN` section defines the event sequence of a complex event, using the `SEQ` structure to specify event sequences composed of two event types. Both event types are `TrafficInfo`, with Kleene closure applied to the second one, indicated by "+" to represent one or more events of the specified type, which must be declared with "[]" .

In addition to defining sequential and closure sequences, the `PATTERN` section can also define negation operations by adding "!" before the event type. For example, `SEQ(A a, !B b, C c)` indicates that between event instances of type A and type C, no event instance of type B is allowed under the condition `a.timestamp < c.timestamp`.

The `WHERE` section specifies the matching strategy used in the current query. `skip-till-any-match` indicates that all results in the event stream will be matched. This paper only discusses complex event matching under this strategy, as results matched by other strategies are subsets of those matched by the skip-till-any-match strategy.

The `AND` section defines event constraints as an extension of `WHERE` constraints. `WITHIN` defines the time window constraint, limiting the time span of matched results to a certain range.

A complex event query Q written according to these rules will be analyzed by the CEP engine on the event stream to obtain query results.

3 Query Optimization Techniques

This chapter provides a detailed introduction to the complex event query optimization techniques proposed in this paper. We first analyze NFA-based complex event matching technology, then design and implement a temporary matching chain partition storage structure and query optimization algorithms to optimize NFA-based CEP methods and improve query matching efficiency.

3.1 NFA-Based Matching Method

The NFA-based matching approach is currently the most widely used and effective complex event matching technology. Taking SASE as an example, processing a complex event query involves: (1) parsing the query into an NFA; (2) reading the event stream; (3) performing complex event matching to generate temporary matching results or successful matching results.

This section constructs a query Q_2 and its matching process to detail the NFA-based matching method.

Q_2 contains a Kleene closure of event type B, indicating matching one or more event instances of type B. Under the constraints for B events, Q_2 only matches sequences of B events with decreasing `val` attribute values. Therefore, complex events matched by Q_2 start with an event instance of type A, followed by a sequence of B event instances with decreasing `val` values, and end with an event instance of type C.

To clearly illustrate the NFA-based matching process, assume Q_2 performs queries on a specific event stream $S(a_1, b_1, b_2, b_3, c_1)$, with timestamps and attribute values shown in Figure 3.

The matching process of Q_2 on S can be demonstrated as shown in Figure 4.

As shown in Figure 4, SASE processes event streams through NFA. When event a_1 arrives and verification succeeds, a temporary match containing a_1 is initialized: $(a_1, -)$. The existence of this temporary match indicates that the "a" node state in the NFA of Figure 2 has a match. When event instance b_1 arrives and verification succeeds, the system backs up $(a_1, -)$ and updates the original temporary match to $(a_1, b_1, -)$. Thus, two temporary matches exist in the system. When the next arriving event is verified successfully, both temporary matches need to be copied and updated, resulting in four temporary matches.

This method can obviously find all correct matching results in the event stream, but its drawbacks are also apparent: many temporary matches are generated during the matching process to represent temporary matching sequences and to preserve NFA matching states. As matching progresses, the number of temporary matches in the system grows exponentially in the worst case, leading to increasing computational and storage overhead.

3.2 Reducing Redundant Computation

To address the redundant computation caused by a large number of temporary matches and copies in NFA-based CEP, this paper proposes the concepts of "event instance coverage" and "complex event instance coverage." We design a temporary matching chain partition storage structure and the CoverMatch matching algorithm based on this structure to reduce redundant computation and improve NFA-based CEP performance. Additionally, we design the CombineMatch algorithm to complete the generation of matching results.

Definition 3 (Event Instance Coverage). When an event instance s_i and the adjacent previous event instance s_{i-1} in the event stream belong to the same event type, and both can be successfully verified by the current NFA and act on the same state node of the NFA, then s_i is called an event instance coverage of s_{i-1}.

Definition 4 (Complex Event Instance Coverage). For two matching results M_1 and M_2, if every event instance in M_1 is equal to the corresponding event instance in M_2 or is an event instance coverage of the corresponding event instance in M_2, then M_1 is called a complex event instance coverage of M_2.

