Efficient Complete Event Trend Detection over High-Velocity Streams

Huiyao Mei[†], Hanhua Chen[†], Hai Jin[†], Qiang-Sheng Hua[†], Bing Bing Zhou[†]
[†]National Engineering Research Center for Big Data Technology and System
Cluster and Grid Computing Lab
Services Computing Technology and System Lab
Huazhong University of Science and Technology, Wuhan, 430074, China
[‡]School of Information Technologies
The University of Sydney, NSW 2006, Sydney
Emails:{hym,chen,hjin,qshua}@hust.edu.cn,bing.zhou@sydney.edu.au

ABSTRACT

Complete Event Trend (CET) detection over large-scale event streams is important and challenging in various applications such as financial services, real-time business analysis, and supply chain management. A potential large number of partial intermediate results during complex event matching can raise prohibitively high memory cost for the processing system. The state-of-the-art scheme leverages compact graph encoding, which represents the common sub-sequences of different complex events using a common sub-graph to achieve space efficiency for storing the intermediate results. However, we show that such a design raises unacceptable computation cost for the graph traversal needed whenever a new event comes. To address this problem, in this paper, we propose a novel *attribute-based indexing* (ABI) graph model to represent the relationship between events. By classifying the predicates and constructing the graph based on both the comparators in the predicates and the attribute values of the events, we achieve parallel event stream processing and efficient graph construction. Our design significantly reduces the total computation cost of graph construction from $O(n^2)$ to $O(nlog(m))$, where n is the number of events and m is the number of the attribute vertices. We further design several efficient traversal-based algorithms to extract CETs from the graph. We implement our design and conduct comprehensive experiments to evaluate the performance of this design. The results show that our design wins a couple of orders of magnitude back from state-of-the-art schemes.

KEYWORDS

Big data, stream process, complete event trend

ACM Reference Format:
Huiyao Mei[†], Hanhua Chen[†], Hai Jin[†], Qiang-Sheng Hua[†], Bing Bing Zhou[†]. 2021. Efficient Complete Event Trend Detection over High-Velocity Streams. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3472456.3472526

The Corresponding Author is Hanhua Chen (chen@hust.edu.cn).

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08. . . $15.00
https://doi.org/10.1145/3472456.3472526

1 INTRODUCTION

Complete Event Trend (CET) [13] detection is to process high-velocity event streams and detect all the complete event sequences matching the a user-defined event pattern in a given time range. An event trend is a time-ordered event sequence that matches a user-defined query in a time window [14]. An event trend is complete only if it is not a sub-sequence of any other event trends. This technology is widely used in real applications such as traffic management, stock trading analysis, and financial fraud detection [13, 15–17].

For example, the complex event of check kiting is a notorious form of financial fraud [11, 20]. Through a list of consecutive unbalanced checks, fraudsters can transfer money from a bank with insufficient funds to another [20]. Q1 shows an example query to detect potential check kiting fraud events. A check event indicates someone transfers money from one bank account, by writing a check, to another bank account. A Kleene operator '+' is applied to express that the system needs to match one or more check events greedily. A kiting fraud detection system needs to process a massive number of data and identify check kiting fraud events as soon as possible to prevent fraudsters from withdrawing the money away.

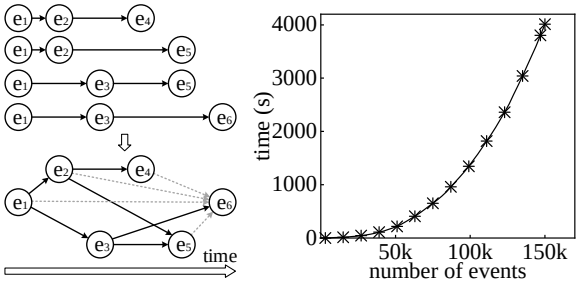
$$\begin{aligned} Q1\ &:PATTERN && check+\ c[\,] \\ &WHERE && c.type = \text{`}not\ covered\text{'}\ AND \\ & && c[i].src = c[i-1].dest \\ &WITHIN && 1\ week\ SLIDE\ 1\ day \end{aligned}$$

Financial trend analysis processes massive financial streaming data and identifies potential opportunities. For example, stock expert traders subscribe to the stock data streams and write queries to find event trends (e.g., increasing, head and shoulder trends) in real-time. Q2 shows a simple example that detects increasing trends of stock prices from a stock event stream using a Kleene operator [13]. Predicate $[id]$ and $s[i].price > ratio * s[i-1].price$ restrict events in a trend to have the same identifier and increasing prices.

$$\begin{aligned} Q2\ &:PATTERN && stock+\ s[\,] \\ &WHERE && [id]\ AND\ s[i].price > ratio * s[i-1].price \\ &WITHIN && 60\ min\ \ SLIDE\ 10\ min \end{aligned}$$

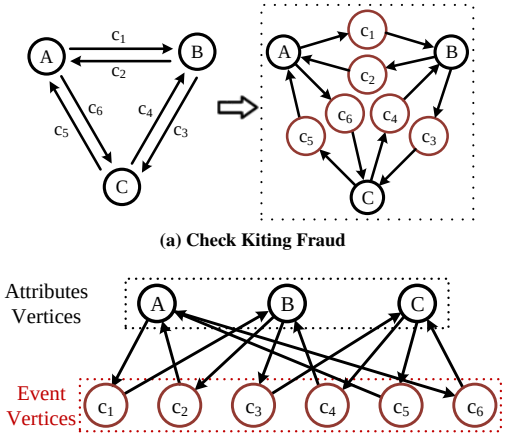
The challenges of building an efficient complete event trend detection system are threefold.

- **Large-scale partial results**. An event sequence may match a sub-pattern or cannot be immediately determined to be complete. The number of such partial results can increase exponentially in real applications, making the efficient storage a challenging issue [13].



(a) CET Graph (b) Graph Construction Time

Figure 1: CET Graph Method



(a) Check Kiting Fraud

(b) ABI Graph of Check Kiting Fraud Example

Figure 2: A Case of Check Kite Fraud

- **High performance requirement**. Applications usually need real-time system response. Efficiently processing the huge partial results is vital for providing low latency.
- **Difficult to parallelize**. A CET is strictly time-ordered. A fine-grained parallelization that processing related events in parallel can cause out-of-time-order issue.

To handle potential huge amount of partial results, Olga et al. [13] propose a compact CET graph, where common paths in the graph can represent common sub-sequences shared by different event trends and avoid redundant storage. Fig. 1(a) shows an example where four CETs are compacted into a CET graph. CETs with common sub-sequences share paths (e.g., (e_1, e_3)) in the graph. It is clear the common path effectively reduces duplicated sub-sequence storage.

However, constructing a CET graph is costly. In the example, when a new vertex e_6 arrives, the system needs to match e_6 with vertices e_4, e_2, e_5, e_3, and e_1 on the CET graph. Most of them fail (denoted as light-gray arrows), while only e_3 succeeds (denoted as a black arrow). Such a CET graph construction needs a worst computation cost of $O(n^2)$ for n events. Fig. 1(b) examines the cost of CET graph construction with large-scale events. It shows the computation cost is prohibitively high for real world applications.

To solve these problems, in this work we propose a novel *Attribute-based Indexing* graph model, called ABI graph, for CET detection. An ABI graph is essentially a bipartite graph with two types of vertices, attribute vertices and event vertices. An edge is generated only between attribute vertices and event vertices based on the predicates in a query. Therefore, the events can be added to the ABI graph independently unlike the conventional graph construction in which events are added to the graph one by one in the order of their arrivals. This makes it possible for parallel graph construction. We further propose a parallel graph construction algorithm that can greatly improve the system performance. Formally, our scheme reduces the cost for graph construction to $(nlog(m))$, where m is the number of attribute vertices and n is the number of event vertices. We design a parallel anchor-based algorithm and a join-based distributed algorithm which efficiently extracts CETs from the ABI graph. Experiment results show our design greatly outperforms previous work.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the ABI graph model. Section 4 proposes the parallel dynamic graph construction method. Section 5 presents several extraction algorithms and a pre-filter algorithm. Section 6 shows the detail of our system design. Section 7 evaluates the performance of this design. Section 8 concludes this paper.

