DI-Tree: A Dual-ended Interval Tree for Efficient Event Matching in Content-based Pub/Sub Systems

Junshen Li[†], Haiyang Ren[§], Zhengyu Liao[§], Wanghua Shi[†], Shiyou Qian[†¶*], Guangtao Xue[†¶*], Jian Cao[¶†] and Zhonglong Zheng[§]

[†]*Shanghai Jiao Tong University, Shanghai, China.* [§]*Zhejiang Normal University, Zhejiang, China.*

[¶]*Shanghai Key Laboratory of Trusted Data Circulation and Governance in Web3, Shanghai, China.*

[*]*Corresponding authors, Email: qshiyou@sjtu.edu.cn, gt_xue@sjtu.edu.cn*

***Abstract*—Content-based publish/subscribe systems have the capability to achieve fine-grained data distribution, rapidly forwarding data from publishers to subscribers with specific requirements. The event matching algorithm is a fundamental component, quickly searching for subscriptions that match an event based on constraints defined by subscribers. As data scales continue to expand, there is a heightened demands for more efficient, robust, and versatile event matching algorithms. In this paper, we propose a novel data structure called the Dual-ended Interval Tree (DI-Tree). Firstly, given the splitting point, this data structure classifies intervals into three categories based on the joint distribution of their left and right endpoints in the attribute value domain. Furthermore, the DI-Tree utilizes blue and green nodes to store these three categories of intervals, resulting in enhanced indexing efficiency. Moreover, by utilizing the DI-Tree to efficiently search matching and unmatching intervals, we develop innovative forward and backward event matching algorithms. Additionally, to enhance matching efficacy and reduce memory usage, we introduce key optimization techniques, focusing on improving node balance and bitset optimization. We conduct extensive experiments to evaluate the performance of DI-Tree. When compared with five state-of-the-art event matching algorithms, the DI-Tree demonstrates an average reduction of up to 76.8% in terms of matching time. This significant improvement highlights the effectiveness of our proposed strategies and the potential of DI-Tree in optimizing event matching performance.**

***Index Terms*—Publish/subscribe, event matching, dual-ended, interval tree, performance**

I. Introduction

With the rapid development of society and the onset of the big data era, the amount of data has been continuously increasing [1]. As a result, there is a growing demand for more precise regulations regarding data distribution patterns. At the same time, the requirement for low-latency data distribution persists, posing significant challenges for large-scale data distribution systems [2], [3].

To address these challenges, content-based publish-subscribe systems have gained widespread adoption in various fields [4], such as emergent situations [5], stock markets [6], and online advertising [7]. The content-based publish-subscribe system comprises publishers, subscribers and proxy servers (also called brokers). Specifically, the proxy server maintains constraints issued by subscribers in the form of subscriptions, executes event matching algorithms, and identifies subscribers interested in published events. The primary objective of event matching algorithms is to minimize the processing latency of events on the proxy server and enhance the data distribution throughput of the entire system.

The following definitions are presented to provide an understanding of the commonly used terms in the publish/subscribe system. Firstly, events are representations of occurred incidents, expressed in the form of attribute-value pairs. The number of pairs is referred to as the event size. The collection of attributes associated with events is defined as $\mathbf{A} = \{a_1, a_2, ..., a_m\}$, where m denotes the dimensionality of content spaces. Secondly, subscriptions are used to express the interests of subscribers in events. Each subscription s has an ID, usually consisting of multiple interval predicates in conjunctive normal form: $s = \{c_1 \wedge c_2 \wedge ... \wedge c_k\}$. An interval predicate c is represented by a triple $c = \{a_i, l, h\}$, where a_i denotes the attribute on which the predicate is defined, and l and h represent the left and right values respectively. Since other kinds of predicates can be transformed into intervals, we use it as the data model for subscriptions. The number of predicates k in subscriptions is referred to as the subscription size. Lastly, the process of event matching involves the identification of all subscriptions in the set $\mathbf{S} = \{s_1, s_2, ..., s_n\}$ that match a given event e. The runtime required for the algorithm to complete the event matching process is defined as the matching time.

Multiple factors influence the performance of event matching algorithms, encompassing characteristics of the subscription set (e.g., the number of subscriptions, subscription size, predicate width, distribution of predicate values, frequency of subscription insertions and deletions, dimension of content space, cardinality of attributes), and characteristics of the events (e.g., event size, distribution of event attributes, distribution of event values, and probability of matching between events and subscriptions). In general, when designing novel data structures and matching algorithms, it is crucial to consider diverse application scenarios to ensure the algorithm's efficiency, generality and robustness.