Event instance coverage is transitive. For example, for three event instances b_1, b_2, b_3 in event stream S of Figure 3, all having the same event type and being successfully verified by the NFA corresponding to Q_2 while acting on the same state node, b_2 is an event instance coverage of b_1, and b_3 is an event instance coverage of b_2, making b_3 also an event instance coverage of b_1. Similarly, complex event instance coverage is also transitive.

For instance, performing matching of `SEQ(A a, B b, C c)` on the event stream in Figure 3 yields matching results $m_1(a_1, b_1, c_4)$, $m_2(a_1, b_2, c_4)$, and $m_3(a_1, b_3, c_4)$. According to Definition 4, m_2 is a complex event instance coverage of m_1, and m_3 is a complex event instance coverage of m_2. By transitivity, m_3 is also a complex event instance coverage of m_1.

Complex event instance coverage can be applied in many scenarios because it operates on complex events closest to the current event. Examples include finding the most recent price rebound event for a specific stock in a stock event stream, or finding the latest congestion event at a location in a traffic data event stream.

During complex event matching, focusing only on complex event instance coverage can complete matching tasks more efficiently when consecutive instances of the same event type appear. Compared with SASE' s skip-till-any-match strategy that uses numerous temporary matches to compute all results, our method can match complex event instance coverage while using the result joining algorithm to obtain all matching results efficiently without generating additional temporary matches, thereby reducing redundant computation.

3.2.1 Temporary Matching Chain Partition Storage Structure To enable NFA to support our proposed optimization method for reducing redundant computation, we enhance NFA by designing a temporary matching chain partition storage structure to replace the original centralized temporary matching storage approach. The drawbacks of centralized temporary matching storage have been mentioned above: each time a new event arrives, all temporary matches must be traversed and verified, which severely consumes computational resources. In contrast, the temporary matching chain partition storage structure leverages the concepts of event instance coverage and complex event instance coverage to avoid the disadvantages of centralized storage.

Assume we have query Q_4: `SEQ(A a, B b, C c)` and event stream $S_t(a_1, a_2, b_1, b_2, a_3, b_3, c_1, c_2)$. For simplicity, the event constraints for Q_4 are empty, and its time window is set to be larger than the time range of event stream S_t. When performing complex event instance coverage matching for Q_4 on S_t, the temporary matching chain partition storage structure is as shown in Figure 5.

After Q_4 is compiled into an NFA, there are three main states (state F is the final state), which generate three partitions: A partition, B partition, and C partition. When event instances a_1 and a_2 from S_t arrive and are processed, they generate temporary matches stored in the A partition and packaged into linked list nodes `<a_1>` and `<a_2>`. Since a_2 is an event instance coverage of a_1, `<a_2>` becomes the successor node of `<a_1>` and is exposed to the next B partition. In each partition, only the tail node of each linked list is exposed to the next partition.

When event instance b_1 arrives, it first finds the exposed node in the previous A partition (node `<a_2>`), copies it, updates it to become node `<a_2, b_1>`, and stores it as the list head in the B partition while exposing it to the C partition. Subsequently, when event instance b_2 arrives, it also finds the exposed node `<a_2>` in the A partition, generating node `<a_2, b_2>`. Since `<a_2, b_2>` is a complex event instance coverage of `<a_2, b_1>`, it becomes the successor node of `<a_2, b_1>` and replaces `<a_2, b_1>` as the exposed node to the C partition. Similarly, after all events in S_t arrive and are processed, we finally obtain three complex event instance coverages: `<a_2, b_2, c_2>`, `<a_2, b_3, c_2>`, and `<a_3, b_3, c_2>`. This matching process is shown in Algorithm 1.