2 RELATED WORK

Traditional complex event processing techniques [2, 9, 10, 18, 23] use Kleene operator to express CET and suffer from memory explosion problem caused by exponentially increasing partial results during CET detection.

To cope with the partial results, Olga et al. [13] propose a compact CET graph model, which represents a CET as a directed path from the vertex standing for the start event to that for the end event. The sub-sequences shared by different event trends use the same path to avoid redundant storage. In a CET graph, the event vertices with an out degree of 0 and the vertices with an in degree of 0 are called start vertex and end vertex, respectively. When receiving a new event, the system traverses forward from each end event vertex to find events that match the new event. If no matching event is found, the system sets the new event as a start event vertex. Though the CET graph can significantly reduce the storage cost, the computational complexity for CET graph construction is $O(n^2)$ for a time window with n events, making such a design impractical in real systems.

Efficient aggregation [15, 16] and multiple queries sharing [14, 17] over event streams attract much recent research efforts. Aggregation is widely required in applications like stock market analysis, cluster monitoring, and traffic management [15]. Greta [15] and COGRA [16] use a graph-based method to aggregate event trends in an online style and support complex Kleene, negation, and nested patterns under various event matching semantics. Multiple queries sharing tends to share computation and storage resources among multiple queries with common sub-patterns. The multiple queries sharing approaches [14, 17] focus on approaching optimal sharing plans by evaluating detection cost based on various models.

3 ATTRIBUTE-BASED INDEXING GRAPH

Let us look at a simple example of check kiting fraud detection shown in Fig. 2(a), where there are six check transaction events $c_1 - c_6$ among

three accounts A, B, and C in three different banks. According to Q1 (Section 1), we can extract three CETs (c_1, c_2, c_6), (c_1, c_3, c_5, c_6), and (c_1, c_3, c_4). If we regard each account and event as a vertex, we can use edges to show the relationships between event vertices and account vertices. For example, event c_1 indicates a check transaction from the source account A to the account B and this transaction can be represented as $A \rightarrow c_1 \rightarrow B$. Hence, we can generate an *attribute-based indexing* (ABI) graph as shown in Fig. 2(b). Using ABI graph we can extract all the CETs similar to that using the CET graph.

For simplicity, we give the following definitions before discussing ABI graph construction and CET extraction.

Attribute Value Range. An attribute value range $AS = \{val_1, val_2, ..., val_p\}$ is a collection of values of an attribute.

Event. $Event = (timestamp, \langle\ attr_1, ..., attr_g\ \rangle)$ represents instances of interest happened at a time point. Each event has a time attribute that denotes its occurrence time and a series of attributes to describe it.

Event Stream. The event stream $I = (e_1, e_2, ...)$ is an infinite sequence of events continuously produced by a source. The events in I are in time order as they occur.

Query. The system receives queries from users to extract CETs of interest. A CET Query has the following form,

PATTERN P [**WHERE** Θ] **WITHIN** w **SLIDE** s

where P is an event pattern, the conditional expression Θ is defined by *WHERE* clause, and a w-length time window slides every s units of time. An event pattern P is a sequence of Kleene operator or event types. A Kleene operator which is defined by an event type followed by a '+' means matching one or more events of the given event type. The *WHERE* clause defines a conditional expression by a series of predicates. Only the CETs that match the expression can be accepted.

Predicate. A predicate θ is a conditional expression made up by one of the six comparators $\{=, \neq, >, \geq, <, \leq\}$. A predicate can always be converted to the form:

$$f(e_{next}.attr, h(tr)) \rightarrow bool$$

where f represents a comparator function, e_{next} indicates the next relevant event. Traditional query languages [21] usually use $e[i]$ to represent e_{next} (as shown in Q1 and Q2), where i is the length of an event trend tr. The function h usually has two categories. One always returns a constant value (e.g., the predicate $c[i].type =$'*not covered*' in Q1). The other adjacent function in the form $h(e[i-1].attr)$ returns a value only related with the last event of the given event trend (e.g., the predicate $c[i].src = c[i-1].dest$ in Q1).

Complete Event Trend. An event trend $tr_l = (e_{i_1}, e_{i_2}, ..., e_{i_l})$, $1 \leq i_1 < i_2 < ... < i_l$ is an event sequence that match the given query. For an event trend tr_l, if there is not any event e_k in the time window that can be added to an event sequence $tr_{l+1} = (e_{i_1}, ..., e_{i_l}, e_k)(k > i_l)$ or $(e_k, e_{i_1}, ..., e_{i_l})(k < i_1)$ to match the given query, the event trend tr_l is regarded as complete.

Related Events or Attributes. For an adjacent predicate θ, if $f(e_j.attr, h(e_i.attr)) = true$ and $j > i$, then e_j is relevant to e_i (or e_i matches e_j). Similarly, if $f(attrValue, h(e_i.attr)) = true$, then we call the attribute value *attrValue* is relevant to e_i (or e_i matches *attrValue*).

We define an ABI graph based on the above definitions.

Table 1: Notations

Notation	Explanation
as	a subrange of attribute value range
e_t	an event with a relative timestamp t
tr_l	an event trend with l events
θ	a predicate
fe or *te*	a from-edge or a to-edge
ev or *av*	an event vertex or an attribute vertex
lval	the calculated value on an event corresponding to the lefthand side of θ
rval	the calculated value on an event corresponding to the righthand side of θ
[min,max)	the lower (include) and upper (exclude) bound of attribute values
n	the number of events in a time window
m	the number of attribute vertices in the graph

Attribute-based Indexing* (ABI) Graph**. Given an input event stream I and a query Q. The notation θ is a predicate in Q. An attribute-based indexing graph *G={EV, AV, FE, TE}* is a directed bipartite graph. *AV* and *EV* are the two vertex sets representing attribute values and events, respectively. Specially, an attribute vertex can represent a single attribute value or a set of attribute values. According to the directions of the edges, we divide the edges into two categories. *FE* is a set of ***from-edges and each from-edge points from an attribute vertex to an event vertex, representing that the attribute value of the event is equal or included in the corresponding attribute values of the attribute vertex. *TE* is a set of ***to-edges*** and each to-edge points from an event vertex to an attribute vertex, indicating one or more relevant attribute values of the corresponding event. Table 1 lists the notations in this work.

4 GRAPH CONSTRUCTION

Based on our ABI design, we propose three optimization technologies. Firstly, we propose a dynamic ABI graph construction method to minimize the generation of attribute vertices. Second, based on the continuity of to-edges of each event vertex, we apply a new edge representation, which can reduce the complexity from $O(mn)$ to $O(nlog(m))$. Finally, we design an effective parallel algorithm to greatly enhance the performance of the graph construction.

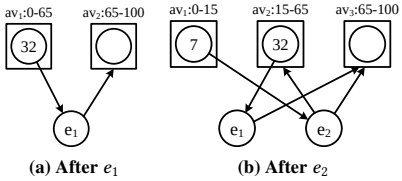
4.1 Dynamic Graph Construction

In the dynamic graph construction, attribute vertices are generated on demand when receiving events from the input stream. The graph construction is simple with the '=' predicate. In such case each event is only involved with two attribute values. At the beginning, there is no attribute vertex. When an event arrives, we find or create two corresponding attribute vertices for these two values. We can use a hash structure to store these attribute vertices while the time cost for adding an event to the graph is then $O(1)$.

For non-equal predicates, dynamic construction becomes more complicated. Each event may match a number of attribute values instead of only one. Hence, we propose composite attribute vertex, which represents a range of attribute values. We stipulate that a to-edge pointing to a composite attribute vertex is equivalent to it pointing to all the attribute values from an event vertex. This means that values in a range that is represented by a composite attribute

Table 2: Attribute Values

event	e_1	e_2	e_3	e_4	e_5	e_6
attr	32	7	15	35	40	17



(a) After e_1 **(b) After e_2**

Figure 3: Split Example

vertex must be all relevant to the corresponding event. If a composite attribute vertex is irrelevant to an event, there are no edges connecting them. When a composite attribute vertex contains both relevant and irrelevant values of an event (violate condition), then the composite attribute vertex needs to be split into two composite attribute vertices, one of which contains all the relevant values of the event, while the other contains all the irrelevant values.