A plethora of innovative event matching algorithms have been proposed by researchers, which can be broadly divided into three categories. The first group comprises forward algorithms that rely on maintaining a counter for each subscription throughout the matching process, with the objective of identifying and tallying all satisfied predicates. Notable examples of forward matching algorithms include TAMA [8], OpIndex [9] and k-Index [10]. The second group comprises backward matching algorithms such as REIN [11], GEM [12], Ada-

REIN [13], HEM [14] and WPA-REIN [15]. These algorithms strive to identify all unsatisfied predicates on each dimension and designate the corresponding subscriptions as unmatching in the bit set. Lastly, a new matching approach, known as ensemble event matching, has emerged by combining multiple event matching algorithms, surpassing the performance of the original algorithms. Representatives of ensemble event matching include PEM [16], fgPEM [17], and Comat [18].

To further improve and stabilize the performance of event matching in content-based publish/subscribe systems, we introduce a novel data structure called the Dual-ended Interval Tree (DI-Tree). Firstly, we classify intervals into three categories based on the joint distribution of their left and right endpoints. Furthermore, we organize the three types of intervals into two types of tree nodes, depending on whether the high and low values of intervals cross the splitting point. Through recursive node splitting, the DI-Tree is constructed. Moreover, we propose forward and backward matching algorithms based on the DI-Tree. These algorithms involve searching for matching and unmatching predicates respectively. Additionally, in terms of bitset optimization, we enhance the counting-based forward matching algorithm by converting arithmetic counting operations into logical operations, resulting in an optimized performance.

We conduct extensive experiments to evaluate the performance of DI-Tree using synthetic and real-world stock datasets. The metric experiments involve setting different parameters, including the number of subscriptions, subscription size, predicate width, event size, and attribute distribution. Compared to five baselines, namely TAMA [8], OpIndex [9], REIN [11], Ada-REIN [13], and fgPEM [17], the experiment results show that DI-Tree has an average reduction in matching time of 69.4%, 76.8%, 74.4%, 73.6%, and 39.8% respectively.

The main contributions of this paper are summarized as follows:

- We propose a novel data structure called DI-Tree, which classifies interval predicates based on the joint distribution of their left and right values, effectively addressing the issue of distribution skew.
- Building upon DI-Tree, we develop forward and backward matching algorithms, which efficiently search for matching and unmatching predicates respectively.
- We propose optimization strategies for the developed matching algorithms from multiple perspectives, including bitset optimization.

The remainder of this paper is organized as follows. We provide a brief overview of the related work in Section II. We describe the detailed design of DI-Tree in Section III. We propose forward and backward event matching algorithms in Section IV. In Section V, we provide and analyze the experiment results. Finally, in Section VI, we conclude the paper and discuss potential future research directions.

II. Related Work

In this section, we first review the related work. We classify the existing matching algorithms into three categories: forward matching, backward matching, and ensemble matching. Then, we provide a concise overview of the matching process and delve into the discussion of representative algorithms for each category.

A. Forward Matching Algorithms

Forward matching algorithms usually use the counting method, which tracks the number of satisfied predicates in each subscription in the matching process. The main goal of the search process is to identify all predicates satisfied by the event values. Once all predicates in a subscription are satisfied, it is identified as a match. It is worth noting that this process involves a forward-looking search, with the aim of identifying all the satisfied predicates.

TAMA [8] and OpIndex [9] are two prominent forward matching algorithms. TAMA proposes a two-layer matching table to map predicates to corresponding buckets. It also utilizes a counting method to establish associations among common predicates in each subscription. Similarly, OpIndex first indexes attributes with the lowest frequency appearing in subscriptions, followed by indexing operators. Moreover, OpIndex efficiently filters out unmatching subscriptions with null attribute as their primary attribute, thereby reducing duplicate counts.

B. Backward Matching Algorithms

In contrast to forward matching algorithms, backward matching algorithms aim to identify all unsatisfied predicates in the matching process. These unsatisfied predicates are used to label the corresponding subscriptions as unmatching in a bitset. Therefore, the unmarked bits after this process represent the matching subscriptions. This approach allows for a more efficient and accurate identification of matching subscriptions. Instead of trying to find all the matching subscriptions, it focuses on eliminating the unmatching ones first. This significantly reduces the time and resources needed for the matching process.