Algorithm 1: CoverMatch Algorithm Based on Chain Partition Storage Structure
Input: Event stream S_e, query Q
Output: Complex event instance coverage set R_s

1. $e \leftarrow$ null
2. $R_s \leftarrow \emptyset$
3. $nfa \leftarrow$ parse(Q)
4. **while** ($e \leftarrow S_e$.nextEvent() and $e \neq$ null) **do**
5. **if** $\neg nfa$.verify(e) **do**
6. `**continue**`
7. $tempMatchList \leftarrow$ buildTempMatch(e)
8. checkTimeWindow($tempMatchList$)
9. **if** $tempMatchList$.isEmpty() **do**
10. **continue**
11. $region \leftarrow nfa$.getRegion(e.eventType)

12. **for each** $tempMatch$ **in** $tempMatchList$ **do**
13. $node \leftarrow$ getDominateMatch($tempMatch, region$.getCandidates())
14. **if** $node \neq$ null **do**
15. `$node.\text{next} = tempMatch$`
16. **else**
17. `$region.\text{setCandidate}(tempMatch)$`
18. **for each** r **in** nfa.getLastRegion().getCandidates **do**
19. R_s.add(r.prev)
20. **return** R_s

In Algorithm 1, the `buildTempMatch` method on line 7 traverses exposed matching nodes in the previous partition using event e, copies nodes that can be updated, performs updates, and then enters the current partition for complex event instance coverage checking. If it is not a complex event instance coverage of any node, it forms a separate chain (line 17); otherwise, it becomes the tail node of a chain (line 15). Finally, it returns the most recent complex event instance coverage. Since the data structure is a doubly-circular linked chain, line 19 uses `prev` to directly locate the tail node and obtain the most recent complex event instance coverage.

We now analyze the time and space complexity of Algorithm 1. When the system is processing event e, assume that in the event stream S within twice the time window, the maximum number of instances of a single event type is K. Then the maximum number of chain structure instances in the partition to which e belongs is K. Let the number of temporary matches generated by event e in its partition be m. Since temporary matches are generated only based on the tail node of each chain structure in the previous partition, the maximum value of m is also K. Therefore, the time complexity for processing event e matching is $O(K^2)$, and the time complexity of Algorithm 1 is $O(|S|K^2)$, where $|S|$ is the number of events in event stream S. Since only tail nodes in the temporary matching chain structure of event e' s partition participate in building new temporary matching nodes, the space complexity for obtaining new temporary matching nodes through event e is $O(K^2)$.

In CEP, temporary match copying requires deep copying. Completing one deep copy of a temporary match requires creating a new temporary match instance in memory and copying all attribute values and references from the original temporary match instance to the new one. Traditional methods perform temporary match copying while traversing all temporary matches, which is detrimental to providing good system throughput and increases computational overhead. Additionally, having all temporary matches explicitly exist in memory as instances increases memory overhead. Taking the matching process of query Q_4 on event stream S_t as an example, traditional NFA-based matching methods need to

generate 24 temporary matches and 21 temporary match copies, whereas the temporary matching chain partition storage structure and the proposed matching algorithm generate only 13 temporary matches and 10 copies. The reduction in the number of temporary matches, copies, and traversal operations improves system throughput and reduces memory overhead.

The temporary matching chain partition storage structure is proposed based on the concept of complex event instance coverage. Leveraging the characteristics of partition storage, processing complex events does not require traversing all temporary matches in the system, achieving reduction in temporary match quantity and copying without affecting the correctness of final matching results.

When users only need the complex event instance closest to the current time, the system can simply store `<a_3, b_3, c_2>`. If users require all results like the skip-till-any-match strategy, all most recent complex event instance coverages are needed, and other node information on their linked lists is used for result joining to obtain all matching results.

3.2.2 Matching Result Joining We further improve the generation of matching results. When users specify the need to obtain all matching results, Algorithm 2 is used to perform reverse matching result joining based on the results from Algorithm 1. Using query Q_4: `SEQ(A a, B b, C c)` and event stream $S_t(a_1, a_2, b_1, b_2, a_3, b_3, c_1, c_2)$ as an example, the query results of Q_4 in the matching process and results shown in Figure 5 are `<a_2, b_2, c_2>`, `<a_2, b_3, c_2>`, and `<a_3, b_3, c_2>`. All matching results are obtained through these results and the chain partition structure.

Since a doubly-circular linked chain structure is used, we can start from the bottommost complex event instance coverage and traverse the linked list in reverse. By collecting the last event instance of each partition' s linked list node and using recursion to combine event instances from each partition, this process is shown in Algorithm 2. For Q_4, simply passing the three nodes `<a_2, b_2, c_2>`, `<a_2, b_3, c_2>`, and `<a_3, b_3, c_2>` into Algorithm 2 yields all matching results of Q_4.