To simplify the discussion, we will use a predicate with '>' comparator as default. The cases of using other types of predicates are similar. Let AS: $[min, max)$ be the range of attribute values, and θ be the predicate. For an event e, its relevant attribute value set can be expressed as a range $[h(e.attr) + \delta, max)$, in which δ is the precision of AS. We use δ here just to ensure the unification of range format in order to explain our method. In the following discussion, we assume that attribute values are integers and $\delta = 1$.

At the beginning of graph construction, we generate a single composite attribute vertex which represents the whole range $[min, max)$. When an event arrives, the new event may violate the stipulation with certain current composite attribute vertices (violate condition). Then, we must split these violated attribute vertices for the stipulation.

Theorem 1. *Let av_p be a composite attribute vertex which represents a range of attribute values $av_p : [min_p, max_p)$, e be a new event. The value of the righthand side of the '>' comparator is r_x. If $min_p < r_x < max_p$, then av_p is a violated composite attribute vertex. Meanwhile, we have: 1) all attribute values in $[min_p, r_x + 1)$ are irrelevant to e. 2) All attribute values in $[r_x + 1, max_p)$ are relevant to e.*

Proof. According to the predicate, the relevant attribute value range of e is $as_{r_x} = [r_x + 1, max)$. The subtraction and intersection of the related attribute value range of av_p and as_{r_x} are $as_{p_1} = [min_p, r_x + 1)$ and $as_{p_2} = [r_x + 1, max_p)$. Since $min_p < r_x < max_p$, both set as_{p_1} and as_{p_2} are not empty. The range as_{r_x} contains both relevant values (in as_{p_2}) and irrelevant values (in as_{p_1}) of event e (violate condition). Hence, av_p is a violated composite attribute vertex. □

Based on the proposition, we need to split vertex av_p into two composite attribute vertices which respectively represent the relevant and irrelevant attribute value ranges $av_{p_1} : [min_p, r_x + 1)$ and $av_{p_2} : [r_x + 1, max_p)$. A composite attribute vertex is always split into two attribute vertices with two disjoint value range. Therefore, the attribute value ranges of all composite attribute vertices are disjoint. When an event arrives, based on the above proportion, the corresponding value range of a violated composite attribute vertex must contain the result of the righthand expression of the predicate on the event. There can only be at most one such composite attribute vertex when all value ranges are disjoint. To efficiently find this violated composite vertex, we can use a sorted structure (e.g., tree map or skip list) to store all composite vertices.

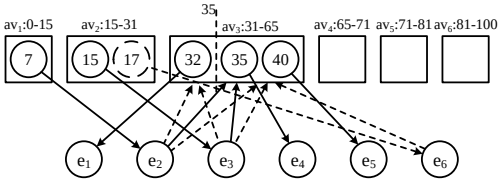


Figure 4: Dynamic Construct

After splitting a violated composite attribute vertex (or no violation occurs) for a new event e, we must generate edges for its corresponding event vertex e. We first find a composite attribute vertex av_f, whose corresponding attribute value set contains the attribute value of e (i.e., the value on the lefthand side of the '>' comparator), and generate a from-edge from av_f to ev. The relevant attribute value set of e is $[h(e.attr) + 1, max)$. We then find all the relevant attribute vertices and generate to-edges from ev to each of these relevant composite attribute vertices.

Example. Assume there are six events with corresponding attribute values (Table 2) and the predicate is $e[i].attr > e[i-1].attr*2$. At the beginning, we only have a composite attribute vertex to represent set $AS : [0, 100)$. When the event e_1 arrives, the relevant attribute value set of e_1 is $as_1 : [65, 100)$. Obviously, the set AS violates the stipulation with e_1. We then split AS into two subranges $av_1 : [0, 65)$ and $av_2 : [65, 100)$. Since $e_1.value = 32$ (in av_1), we generate a from-edge $av_1 \rightarrow e_1$ and a to-edge $e_1 \rightarrow av_2$. Similarly, we split composite attribute vertex $[0, 65)$ and generate two to-edges for e_2, as shown in Fig. 3.

By splitting, we successfully eliminate all violated composite attribute vertices. Before we split a violated attribute vertex, however, there may be several edges connected with the vertex. We must well manage these edges after we split a composite vertex to ensure the correctness of the ABI graph.

Let av be a violated composite attribute vertex, fes be the from-edges that point from av to several event vertices, and tes be the to-edges that point from several event vertices to av. After splitting av, we obtain two new composite attribute vertices av_1 and av_2 for the new event. 1) After the split, the existing fes edges are divided into two groups according to the attribute values of the corresponding events and then are added to av_1 and av_2, respectively. 2) Since all the values in av are relevant to the source event vertices of edges in tes, values in av_1 are also relevant to those vertices. After the split, we then need two new to-edges to point from each of those event vertices to av_1 and av_2 respectively.

Example. Figure 4 shows the ABI graph of the case shown in Table 2 after events e_1 to e_5 are added to the graph. The attribute value set $0 - 100$ has been split into six subranges. In the figure, solid lines indicate the edges connected between the attribute vertices and their corresponding event vertices. To illustrate the split process

more clearly, we omit the to-edges that do not point to the attribute vertex $v_{ori} : 31 - 65$. Since the attribute value of e_6 is 17, according to the predicate, the relevant attribute values of e_6 are greater than 34. Therefore, we split the attribute vertex av_3 with the value 35 (the vertical dashed line) into two vertices av_{31} and av_{32} with attribute values in the ranges $[31, 35)$ and $[35, 65)$, respectively. After splitting, we copy the original to-edges $e_2 \rightarrow av_3$ and $e_3 \rightarrow av_3$ and point them to both av_{31} and av_{32} and also point the original from-edges to av_{32} (as indicated by the dashed edges in the figure).

4.2 Complexity

Theorem 2. *Given n events and m attribute vertices, both the time complexity and the memory complexity of graph construction are $O(n)$ for equal predicates and $O(mn)$ for non-equal predicates.*

Proof. For equal predicates, we always generate only a from-edge and a to-edge for each event. We use a hash structure to store dynamic attribute vertices. Therefore, the time and memory complexities of graph construction are $O(n)$.

For non-equal predicates such as '>', the memory complexity is $O(mn)$ since we may generate up to m edges for each event. Because splitting attribute vertices raises cost for copy operations, the cost for graph construction for non-equal predicates increases. In the following, we prove the time complexity is also $O(mn)$.

The time cost of graph construction mainly consists of two parts. One part is the time cost for finding correct attribute vertices when each event arrives. Since we use a sorted structure to store attribute vertices, it costs $O(log(m))$ time to find a correct attribute vertex. The other part is the cost for generating edges. Hence, we have

$$CPU_{construct} = O(nlog(m)) + CPU_{edges} \tag{1}$$

We define original edges as the edges that are not copied from another edge. For example, in Fig. 4, the edges $e_6 \rightarrow av_{32}$ and $av_2 \rightarrow e_3$, are not generated from copy operation. In contrast, the edge $e_2 \rightarrow av_{32}$ is not an original edge because it is copied from $e_2 \rightarrow av_3$. All the edges in an ABI graph originate from original edges. Specially, we stipulate an original edge originate from itself for unification.