REIN [11] constructs indexes for both the lower and upper bounds of interval predicates defined on the same attribute. During event matching, it primarily involves traversing buckets and performing bit marking operations. On the other hand, GEM [12] presents an alternative backward matching algorithm that incorporates a caching method to expedite the removal of unmatching subscriptions, thereby reducing redundant calculations. To effectively handle varying workloads, Ada-REIN [13] accomplishes dynamic performance adjustment by seamlessly transitioning between exact matching and approximate matching. It enables adaptive performance tuning and formulates appropriate adjustment plans in response to fluctuating workloads.

C. Ensemble Matching Algorithms

Each matching algorithm has its own unique advantages, disadvantages, and suitable application scenarios. In order to leverage the strengths of different matching algorithms and mitigate their limitations, ensemble matching methods have been proposed [16]–[18]. These approaches involve running multiple event matching algorithms concurrently, resulting in a novel matching solution that demonstrates enhanced performance.

Comat [18] is a composite matching approach that is based on event set partitioning. It utilizes a neural network model to predict the matching time of events across multiple matching algorithms. Then, it selects the algorithm with the smallest predicted matching time to execute the matching task for the event. PEM [16] categorizes predicates based on their width and stores them in the data structures of REIN [11] and TAMA [8]. It employs two sub-matching algorithms for parallel matching, and intersects the matching results from each component to obtain the final matching outcome. Building upon PEM, fgPEM [17] addresses high-dimensional subscriptions by partitioning them at a finer granularity. By dividing subscriptions into sub-subscriptions, fgPEM effectively leverages matching algorithms with complementary behaviors, thereby further improving performance.

In the face of the rapid growth of big data, our work shares the common goal of improving event matching efficiency, similar to previous studies. The main difference between our work and existing algorithms lies in the classification and storage strategy used for interval predicates that are defined on the same attribute. Our approach involves classifying and storing predicates based on the joint distribution of their low and high values, taking into full consideration the challenge posed by distribution skewness.

III. Design of DI-Tree

A. Basic Idea

As discussed in Section II, most event matching methods construct their data structures based on attributes to efficiently separate, organize and index predicates. The existing data structures can be understood and differentiated from the perspective of interval splitting. For instance, REIN [11] utilizes a high-low splitting method, which divides an interval into two elements based on its lower and upper bounds. It then indexes these two types of elements separately. On the other hand, TAMA [8] employs interval width splitting, where fixed sub-intervals are partitioned in a manner similar to a complete binary tree. For any interval, TAMA selects the minimal number of pre-defined sub-intervals that can cover it.

From this perspective, we can outline the challenges that these methods still face. Generally, most data structures partition interval predicates and use buckets to index and store them. Subsequent matching algorithms locate the target buckets and perform searching within them. However, many algorithms have not taken into consideration the issue of data distribution. In cases of severe predicate skew, the concentration of storage has a negative impact on matching performance. Excessive bucket scanning can directly result in a decrease in matching speed.

To address the aforementioned issues, we propose a novel data structure called DI-Tree. The basic idea of DI-Tree is manifested in two aspects. Firstly, DI-Tree utilizes an iterative splitting technique to classify interval predicates into three distinct categories based on the joint distribution of low and high values, aiming to improve filtering efficiency. The advantage of this approach lies in its comprehensive consideration of distribution skewness, which enables the achievement of superior and more stable performance. Secondly, DI-Tree is capable of dynamically splitting based on node size, which prevents the high load caused by excessively large nodes to further enhance the matching performance.

Notably, we summarize a comparative analysis between the traditional interval tree [19] and the DI-Tree. Traditional interval trees utilize a recursive bisection method, focusing on the interval's left endpoint as the critical element. This approach is effective in scenarios where both the high and low values of the interval are on the same side of the splitting point. In contrast, the DI-Tree adopts a dual-ended strategy for classifying intervals, enabling it to efficiently separate intervals that cross the splitting point. This dual-ended approach significantly reduces the number of checks needed at both ends of the interval, thereby enhancing filtering efficiency.