Algorithm 2: CombineMatch Algorithm for Matching Result Joining
Input: Complex event instance coverage R
Output: All matching results M

1. $M \leftarrow \emptyset$
2. $list \leftarrow \emptyset$
3. **while** true **do**
4. $list$.add(R.getLastEvent())
5. $R \leftarrow R$.prev
6. **if** R.isHead() **do**

7. `$R \leftarrow \text{getPrevBlockNode}(R)$`
8. `**break**`
9. **return** $M \leftarrow \text{doCombine}(list, \text{CombineMatch}(R))$

In Algorithm 2, the `getPrevBlockNode` method on line 7 obtains the transfer node of R in the previous partition by passing the linked list vertex R. Line 9 uses recursion to obtain the final joining results, where the `doCombine` method performs the joining operation.

We now analyze the time and space complexity of Algorithm 2. Assume the current number of partitions is n, and the longest complex event sequence length defined in multi-complex-event queries is N, making the maximum value of n equal to N. The number of recursive function calls is at most N. If the average length of temporary matching chains related to complex event instance coverage S in each partition is m, then the space complexity of Algorithm 2 is $O(Nm)$. Since the execution operations in each recursive function body have complexity $O(m)$, the time complexity of Algorithm 2 is $O(Nm)$.

Using node information on temporary matching chains for result joining can obtain all matching results based on Algorithm 1. The joining process does not require building or copying temporary matches and does not increase the scale of temporary matches in the system, thus saving memory overhead.

Using the leftmost linked list in Figure 5 as an example, we first obtain node `<a_2, b_2, c_1>` from the C partition. In the C partition, we get $[c_1, c_2]$ by obtaining the last event instance of each node. Then we obtain $[b_1, b_2]$ from the corresponding linked list in the B partition, and subsequently obtain $[a_1, a_2]$ from the A partition. Through recursive `doCombine` method, we combine the three to get eight matching results: (a_1, b_1, c_1), (a_1, b_1, c_2), (a_1, b_2, c_1), (a_1, b_2, c_2), (a_2, b_1, c_1), (a_2, b_1, c_2), (a_2, b_2, c_1), and (a_2, b_2, c_2). Similarly, using the other two complex event instance coverages from the C partition also yields all their corresponding matches. This approach ultimately obtains all matching results to achieve the skip-till-any-match strategy while avoiding massive temporary match generation and copying.

4 Experiments

This chapter analyzes and validates the effectiveness of the proposed matching optimization method based on the temporary matching chain partition storage structure through experimental comparisons. The optimization method was implemented in Java for the experiments. This chapter analyzes the experimental results from multiple perspectives and demonstrates its effectiveness through performance comparisons.

4.1 Experimental Setup

Experiments were conducted on two datasets. The first is a simulated data stream generated by an event stream generator, and the second is a real dataset.

The first dataset is an ABC-type event stream, where event types are defined using uppercase English letters. Each event carries a timestamp and various attribute values. Before generating the event stream, the event stream generator allows customization of the number of event types, attribute count, and attribute value ranges. Each event in the stream is randomly generated. This dataset contains 100,000 raw events.

The second dataset contains vehicle traffic data collected by sensors from Aarhus, Denmark [23]. The dataset was obtained from four months of data collected at 449 observation points, comprising 13,577,132 raw events. Each event represents traffic conditions at an observation point, with attributes including ID, average vehicle speed at that point, and total vehicle count observed in the past five minutes.

The proposed optimization method is compared with the popular NFA-based SASE and Siddhi methods. The optimization method based on the temporary matching chain partition storage structure is denoted as **LinkedCEP**. Experiments were conducted on a Linux system with an Intel Core i7 2.60 GHz CPU and 16 GB memory.

4.2 Experimental Analysis

First, we compare the matching performance and number of temporary matches generated by LinkedCEP, SASE, and Siddhi on different datasets. Since the proposed CoverMatch algorithm only matches complex event instance coverages in the event stream and does not generate all matching results, to ensure fair performance comparison, LinkedCEP includes the CombineMatch result joining algorithm to support generating all matching results.