Firstly, we consider only the complexity of generating to-edges. The complexity is mainly made up of two parts,

$$CPU_{te} = CPU_{ori} + CPU_{copy} \tag{2}$$

For each event, we will generate up to m to-edges. Therefore, we have $CPU_{ori} = O(mn)$. Let $OE = \{oe_1, oe_2, ..., oe_k\}$ be the set of all original to-edges that are generated during graph construction, and CE_i be the set of edges that originate from oe_i. Then, the total number of edges is computed by

$$|TE| = \Sigma_1^k |CE_i| \tag{3}$$

When splitting an attribute vertex, we copy a to-edge connected with the attribute vertex to two new to-edges. This process involves a constant number of operations. Given NS copy operations, we have

$$CPU_{copy} = O(NS) \tag{4}$$

Let us take a closer look at oe_i and its child edge set CE_i. Based on the algorithm, all edges in CE_i have the same source event vertex and various destination attribute vertices. Since we only split no more than one attribute vertex when an event comes, at most one edge in CE_i will be copied to two new edges every time we split an attribute vertex. Let ns_i be the number of copy operations that are processed on the edges in CE_i. Each time an edge from CE_i is copied to two new edges, the size of CE_i increases by one. Therefore,

$$ns_i = |CE_i| - 1 \tag{5}$$

For all original edges and their child edge sets, we have

$$NS = \Sigma_1^k |ns_i| \tag{6}$$

Sum up Eqs. (2) - (6), and we have

$$CPU_{te} = O(|TE|) + O(mn) = O(mn) \tag{7}$$

To estimate the complexity of generating from-edges, we can assume that we apply the same copy operations for from-edges. Then, we also obtain the time complexity $O(mn)$. Actually, from-edges involve fewer operations than to-edges. Hence, the time complexity to generate from-edges is less than $O(mn)$ and the time complexity of graph construction for non-equal predicates is $O(mn)$. □

4.3 Range To-edges Representation

Even though our dynamic graph construction algorithm has tried to reduce the number of attribute vertices. The time and memory complexity of these algorithms is still $O(mn)$. However, we observe: 1) the production of to-edges contributes to most computation and memory costs; 2) the attribute vertices pointed by to-edges that start from the same event vertex are continuous (e.g., $e_3 \rightarrow \{av_{31}, av_{32}, av_4, av_5, av_6\}$ in Fig. 4). Values in the union of the ranges represented by these attribute vertices are relevant to the event (e.g., greater than $2 * e_3.value$). Based on these observations, we propose to use a range to-edges representation to reduce the time and memory costs for graph construction.

Due to the contiguity of destination vertices of to-edges from the same event vertices, we can use a range to represent these edges. We use two pointers to represent all to-edges of the same event vertex. One points to the start attribute vertex; the other to the end attribute vertex. The attribute vertices between the start vertex and the end vertex are all the destination of to-edges from the event vertex.

Generating the two pointers is simple. We use a sorted structure to store these attribute vertices. Under a predicate with comparator '>', the end attribute vertex is apparently the largest attribute vertex (e.g., av_6). The start attribute vertex is the smallest attribute vertex whose corresponding value range is larger than the righthand value of the predicate on the event (e.g., av_{31}). According to the above analysis, only a cost of $O(log(m))$ is needed to find these two pointers. Therefore, we can successfully reduce the time and memory complexity of graph construction to $O(nlog(m))$ and $O(n)$.

4.4 Parallel Algorithm

The above scheme dynamically generates a minimal number of necessary attribute vertices. However, since we must search these attribute vertices while splitting violated composite vertices, read-write conflicts may occur if we process events in parallel. According to Proposition 1, when an event arrives, we split the violated composite attribute vertex by *rval* of the event (i.e., value of function h above). If we can calculate these values of all events first, we can generate all attribute vertices immediately rather than generate these vertices when searching relevant attribute vertices. Hence, we can eliminate the read-write conflicts.

Algorithm 1: Parallel Dynamic Graph Construct

Input: EventStream I, AttributeSet AS, Predicate θ
Output: Graph *graph*

1 $EV \leftarrow \emptyset$; $FE \leftarrow \emptyset$; $TE \leftarrow \emptyset$; $AV \leftarrow \emptyset$;
2 *fromCaches* $\leftarrow \emptyset$; *toCaches* $\leftarrow \emptyset$;
// 1st: generate event vertices and tuple caches
3 *partitionEvents* $\leftarrow$ *randPartition(I)*;
4 **for** *partition in partitionEvents* **do**
5 *fCache* $\leftarrow \emptyset$; *toCache* $\leftarrow \emptyset$;
6 **for** *e in partition* **do**
7 *eVertex* $\leftarrow$ *createEventVertex(e)*;
8 $EV \leftarrow EV \cup$ *eVertex*;
// calculate values from both sides
9 *lval*,*rval* $\leftarrow$ *extractVals(e,*θ*)*;
10 *fCache* $\leftarrow$ *fCache* $\cup$ *(eVertex, lval)*;
11 *toCache* $\leftarrow$ *toCache* $\cup$ *(eVertex, rval)*;
12 *fromCaches* $\leftarrow$ *fromCaches* $\cup$ *fCache*;
13 *toCaches* $\leftarrow$ *toCaches* $\cup$ *toCache*;
// 2nd: generate all attribute vertices
14 *rvals* $\leftarrow$ extract *rval* from *toCaches* and de-duplicate;
15 *rvals* $\leftarrow$ *parallelSort(rvals)*;
16 *AV* $\leftarrow$ combine adjacent *rval* to generate attribute vertices;
// 3rd: generate from-edges
// map *lvals* to attribute vertices
17 *fedges* $\leftarrow$ *mapAttrVertex(fCache, AV)*;
18 *fpartitions* $\leftarrow$ *partitionByAv(fedges)*;
19 **for** *partition in fpartitions* **do**
20 **for** *(av,ev) in partition* **do**
21 $FE \leftarrow FE \cup (av \rightarrow ev)$;
// 4th: generate to-edges
22 *tedges* $\leftarrow$ *mapAttrVertex(tCache, AV)*;
23 *tpartitions* $\leftarrow$ *randPartition(tedges)*;
24 **for** *partition in tpartitions* **do**
25 **for** *(ev,av) in partition* **do**
26 *trange* $\leftarrow$ *(av, last(AV))*;
27 $TE \leftarrow TE \cup$ *trange*;
28 *graph* $\leftarrow$ *{EV, AV, FE, TE}*;
29 return *graph*.

We optimize the dynamic scheme in parallel. The parallel algorithm has four steps. Firstly, in the stream processing stage, we randomly partition events from the stream into different processing units and process them in parallel. In each processing unit, for each event e we calculate the value of the lefthand side ($lval = e.attr$) and the righthand side $rval = h(e.attr)$ of the comparator in the predicate. We generate an event vertex ev for e. Then, we store two lists of tuple $(lval, ev)$ and tuple $(ev, rval)$ for each event.

To generate all attribute vertices, in the second step, we need to obtain a distinct and sorted set of $rval$. We use a parallel sort algorithm (e.g., odd-even sort) and a distinct algorithm for efficiency. Then, we split the range AS into a list of value ranges based on the value in the distinct $rval$ set and generate composite attribute vertices for them. Since the set of $rval$ is sorted in advance, the set of attribute vertices is also sorted by their corresponding value ranges.

In the third step, we transform each tuple in the $(lval, ev)$ tuple list and the $(ev, rval)$ tuple list to edges. For each tuple, we find the attribute vertex that contains $lval$ or $rval$ and generate two new tuple list (av_f, ev) and (ev, av_t). The two elements in a new tuple are the source vertex and destination vertex of an edge. The procedure for generating an edge only involves search operations on the attribute vertices. Hence, we can process them in parallel.

We store these edge tuples and construct the graph in the final step. In this step, the main problem is the synchronization problem caused by the storage structure of the graph. For example, we use a list to store all the outgoing edges of a vertex (adjacent list) in our implementation. If we partition the (av_f, ev) tuple list by ev, the write-write conflicts may occur when two from-edges have the same source attribute vertices since they must be written into the edge storage of the same vertex. Therefore, we can either partition the (av_f, ev) tuple list by av_f or use locks on each av_f. With our range to-edges representation, generating to-edges is simple. We use a tuple $(av_t, avs.last)$ to represent all the to-edges of an event vertex, where $avs.last$ means the biggest attribute vertex in the set.

Algorithm 1 presents the detail of our parallel dynamic ABI graph construction method. Each event in the stream I is processed in a random processing unit (lines 3-13). After we complete processing all the events, we extract values $rval$ from tuples in the list and sort them into a distinct value set (lines 14-15). We generate attribute vertices from each adjacent tuple of $rval$ in the sorted distinct value set (lines 16). We find the proper attribute vertex for each tuple of $(lval, ev)$ tuple list and $(ev, rval)$ tuple list (line 17, line 22). To avoid synchronization problems, we partition $fedges$ by its corresponding attribute vertices and store all from-edges (lines 18-21). We store to-edges in the range representation (lines 23-27). All these steps are processed in parallel.