B. Data Structure of DI-Tree

For discussion simplicity, we suppose that the value domain of attributes is normalized to the range [0, 1]. Given an attribute a_i and a splitting point sp, the DI-Tree divides all interval predicates defined on a_i into three categories based on the joint distribution of interval's low and high values relative to sp: 1) L-L intervals, $l \leq h \leq sp$, where both low and high values are less than or equal to sp, 2) R-R intervals, $sp < l \leq h$, where both low and high values are greater than sp, and 3) L-R intervals, $l \leq sp < h$, where low and high values cross sp. These three types of intervals are non-overlapping and cover all types of intervals on the entire value domain. Therefore, any predicate interval belongs to only one category, realizing a one-to-one mapping.

In accordance with the aforementioned splitting, we define the concepts of blue and green nodes in DI-Tree. The first two categories restrict the high and low values of intervals within the same range, which can be represented by the same type of node, called "blue nodes". The third category refers to intervals whose high and low values belong to two different ranges. Since the event values have opposite matching semantic for the high and low values of intervals, these intervals are stored separately based on their lower and upper bounds, stored in "low-green nodes" and "high-green nodes" respectively, similar to REIN [11].

This dual-ended interval splitting scheme can be recursively applied. For example, starting with 0.5 as the initial splitting point, intervals in blue nodes fall within the ranges of either $[0, 0.5]$ or $(0.5, 1]$. These can be sub-divided further into three categories by employing 0.25 and 0.75 as additional splitting points respectively. Conversely, intervals in green nodes are only divided into two categories. This classification process is repeated recursively until either the height of the DI-Tree reaches a threshold or the size of the leaf nodes meets a specific limit. Figure 1 shows an example of a three-layer DI-Tree. For ease of description, each node is labeled with a number. "L" type green nodes represent low green nodes, while "H" type green nodes represent high green nodes.

C. Inserting Intervals

The procedure of inserting the intervals of subscriptions into the DI-Tree is outlined as follows. Initially, the DI-Tree's root is denoted as a blue node, which encompasses all intervals,

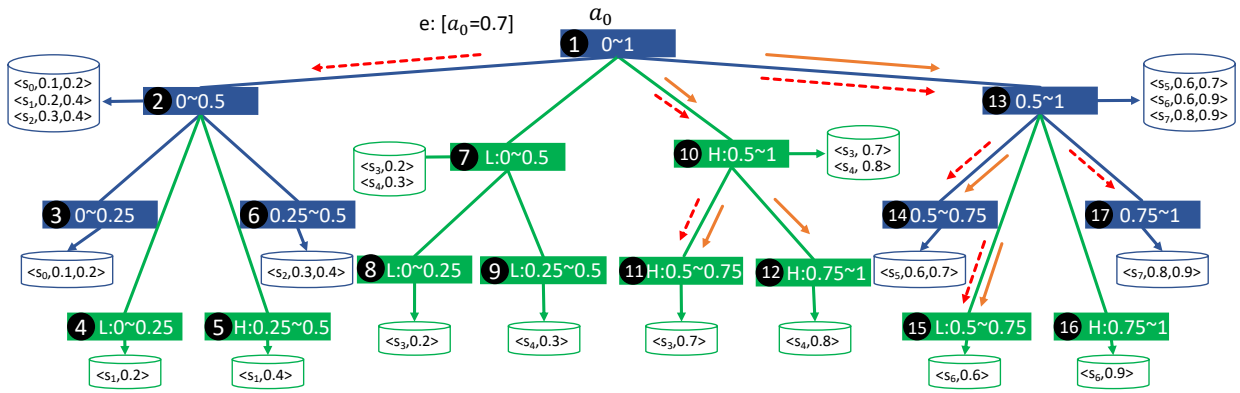

Fig. 1. The data structure of an example of a three-layer DI-Tree

TABLE I
SAMPLE SUBSCRIPTIONS

SubID	a_0	SubID	a_0
s_0	$[0.1, 0.2]$	s_4	$[0.3, 0.8]$
s_1	$[0.2, 0.4]$	s_5	$[0.6, 0.7]$
s_2	$[0.3, 0.4]$	s_6	$[0.6, 0.9]$
s_3	$[0.2, 0.7]$	s_7	$[0.8, 0.9]$

exempt from any filtration process. Similar to the first layer of TAMA, it does not need to store data. Subsequent blue nodes, in contrast, store triples, each encompassing subscription IDs (SubIDs) along with the high and low values of predicates. Additionally, the low green nodes store subscription IDs and the low values of predicates, whereas the high green nodes store subscription IDs and the high values of predicates.