Since the raw events and attributes differ between the two datasets, different queries were synthesized for each dataset. LinkedCEP, SASE, and Siddhi were used to execute corresponding queries on both datasets. Two groups of experiments were designed for comparative analysis.

Experiment 1: Five groups of queries were synthesized for each dataset, with each group containing 10 basic complex event queries. These queries have a sequence length of 3. Since time window significantly impacts matching time, the time window was unified to 50 seconds, with results showing the average matching time for each query group.

Figures 6 and 7 show performance comparisons on the ABC event stream and traffic event stream, respectively. With consistent query sequence length and time window, LinkedCEP's matching efficiency outperforms the other two methods for every query group. For example, for the matching results of query group

Q_3 in Figure 6, LinkedCEP takes 220 ms, while SASE and Siddhi take 385 ms and 290 ms, respectively. Similarly, in Figure 7 for traffic event stream processing, LinkedCEP' s matching efficiency is superior to SASE and Siddhi. For the matching results of query group Q_2, LinkedCEP takes 130 ms, while SASE and Siddhi take 240 ms and 160 ms, respectively.

Experiment 2: We test matching performance under different time windows. Using the query groups from Experiment 1, the time window is varied to 50s, 100s, 150s, 200s, and 300s, with results showing the average matching time for each query group.

As shown in Figures 8 and 9, LinkedCEP' s processing performance is superior to the other two methods under different time windows. For example, in Figure 8, when the time window is 200s, LinkedCEP' s processing performance is 34% higher than SASE and 19% higher than Siddhi. Figures 8 and 9 also show that as the time window increases, LinkedCEP' s performance improvement over the other two methods becomes more significant. This is because the time window size affects the number of temporary matches—larger time windows accumulate more temporary matches, impacting performance more severely.

Experiment 3: We test event throughput under different query scales. Throughput refers to the number of events processed per second; higher throughput indicates higher computational efficiency. Five groups of queries were generated for both the ABC dataset and traffic dataset, with query quantities of 50, 100, 150, 200, and 250. Each query has a fixed time window of 50 seconds. SASE, Siddhi, CoverMatch, and LinkedCEP were tested.

As shown in Figures 10 and 11, LinkedCEP has higher throughput than SASE and Siddhi on both datasets. Since CoverMatch only matches complex event instance coverages, and LinkedCEP obtains all matching results through result joining on top of CoverMatch, CoverMatch has higher throughput than LinkedCEP. In Figure 11, due to fewer complex event instance matches in the traffic dataset, the throughput of CoverMatch and LinkedCEP are similar. In the ABC dataset, the throughput of CoverMatch and LinkedCEP exceeds SASE and Siddhi under all five query scales, with more significant improvements than on the traffic dataset. This demonstrates that when the data stream contains adjacent events of the same type, the proposed method achieves greater performance improvements.

These experiments demonstrate that the proposed method can effectively improve the matching efficiency of NFA-based CEP technology, achieving efficiency gains through reduced redundant computation in scenarios with longer query sequences or larger time windows.

5 Conclusion

This paper studies the optimization of complex event matching based on NFA. To address the inefficient matching problem caused by numerous temporary

matches during the matching process, we propose the concept of complex event instance coverage. We design a chain partition storage structure for temporary matches and efficient matching methods on this structure to utilize complex event instance coverage for reducing redundant computation during matching. Experimental results demonstrate that the matching technology using complex event instance coverage can effectively improve CEP performance.

Complex event processing technology has broad application prospects. Future research will focus on query optimization across multiple complex events. We will first design an algorithm to share complex event instance coverage chains among multiple queries using the proposed temporary matching chain partition storage structure, and explore and design a monitoring model suitable for result sharing in multi-query scenarios. Finally, we will analyze the sharing capability of the model and evaluate its processing performance through experiments.

References

[1] Demers A, Gehrke J, Hong M, et al. Towards expressive publish/subscribe systems [C]// International Conference on Extending Database Technology. Springer, Berlin, Heidelberg, 2006: 627-644.