Complexity. Let p be the number of process unit, m and n be the number of attribute vertices and event vertices, the time complexity of the parallel graph construction is

$$CPU = CPU_1 + CPU_2 + CPU_3 + CPU_4 \tag{8}$$

where CPU_i be the time complexity of the ith step. In the first step, the system uses $O(n/p)$ time to generate event vertices and tuples for n events. In the second step, the distinct operation uses $O(n/p)$ time and generate $m-1$ $rval$ as split boundaries. The complexity of the parallelsort operation depends on the choice of parallel algorithm, which may be $O(m^2/p)$ with an odd-even sort algorithm. In the last two steps, the main time costs come from searching in the sorted attribute vertices. Therefore, the time complexities are $O(nlog(m)/p)$. Considering that $m \leq n$, the time complexity of the parallel graph construction is

$$CPU = O(nlog(m)/p) + O(m^2/p) \tag{9}$$

5 CET EXTRACTION

In this section, we first define the CET path, which is a path from a start event vertex to an end event vertex and represents a CET in the ABI graph. Then, we present a parallel anchor-based CET extraction algorithm to travel through all CET paths and extract all CETs from the ABI graph. We also introduce a join-based traversal algorithm to support huge scale of events on top of Spark [22], a popular distributed processing system. A pre-filter algorithm is used to find out all start event vertex before traversal.

5.1 CET Path

Figure 2 shows an example of an ABI graph, where two relevant event vertices are connected by an attribute vertex, a from-edge and a to-edge, e.g., $c_1 \rightarrow B \rightarrow c_3$. Consecutive relevant event vertices and the attribute vertices between them form a path in the graph. This path represents a CET consisting of the corresponding events.

Theorem 3. *Let ev_i, ev_j be two event vertices and av_p be an attribute vertex in an ABI graph with the predicate θ. Let e_i, e_j be two events that respectively correspond to ev_i, ev_j. If there are a to-edge $ev_i \rightarrow av_p$ and a from-edge $av_p \rightarrow ev_j$ in the graph, and e_i, e_j satisfy the condition $e_i.timestamp < e_j.timestamp$, then e_i and e_j are relevant on θ.*

Proof. Beginning with the vertex ev_i, since a to-edge of an event vertex points to the relevant attribute value of its corresponding event, the to-edge $ev_i \rightarrow av_p$ indicates that the corresponding attribute values of av_p are relevant to e_i. Similarly, the from-edge $av_p \rightarrow ev_j$ represents that the attribute value of ev_j is equal to or included in av_p. Thus, the attribute value of ev_j is relevant to ev_i. Meanwhile, the constraint $e_i.timestamp < e_j.timestamp$ is satisfied. Therefore, e_i and e_j are relevant. □

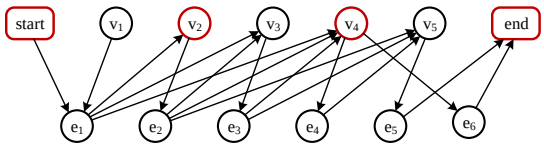
According to Theorem 3, two relevant events are connected by a from-edge and a to-edge in an ABI graph. Therefore, if we start from a start event vertex and repeatedly traverse by from-edges and to-edges until we cannot find more relevant events, we can obtain a CET path. The corresponding events of event vertices in this path make up a CET.

Definition 1. ***(CET path)*** *In an ABI graph, a CET path is a path representing a CET matched by predicate θ. The first event vertex of the CET path (also called **start (event) vertex**) corresponds to the start event of the CET, while the last event vertex of the CET path (**end (event) vertex**) corresponds to the end event of the CET. Two CET paths are equal when and only when they pass the same event vertices.*

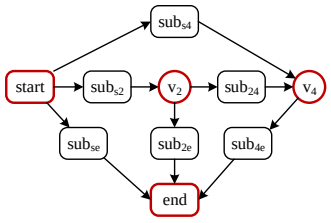
5.2 Anchor-based CET Extraction

Through the ABI graph, we successfully represent CETs as CET paths. To extract a CET from the graph, we only need to traverse from each start vertices to end vertices. Olga et al. [13] have tried to use a H-CET algorithm that combines a BFS-based algorithm and a DFS-based algorithm to achieve a trade-off between memory and computation consumption. However, this method is not applicable to a large-scale of events because of its poor graph construction method and unpractical graph partition method. In this section, we propose an anchor-based CET extraction method with a lightweight but balance graph partition method.

Our anchor-based method first selects a certain number of attribute vertices as anchors. In addition, we add two types of special anchors: start anchor and end anchor. A start anchor always points to a number of start vertices. All end vertices point to the end anchor. Then, the anchor-based method has the following two steps: 1) Starting from each anchor and ending in anchors, we apply the BFS-based method to extract sub-sequences. Since there is no conflict between the processes from anchors, this stage can be processed in parallel. 2) Starting from each start anchor and ending in the end anchor, we apply



(a) Original ABI Graph



sub_{s2}	(e_1)
sub_{s4}	$(e_1),(e_1,e_3)$
sub_{se}	(e_1,e_3,e_5)
sub_{24}	$(e_2),(e_2,e_3)$
sub_{2e}	(e_2,e_5)
sub_{4e}	$(e_4,e_5),(e_6)$

(b) After BFS stage

Figure 5: Anchor-based Detection

the DFS-based method to traverse among anchors and concatenate sub-sequences into CETs. Much work [1, 5] has addressed at parallel DFS. In this work, we implement the parallel DFS-based method in a work-stealing manner for speeding up.

Example 4. Figure 5 shows an example of anchor-based extraction. The nodes $v_1 - v_5$ are five attribute vertices. We select v_2 and v_4 as anchors. In the BFS stage, we start BFS-based extraction from the start anchors v_2 and v_4, respectively. After the BFS stage, we obtain a new temporary graph shown in Fig. 5(b). In Fig. 5(b), the vertex sub_{xy} contains all the sub-sequences that start from anchor x and end in anchor y. Then, we start the DFS stage from the start anchor and obtain the paths that reach the end anchor. We concatenate sub-sequences in the path under a join manner. For example, we extract the path $start \rightarrow sub_{s2} \rightarrow sub_{2e} \rightarrow end$ and obtain CETs $sub_{s2} \times sub_{2e}$.

Work-stealing in DFS. During a DFS task, a processing unit preserves two state values, the current path $path$ that has been passed through and the next vertex v that is to be visited. When the processing unit visits v, v is pushed into $path$ (forwarding state). After all the vertices behind v are visited, we must pop v out from the current $path$ (backing state) and continue to visit other vertices. Hence, a DFS task can be split into two disjoint tasks. One inherits the forwarding state and traverses the vertices behind v. The other inherits the backing state and traverses other vertices. Our work-stealing DFS algorithm is based on this observation. an idle processing unit sends a signal to a busy processing unit. Then, the busy processing unit that is ready to visit v splits its task on v. The idle processing unit copies states from the busy one and starts to process the forwarding child task. The busy processing unit continues to process the backing child task.

Load Balance. Generally, we ensure the load balance of our algorithm by selecting a sufficient number of anchors, which is equivalent to the number of BFS tasks during the BFS stage. When the number of tasks is far greater than the number of processing units, a simple scheduling algorithm [12] is enough to ensure the load balance. The load imbalance is slight during the DFS stage because of work-stealing. However, load imbalance is still possible on the task that starts from a start anchor. Since all CETs start from

start vertices, the BFS and DFS that start from the start vertices can cause more cost than others. Therefore, to avoid load imbalance, we need enough start anchors. In our implementation, we set the number of start anchors much greater than that of processing units.

5.3 Join-based Distributed CET Extraction

The anchor-based algorithm achieves high performance on CET extraction but fails to handle large-scale of events with memory limit. We design a distributed join-based CET extraction algorithm.