Considering a scenario where the splitting point (sp) is 0.5 in the first layer (the median value within the interval is utilized as sp), we index 8 subscriptions listed in Table I, as exemplified in Figure 1. The exemplified events encompass an attribute a_0 with a value range of [0, 1], and we store the SubIDs within the corresponding buckets associated with the pertinent nodes. For instance, when indexing subscription s_1, the interval for s_1 on a_0 is [0.2, 0.4], with both the low and high values being lower than 0.5, thus it is initially directed to node 2 (left blue node). Node 2, which is blue, continues the recursive splitting process by setting $sp = 0.25$. Given that at this juncture, the low value of s_1 0.2 is less than sp 0.5 while the high value 0.4 is greater than sp, the low and high values are stored separately in nodes 4 (low-green node) and 5 (high-green node), respectively. The indexing process of the remaining subscriptions in the DI-Tree is analogous. It is important to note that the leaf nodes on the third layer can be further subdivided as needed.

We further analyze the unique properties of the DI-Tree. The subtree that originates from a green node in a DI-Tree is essentially a binary search tree. Similarly, the tree formed by all blue nodes is also a binary search tree. Therefore, the fundamental nature of DI-Tree can be described as a pseudo-binary tree. This configuration essentially merges multiple smaller binary search trees into a singular, expansive binary search tree. As a result, in a DI-Tree with h layers, the number of blue nodes is $2^h - 1$. Meanwhile, the number of green nodes can be derived recursively as follows:

$$G(h) = G(h-1) + 2 * T(h-1) + 2 * 2^{h-2} \quad (1)$$

$$T(h) = 2 * T(h-1) + 2 * 2^{h-2} \quad (2)$$

For $h \geq 1$, $G(h)$ represents the total number of green nodes in the DI-Tree with h levels, $T(h)$ represents the number of green nodes at level h, and 2^{h-2} represents the number of blue nodes at level $h-1$. Solving the above equation, we get:

$$G(h) = (h-2) * 2^h + 2 \quad (3)$$

$$T(h) = (h-1) * 2^{h-1} \quad (4)$$

Therefore, the number of green nodes in the DI-Tree of depth h is $(h-2) * 2^h + 2$.

D. Dynamic Layering for Load Balance

Given a tree height h, the DI-Tree is considered to be a complete tree. However, skewed interval predicates can result in some branches in the DI-Tree having a small number of predicates, leading to many empty nodes. Conversely, other branches may have too many predicates, resulting in leaf nodes with an excessive number of predicates. This can cause leaf nodes to become enlarged, increasing search cost. To address this imbalance, a dynamic layering scheme is introduced instead of a static one. Under this approach, nodes are not recursively split unless their size exceeds a threshold v. This method effectively solves the problem of excessively small or large nodes, reducing spatial occupancy and improving filtering efficiency. In the implementation of DI-Tree, we use this dynamic scheme.

IV. Matching Algorithms

The DI-Tree can support both forward and backward matching algorithms. The commonality of these two types of matching algorithms is that forward matching involves the identification of matching predicates on each dimension, whereas backward matching pertains to the detection of unmatching predicates. Any forward and backward matching algorithm can be mutually transformed through bit sets. From this perspective, we can derive a method for designing a forward or backward matching algorithm based on the same data structure.

A. Forward Matching Algorithm fDI-Tree

Taking the DI-Tree depicted in Figure 1 as an example, we illustrate the execution logic of the forward matching algorithm called fDI-Tree and simulate the process by the

Algorithm 1: The forward matching on blue nodes in fDI-Tree

Input: The event value e_i on attribute a_i, the blue node BN_i on attribute a_i
Output: Cnt: the number of satisfied predicates for subscriptions

```
1  if BN_i is a leaf node then
2      for Each pair < s.ID, c_j > stored by BN_i do
3          if c_j.l ≤ e_i ≤ c_j.h then
4              Cnt[s.ID] + +;
5  else if e_i ≤ the median in BN_i then
6      Perform the blue node forward matching on the leaf blue
         child node of BN_i;
7      Perform the low-green node forward matching on the
         low-green child node of BN_i;
8  else
9      Perform the blue node forward matching on the right
         blue child node of BN_i;
10     Perform the high-green node forward matching on the
         high-green child node of BN_i;
```

orange solid-line arrows. Given an event value of 0.7, fDI-Tree searches for candidate nodes that may contain satisfied predicates, finding four leaf nodes: 11, 12, 14, and 15. These nodes are then divided into three cases based on the number of further comparisons required. Firstly, the high predicate values stored in node 12 are satisfied by 0.7, without any further evaluation needed. Secondly, in node 11, comparisons are required to determine whether the high predicate value is greater than or equal to 0.7 for a match. Similarly, node 15 requires comparing to check if the low predicate value is less than or equal to 0.7, Lastly, node 14 requires comparisons of both high and low predicate values for a match.