[2] Chan Yuandong, Gou Qiaojin, Liang Zhongyan, et al. A Survey of Rule Engine [J]. Informatization Research, 2021, 47 (2): 6.

[3] Hill M, Campbell M, Chang Y C, et al. Event detection in sensor network traffic monitoring using large-scale mobile signaling data [C]// Proceedings of the second international conference on Distributed event-based systems. 2008: 95-106.

[4] Zhou Q, Simmhan Y, Prasanna V. Incorporating semantic knowledge into dynamic data processing for smart power grids [C]// International Semantic Web Conference. Springer, Berlin, Heidelberg, 2012: 257-273.

[5] Zhao Huiqun, Li Huifeng, Liu Jinluan. Study on RFID complex event pattern clustering algorithm of Internet of things [J]. Application Research of Computers, 2018, 35 (2): 3.

[6] Rahmani A M, Babaei Z, Souri A. Event-driven IoT architecture for data analysis of reliable healthcare application using complex event processing [J]. Cluster Computing, 2021, 24 (2): 1347-1360.

[7] Qiao Yazheng, Cheng Lianglun, Wang Tao, et al. Study on multi-dependency complex event processing in subway train environment [J]. Application Research of Computers, 2019, 036 (008): 2355-2358, 2367.

[8] Agrawal J, Diao Y, Gyllstrom D, et al. Efficient pattern matching over event streams [C]// Proceedings of the 2008 ACM SIGMOD international conference on Management of data. 2008: 147-160.

[9] Zhang H, Diao Y, Immerman N. On complexity and optimization of expensive queries in complex event processing [C]// Proceedings of the 2014 ACM SIGMOD international conference on Management of data. 2014: 217-228.

[10] Demers A J, Gehrke J, Panda B, et al. Cayuga: A General Purpose Event Monitoring System [C]// Cidr. 2007, 7: 412-422.

[11] Brenna L, Demers A, Gehrke J, et al. Cayuga: a high-performance event processing engine [C]// Proceedings of the 2007 ACM SIGMOD international conference on Management of data. 2007: 1100-1102.

[12] Suhothayan S, Gajasinghe K, Loku Narangoda I, et al. Siddhi: A second look at complex event processing architectures [C]// Proceedings of the 2011 ACM workshop on Gateway computing environments. 2011: 43-50.

[13] Noh W F. CEL: A time-dependent, two-space-dimensional, coupled Eulerian-Lagrange code [R]. Lawrence Radiation Lab., Univ. of California, Livermore, 1963.

[14] Wang Yixiong, Liao Husheng, Kong Xiangxuan, et al. CEStream: A Complex Event Stream Processing Language [J]. Computer Science, 2017, 44 (4): 5.

[15] Diao Y, Immerman N, Gyllstrom D. Sase+: An agile language for kleene closure over event streams [J]. UMass Technical Report, 2007.

[16] Online Apache flinkcep [EB/OL]. [2020-12-04]. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html.

[17] Mei Y, Madden S. Zstream: a cost-based query processor for adaptively detecting composite events [C]// Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. 2009: 193-206.

[18] Fairoz Q. Kareem, Adnan Mohsin Abdulazeez, Dathar A. Hasan. Predicting Weather Forecasting State Based on Data Mining Classification Algorithms [J]. Asian Journal of Research in Computer Science, 2021.

[19] Patroumpas K, Alevizos E, Artikis A, et al. Online event recognition from moving vessel trajectories [J]. GeoInformatica, 2017, 21 (2): 389-434.

[20] Huang Qiuyang, Yang Yongjian, Xu Yuanbo, et al. Citywide road network traffic monitoring using large-scale mobile signaling data [J]. Neurocomputing, 2021, 444.

[21] Zhang J. Multi-source remote sensing data fusion: status and trends [J]. International Journal of Image and Data Fusion, 2010, 1 (1): 5-24.

[22] Lu Liping, Zhang Xiaoqian. Data fusion method of multi-sensor target recognition in complex environment [J]. Journal of Xidian University, 2020, 47 (4): 8.

[23] Ali M I, Gao F, Mileo A. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets [C]// International Semantic Web

Conference. Springer, Cham, 2015: 374-389.

Note: Figure translations are in progress. See original paper for figures.