Popular distributed graph processing systems [7] often use a vertex-central programming model for graph processing. They are typically built on general distributed processing systems (e.g., Spark [22]) and use join operators to transfer messages between vertices iteratively. For most graph algorithms, such as PageRank, the vertex-central programming model is quite efficient since they will not produce many messages during each iteration. However, a CET extraction system may produce partial results of hundreds of times the number of events. Such a model needs to transfer these partial results between vertices iteratively, leading to heavy network pressure.

Broadcast join is a variety of the general join operator. This kind of join operator is optimized when a big dataset joins with an immutable small dataset. When a broadcast join is performed, the processing system firstly broadcasts the small dataset to all processing units. Each copy of the small dataset is organized as a map (usually a hashmap). Then, for each item in the big dataset, the processing system finds all items with the same key in the small dataset by searching the local map of the small dataset, where only $O(1)$ time is needed. This process can be completed by a map operator. Although data transmission is still required, the broadcast join operator has two main benefits. First, it avoids transferring the big dataset. Second, when the small dataset is involved in multiple join operations, the processing system broadcasts the small dataset only once, which makes the cost of broadcasting negligible. Traditional graph algorithms usually are trapped by the large scale of the graph [3] and dynamic attributes of vertices. Hence, broadcast join is not applicable to these algorithms. However, the main bottleneck of CET extraction in ABI graph is the partial results of exponential growth rather than the scale of the immutable ABI graph. Therefore, we propose to use broadcast join operator and design a distributed CET extraction scheme.

We have two datasets during graph traversal. The first one is the ABI graph, which is broadcast to each processing unit at the beginning of CET extraction. The other is the dataset of all CET paths. The initial value of the path dataset is the set of all start event vertices. Then, we broadcast join the path dataset with the graph dataset to traverse through the graph and extend corresponding event sequences. Once a path reaches an end event vertex, this path is a CET path. We repeat this until all the CET paths are extracted.

There are two join steps in a traversal operation. Let $FE = \{(av, ev)\}$ and $TE = \{(ev, av)\}$ be the from-edges and to-edges in an ABI graph. Let $TR = \{(ev_l, tr_l)\}$ be the current uncompleted CET path dataset, where ev_l is the last event vertex of the path tr_l that consists of l event vertices. First, we join dataset TR with TE on the key ev and obtain a new dataset $Temp = \{(av, tr_l)\}$, which indicates we traverse from event vertices to attribute vertices. Second, we join dataset $Temp$ with dataset FE on the key av and get a new dataset $TR_{new} = \{(ev_{l_{new}}, tr_{l+1})\}$, where we extend the path tr_l by the current event vertex $ev_{l_{new}}$ and obtain a new path tr_{l+1}. If the event vertex $ev_{l_{new}}$ is an end event vertex, then path tr_{l+1} is a CET path. We filter out all CET paths and repeat the above step until the size of path dataset TR reaches zero.

Algorithm 2: Join-based CET extraction

Input: Graph *G (EV,AV,FE,TE)*, Start Event Vertices *starts*, End Event Vertices *ends*
Output: CETs

```
1  bGraph(bevs, bavs, bfes, btes) ← broadcast(G);
2  epaths ← starts − ends; cnt ← count(epaths);
3  CETs ← extractEvents(starts ∩ ends);
4  while cnt > 0 do
5      apaths ← join(epaths, btes) ;          // join from-edges
6      epaths ← join(apaths, ftes) ;          // join to-edges
7      completePaths← filterUncomplete(epaths,ends);
8      uncompletePaths ← filterComplete(epaths,ends);
9      CETs ← CETs ∪ extractEvents(completePaths);
10     cnt ← count(uncompletePaths);
11     epaths ← uncompletePaths;
12 return CETs.
```

The broadcast join optimizes CET extraction for three reasons: 1) with broadcast join, we need not to transfer partial results between processing units, and avoid a large amount of communication cost; 2) the scale of the ABI graph is relatively small, especially with our range to-edges representation; 3) during CET extraction, the ABI graph is immutable.Hence, we only need to broadcast the graph dataset only once.

Algorithm 2 presents the broadcast join-based CET extraction method in detail. The system broadcasts the graph at the beginning (line 1). The initial paths is calculated from the start (end) event vertices (line 2). The system repeatedly traverses the entire graph until obtaining all the CET paths (line 5-12).

Complexiy.Since we avoid shuffle operations by broadcast join during message transfer, the main communication costs come from broadcasting the graph, which are equal to $O((|AV| + |EV| + |FE| + |TE|)p)$, where p is the number of executors in the Spark cluster. The main costs of our distributed CET extraction algorithm come from graph traversal. We traverse the ABI graph under a BFS-like strategy. The process may iterate up to $2 * \max_1^k(|tr_i|)$ rounds and the total time complexity is $\Sigma_{i=1}^{k}|tr_i|$, where tr_i is a CET and k is the number of CETs that will be extracted.

5.4 Pre-Filter

Extracting CETs by traversing the ABI graph has another issue. If the algorithm does not start from a start vertex or end in an end vertex, the result sequence cannot be completed. We further design an algorithm to find the start vertices and the end vertices which saves a significant amount of unnecessary event sequences during CET extraction.

Based on the definition of CET, a CET sequence cannot be a subsequence of another CET sequence. Let ev_x be an event vertex and e_x be its corresponding event, we have the following observation: 1) If $\exists ev_i \in EV$, its corresponding event e_i matches e_x, then ev_x cannot be a start vertex; 2) If $\exists ev_j \in EV$, ev_x matches its corresponding event e_j, then ev_x cannot be an end vertex.

Algorithm 3: Fast Pre-Filter

```
Input: Graph G (EV,AV,FE,TE)
Output: Start and End Vertices (starts, ends)
1  starts ← set(EV); ends ← set(EV); metas ← map();
2  for ev in EV do
3      for av in outgoing(ev,TE) do
4          if av not in metas then
5              mo ← -1;
6              for ev in outgoing(av,FE) do
7                  if ev.timestamp > mo then
8                      mo ← ev.timestamp;
9              metas ← metas ∪ (av → (mo,mo+1));
10         meta ← metas.get(av);
11         if ev.timestamp < meta[0] then
12             ends ← ends − ev;
13         if ev.timestamp < meta[1] then
14             meta[1] ← ev.timestamp;
15 for (av → meta) in metas do
16     if meta[1] < meta[0] then
17         for ev in outgoing(av,FE) do
18             if ev.timestamp > metas[1] then
19                 starts ← starts − ev;
20 return (starts, ends).
```

We prove the correctness of the observation as follows. Let $tr = \{e_{x_1}, e_{x_2}, ..., e_{x_l}\}$ be an event trend, ev_i and ev_j be two event vertices. If e_i matches e_{x_1}, we can obtain a new event trend $tr_1 = \{e_i, e_{x_1}, ..., e_{x_l}\}$, where tr is a sub-sequence of tr_1. In this case, any event trend that starts with e_{x_1} cannot be complete. If e_{x_l} matches e_j, we can obtain a new event trend $tr_2 = \{e_{x_1}, ..., e_{x_l}, e_j\}$, where tr is a sub-sequence of tr_2. In this case, the event trends ending with e_{x_l} cannot be complete.

Based on this observation, we can easily obtain all start and end event vertices using join-based traversal. However, at the beginning of traversal, the scale of event sequences is very small. Hence, a distributed method may lead to unnecessary distributed overhead. Therefore, we propose an optimized centralized pre-filter method which can find out all start and end event vertices efficiently before the graph is broadcast to all processing units.

The algorithm has two steps. First, starting from each event vertex ev_x, following by its outgoing edges, we traverse to its relevant attribute vertices. Second, following by the outgoing edges of these attribute vertices, we can traverse to a series of event vertices evs (connectivity condition). For each event vertex in evs, if the timestamp of its corresponding event is larger than that of ev_x (time condition), this event vertex cannot be a start vertex (observation 1). If such a relevant event vertex is found, ev_x cannot be an end vertex (observation 2).

To avoid missing any event vertex, we must judge each event vertex with all relevant event vertices. When an attribute vertex is pointed by more than one event vertices, its outgoing edges are accessed repeatedly, raising significant unnecessary costs.