Overall, there is only one blue node that needs to be processed and two comparison checks are required, while one or more green nodes are entirely matching or require one comparison check. Specifically, on the blue node of the branch, a portion of unsatisfied predicates can be further filtered out through distribution classification. On the green node of the branch, its two child nodes may either have one entirely matching, and the other requiring recursive processing, or one entirely unmatching, and the other requiring recursive processing.

The pseudocode for forward matching on blue nodes is shown in Algorithm 1. When dealing with the blue node, if it is a leaf node, the predicates stored in the node are iterated, performing two comparisons on each predicate to determine if it is a match (lines 1-4); otherwise, the event value must fall within the range responsible for one of its blue child nodes and green child nodes, and then recursively process these two child nodes (lines 5-10).

When processing the green node, there are three cases that require separate processing due to the opposite matching semantic of low-green and high-green. Case 1: if it is a green leaf node, each stored predicate is traversed and the comparison between the predicate and the event value is performed. Case 2: if it is a low-green branch node, either the left low-green child node needs to be recursively processed and the right low-green child node does not match at all, or the left low-green child node matches entirely and the right low-green child node needs to be recursively processed. Case 3: if it is a high-green branch node, either the left high-green child node needs to be recursively processed and the right high-green child node matches entirely, or the left high-green child node does not match at all and the right high-green child node needs to be recursively processed.

B. Backward Matching Algorithm bDI-Tree

We also use the DI-Tree illustrated in Figure 1 to describe the process of the backward matching algorithm called bDI-Tree, and depicted by the red dashed-line arrows. Initially, bDI-Tree searches for candidate nodes containing potentially unmatching predicates for the event value of 0.7, identifying nodes 2, 11, 14, 15, and 17. Similar to fDI-Tree, these nodes are then divided into three cases based on the number of further comparisons required. In case 1, the predicates stored in nodes 2 and 17 are unmatching, thus their corresponding subscriptions can be directly marked. In case 2, nodes 11 and 15 require comparison to determine if the high value of the predicate is less than 0.7 and if the low value of the predicate is greater than 0.7 respectively. In case 3, node 14 needs to iterate and compare each stored predicate no more than twice to determine if it is an unmatching predicate.

It is easy to observe that for backward matching, on the branch blue node, it is certain that unmatching predicates can be further found in the child nodes based on the distribution splitting of DI-Tree. The two blue child nodes must have one that is entirely unmatching, while the other needs to be recursively processed. The processing of the two green child nodes is similar to forward matching, in that only one of them needs to be recursively processed. On the branch green node, both child nodes, like forward matching, require one to be processed recursively, while the other is either entirely matching or entirely unmatching. If it is unmatching, then direct traversing and marking are performed. Overall, there are exactly $h-1$ branch nodes that are entirely unmatching and require reverse marking, while only 1 node that requires two comparisons to determine unmatching predicate. The green node also consists of one or more child nodes that are entirely unmatching or require a single comparison check.

The pseudocode for backward matching on blue nodes is delineated in Algorithm 2. When processing a blue node, if it is a leaf node, two comparisons are performed to determine whether a predicate is match (lines 1-4). If it is a branch node, bDI-Tree recursively processes a blue child node and a green child node, while marking another blue child node (lines 5-14).

When processing a green node, if it is a leaf node, the unmatching predicates can be determined through a single comparison. For branch nodes, two cases are discussed. Among the child nodes of a low green branch node, one must require recursive processing, while the other either does not need processing or is an entirely unmatching node, requiring traversal and marking. The same processing applies to high green branch nodes.