To solve the problem, we propose the following optimization. Actually, to check whether an event vertex is a start/end vertex, we only need to check whether it can match a later event vertex or can be matched by a previous event vertex.

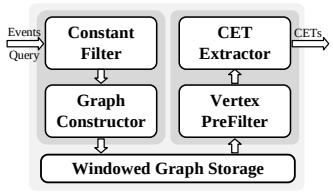


Figure 6: Graph-Based CET Detection System

To achieve this goal, in the first step, we record the metadata for each relevant attribute vertex av_y, including two timestamps mo and mi, where mo is the maximum timestamp of the corresponding events of the outgoing event vertices of av_y, while mi is the minimal timestamp of the corresponding events of the ingoing event vertices of av_y. We calculate mo the first time when av_y is visited. Each time we visit av_y, we update mi and check the time condition. Let e_x be the corresponding event of ev_x. If $e_x.timestamp < mo$, there is at least one event vertex that can match the time condition with e_x, i.e., there is at least one relevant event of e_x. Hence, ev_x is not an end vertex. In the second step, we compare mi and mo in each attribute vertex av_y. Let avs be the outgoing vertices of av_y. If $mi < mo$, all the corresponding events of vertices in avs whose timestamp is larger than mi can be matched by no less than one event. Hence, their corresponding event vertices are not start vertices.

Algorithm 3 presents the details of the fast pre-filter algorithm. We compute mo only once for each relevant attribute vertex, thus avoiding repeated access of the outgoing event vertices of the relevant attribute vertices. Each to-edge is accessed only once (lines 3-14). Each from-edge is accessed no more than twice (lines 6-8, lines 17-19). The total computation cost is $O(2|FE| + |TE|)$, linear to the number of edges.

6 IMPLEMENTATION

We implement our parallel designs with Java and the distributed algorithms on Spark [22] v3.0.0. We make the source code publicly available[1]. Figure 6 shows that our system consists of three modules: 1) the graph construction module including a constant filter and a graph Constructor. It receives a query and an event stream as inputs and generates vertices and edges; 2) the windowed graph storage module which receives the vertices and edges generated by the graph constructor and stores graphs for each window; and 3) the CET extraction module including a vertex prefilter and a CET extractor.

We design two different ABI graph construction methods (Section 4). The dynamic graph construction method is a sequential algorithm, while the parallel algorithm is suitable for both multi-core machines [8] and popular distributed processing systems. Therefore, we implement both versions of parallel methods on top of the multi-thread model and Spark. We use a treemap which leverages a red-black tree to store all attribute vertices and support the dynamic nature of attribute vertices in the dynamic algorithm.

For the parallel algorithm, we use an array (or RDD) to store these attribute vertices. Generating all attribute vertices requires combine adjacent split values ($rvals$), which is easy in a shared

[1]https://github.com/CGCL-codes/DynamicTG

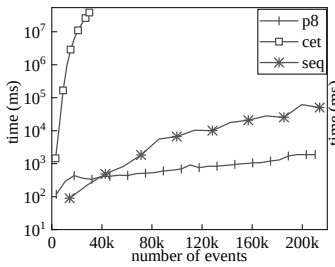


Figure 7: Construction Time (ABI vs CET Graph) (Q2)

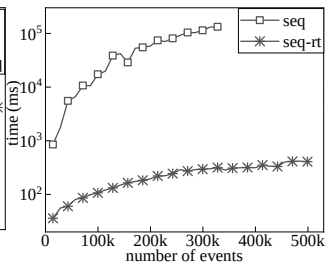


Figure 8: Construction Time with Range To-edges (Q2)

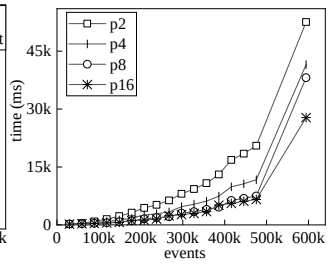


Figure 9: Construction Time with Various Parallelism (Q2)

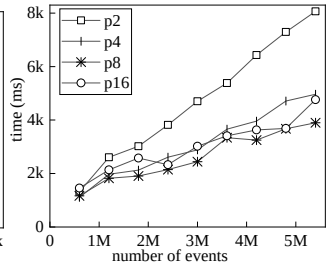


Figure 10: Construction Time with Various Parallelism (Q1)

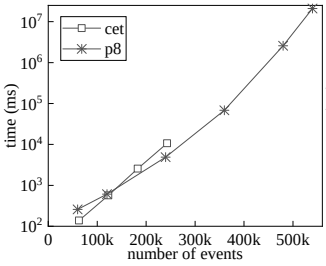


Figure 11: CETs Extraction Time (Q1)

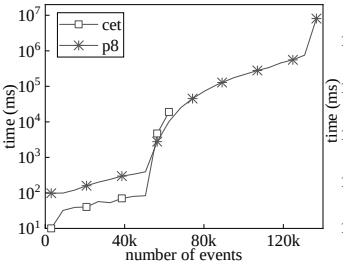


Figure 12: CETs Extraction Time (Q2)

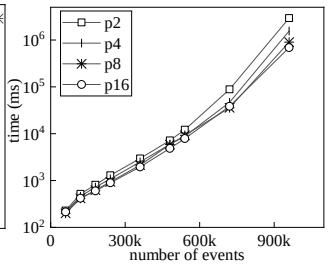


Figure 13: Extraction Time with Various Parallelism (Q1)

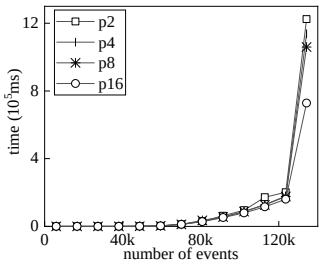


Figure 14: Extraction Time with Various Parallelism (Q2)

memory environment. However, Spark does not provides an operator to support this operation directly. Therefor, we complete this process by three steps: 1) We generate index for each item in the *rvals* RDD by *zipWithIndex* operator. 2) We decrease each index by one using a *map* operation. 3) We combine adjacent *rvals* by a *join* operation on the original *rvals* RDD and calculated RDD.

The CET Extractor receives an ABI graph as input and extracts all the CET paths as output. Before performing the CET extraction algorithm, the processing system first uses the pre-filter algorithm to find all the start and end event vertices. We implement the work-stealing DFS algorithm on a thread pool with fixed number of threads. We deploy the join-based algorithm on a Spark cluster. The system broadcasts the ABI graph and organizes event and attribute vertices as hashmaps. Since there are more than one out-going edges for each vertex, the edges are organized as multi-hashmaps, whose values are arrays of edges.

7 PERFORMANCE EVALUATION

We deploy our system on a cluster with six machines, each equipped with a 16-core 2.4GHz Xeon CPU, 64GB RAM, and 1TB HDD. We compare our design with the CET Graph method [13].

We use two datasets to evaluate our design, including a check kite dataset and a stock dataset. Since it is difficult to find a public dataset to detect check kite fraud, we design and implement an event generator to create the check dataset. Each event in the check kite dataset contains a timestamp attribute as well as the source and the destination bank account attributes. The values of bank account attributes are randomly selected from a pre-defined bank account set. We omit the type attribute because it is mainly used in filter operations which are well supported in traditional stream processing systems [4, 19]. We perform Q1 (depicted in Section 1) on the dataset. We have also crawled one-hour data from Shanghai Stock Exchange [6]. In the stock dataset, each event contains a timestamp attribute and a price attribute. We perform Q2 on this dataset.

We mainly evaluate our graph construction scheme in two aspects: 1) we compare both the sequential and parallel method with the CET graph method by varying the number of events on the stock dataset. We show the effect of the range to-edge representation by leveraging it on the sequential dynamic method; 2) we evaluate the parallel algorithm on both the check kite and stock datasets.

For CET extraction, we compare our anchor-based algorithm with the CET graph method under various scales of events and examine the performance of our parallel algorithm. We evaluate the performance and scalability of the join-based algorithm. In addition, we evaluate the performance of the fast pre-filter algorithm and the impact of the number of anchors on memory and CPU time.

We examine the average *CPU time* [13] of our design. For graph construction, the average CPU time is measured from the end of a window to the end of graph construction. For CET extraction, the average CPU time is measured from the end of graph construction to the time when we finish extracting all the CETs. We also evaluate the relative memory cost of our design by counting the event references during CET extraction.

Before presenting the results, we first briefly explain the symbols. The term 'cet' represents the CET graph method, 'seq' denotes our dynamic graph construction method, 'seq-rt' denotes the dynamic method with range to-edges representation, and 'px' denotes our parallel methods, where 'x' is the parallelism.

Figure 7 shows the time for graph construction. In the experiment, we evaluate our parallel method, our dynamic method, and the CET graph method. The time of the CET graph method increases rapidly and reaches several hours when the number of events is hundreds of thousands. In contrast, our algorithm can efficiently process more than 100k events in a few seconds, revealing three

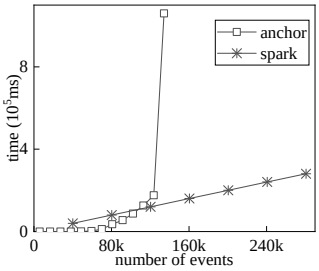


Figure 15: Parallel and Distributed Extraction (Q2)

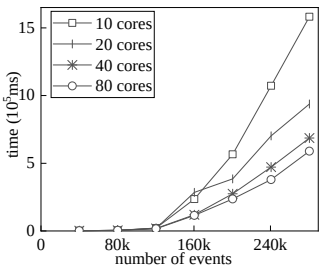


Figure 16: Distributed Extraction (Q2)

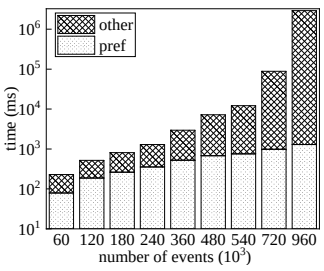


Figure 17: Proportion of Pre-Filter (Q2)

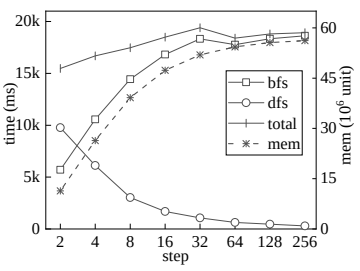


Figure 18: Effect of Number of Anchors (p8)

orders of magnitude improvements. The result also shows that our parallel algorithm achieves 40× speedup than the sequential design when the number of events reaches 200k. In addition to parallelism, this is also because of the invariance of attribute vertices which can make good use of cache and avoid extra overhead to support dynamic nature.

Figure 8 shows the graph construction time of the dynamic algorithm with range to-edges representation is two orders of magnitude shorter than that of the basic dynamic algorithm. Meanwhile, the basic algorithm cannot support over 300k events due to the memory limit, while the optimized algorithm can support 500k or more with much better performance.

Figures 9 and 10 show the performance of our parallel method under different parallelisms on non-equal and equal comparator. Our algorithm with 16 threads significantly reduces the time cost of that with two threads by 41% (Q1 with 5M events) and 48% (Q2 with nearly 600k events), respectively.

We compare the performance of our anchor-based extraction algorithm with eight threads and the CET graph method by varying the number of events. Figures 11 and 12 show the CET graph method cannot support large-scale events and it cannot finish graph construction even in several hours. It is clear that our scheme greatly outperforms the CET graph method.

Figures 13 and 14 show our algorithm with 16 threads reduces the time cost of the algorithm with two threads by 76% (Q1 with 1,000k events) and 40% (Q2 with 140k events), respectively.

Figure 15 compares the performance of our parallel anchor-based extraction with eight threads and distributed join-based extraction with 40 CPU cores. The distributed algorithm fails to beat the parallel one when the number of events is less than 90k because of distributed overheads. However, the distributed algorithm can handle over 250k events with acceptable CPU time, while the parallel algorithm cannot support over 120k events. Figure 16 shows the good scalability of the join-based algorithm, which can effectively reduce the time cost by 63% (280k events) when the number of CPU cores used in the cluster increases from 10 to 80.

Figure 17 plots the time of the pre-filter stage (*perf*) during CET extraction. When the number of events increases to 960k, the time of pre-filter accounts for only 0.04% of the total time.

Figure 18 shows the influences of the number of anchor vertices. In the experiment we vary the number of anchors and examine time and memory cost for both the BFS and the DFS stages. The result shows that the time cost for the BFS stage increases while that for DFS stage decreases as the number of anchors increases. The result also shows the memory cost has similar trend with the time cost for the BFS stage. The total time cost has a slow increasing trend, demonstrating the efficiency of our parallel DFS algorithm.

8 CONCLUSIONS

In this paper, we propose ABI, a novel attribute-based indexing graph model for efficient CET detection. We design a parallel dynamic algorithm to construct ABI graphs from the input event stream. We also propose a parallel anchor-based algorithm and a distributed join-based algorithm to efficiently extract CETs from ABI graphs. We further propose an efficient pre-filter algorithm to filter out unnecessary partial results. Experiment results show that our design greatly outperforms existing designs.

ACKNOWLEDGMENTS

This research is supported by the National Key Research and Development Program of China under grant No.2018YFB1004602, NSFC under grants Nos. 61972446, 61422202.

REFERENCES

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of SC, 2015*.
[2] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of SIGMOD, 2008*.
[3] Hanhua Chen, Hai Jin, and Shaoliang Wu. 2016. Minimizing Inter-Server Communications by Exploiting Self-Similarity in Online Social Networks. *IEEE TPDS* 27, 4 (2016), 1116–1130.
[4] Hanhua Chen, Fan Zhang, and Hai Jin. 2017. Popularity-aware Differentiated Distributed Stream Processing on Skewed Streams. In *Proceedings of ICNP, 2017*.
[5] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. 2008. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *Proceedings of ICPP, 2008*.
[6] Shanghai Stock Exchange. 2021. http://english.sse.com.cn/.
[7] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of OSDI, 2014*.
[8] Ruixuan Li, Zhiyong Xu, Wanshang Kang, Kin Choong Yow, and Cheng-Zhong Xu. 2014. Efficient Multi-keyword Ranked Query over Encrypted Data in Cloud Computing. *FGCS* 30, 1 (2014), 179–190.
[9] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of SIGMOD, 2009*.
[10] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. 2012. High-performance complex event processing over XML streams. In *Proceedings of SIGMOD, 2012*.
[11] U.S. Attorney's Office. 2016. Finacial fraud. https://www.justice.gov/usao-ndoh/pr/three-cleveland-women-indicted-165000-check-kiting-scheme-0.
[12] Bo Peng, Zhipeng Lü, and Tai Chiu Edwin Cheng. 2015. A tabu search/path relinking algorithm to solve the job shop scheduling problem. *Computers & Operations Research* 53 (2015), 154–164.

[13] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A. Rundensteiner. 2017. Complete Event Trend Detection in High-Rate Event Streams. In *Proceedings of SIGMOD, 2017*.

[14] Olga Poppe, Chuan Lei, Lei Ma, Allison Rozet, and Elke A. Rundensteiner. 2021. To Share, or not to Share Online Event Trend Aggregation Over Bursty Event Streams. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*.

[15] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2017. GRETA: Graph-based Real-time Event Trend Aggregation. *PVLDB* 11, 1 (2017), 80–92.

[16] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2019. Event Trend Aggregation Under Rich Event Matching Semantics. In *Proceedings of SIGMOD, 2019*.

[17] Olga Poppe, Allison Rozet, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2018. Sharon: Shared Online Event Sequence Aggregation. In *Proceedings of ICDE, 2018*.

[18] Medhabi Ray, Elke A. Rundensteiner, Mo Liu, Chetan Gupta, Song Wang, and Ismail Ari. 2013. High-performance complex event processing using continuous sliding views. In *Proceedings of EDBT, 2013*.

[19] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. In *Proceedings of SIGMOD, 2014*.

[20] Wikipedia. 2021. Check kiting. https://en.wikipedia.org/wiki/Check_kiting, 2021.

[21] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of SIGMOD, 2006*.

[22] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of NSDI, 2012*.

[23] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of SIGMOD, 2014*.