Algorithm 2: The backward matching on blue nodes in bDI-Tree

Input: The event value e_i on attribute a_i, and the blue node BN_i on a_i
Output: B: not-matching subscription bit set

if BN_i *is a leaf node* **then**
  for *Each pair* $< s.ID, c_j >$ *stored in* BN_i **do**
    if $e_i < c_j.l$ *or* $c_j.h < e_i$ **then**
      $B[s.ID] \leftarrow 1$;
else if $e_i \leq$ *the median in* BN_i **then**
  Perform the blue node backward matching on the left blue child node of BN_i;
  Perform the low-green node backward matching on the low-green child node of BN_i;
  for *Each pair* $< s.ID, c_j >$ *stored in the right blue node of* BN_i **do**
    $B[s.ID] \leftarrow 1$;
else
  Perform the blue node backward matching on the blue child node of BN_i;
  Perform the high-green node backward matching on the high-green child node of BN_i;
  for *Each pair* $< s.ID, c_j >$ *stored in the left blue node of* BN_i **do**
    $B[s.ID] \leftarrow 1$;

TABLE II
PARAMETER SETTINGS IN THE EXPERIMENTS

Para.	Description	Values
n	Number of subscriptions	**1M**, 2M...8M, 9M
k	Subscription size	2, 4, 6, 8, **10**
w	Predicate width	0.1, 0.2, 0.3, **0.4**...0.8, 0.9
m	Event size	**20**, 30, 40...80
α	Parameter for Zipf distribution	**0**, 0.25, 0.5, 0.75, 1

C. Bitset Optimization Method

By converting the arithmetic operation of counting to marking and logical operations through bitsets, an optimized variant of fDI-Tree, called fDI-Tree-o, can be derived. The transformation principle is that when inserting predicates, a bitset B_{null} is set up for each attribute to record the subscriptions that have no predicates defined on the corresponding attribute. During matching, the satisfied predicates identified by the DI-Tree are marked in the bitset B_{null} of the corresponding attribute. Finally, bitwise logical AND operations are performed on the bitset B_{null} of each attribute to obtain the matching subscriptions. This optimization is also applied to bDI-Tree.

V. EXPERIMENTS

A. Experiment Setup

We employ real-world stock market datasets as event data, which are also used to generate subscriptions. The parameters used in the experiments are listed in Table II. The default setting of parameters is highlighted in bold. In each experiment, we match a total of 1,000 events and subsequently calculate the average matching time. All the experiments were performed on a server equipped with 128 1.4GHz CPUs, running Ubuntu 18.04.6 with Linux kernel 5.4.0-136. All code was written in C++ language and compiled by g++ of version 7.5.0.

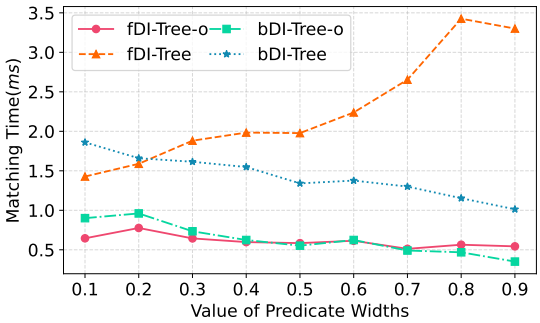

Fig. 2. The Improvement of Bitset Optimization by w

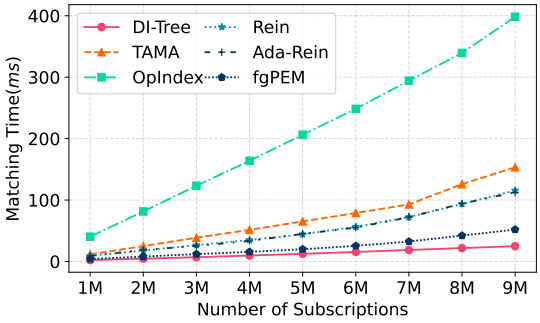

Fig. 3. Effect of the number of subscriptions n

B. Ablation Studies

In this section, we conduct ablation experiments to validate the effectiveness of the bitset optimization method described in Section IV-C. In this content, fDI-Tree denotes the native forward matching algorithm, whereas fDI-Tree-o represents the variant after bitset optimization. Similarly, bDI-Tree represents the native backward matching algorithm. By searching for predicates that are not satisfied on each attribute, we obtain a version with a principle similar to that of the forward matching algorithm. Consequently, we derive bDI-Tree-o, a variant enhanced through bitset optimization.

The comparative analysis of the four tested algorithms is illustrated in Figure 2, as the predicate width changes from 0.1 to 0.9. It can be seen that, compared with fDI-Tree and bDI-Tree, fDI-Tree-o and bDI-Tree-o show a significant improvement in terms of matching time, with 60.8% and 40.57% respectively on average. Furthermore, due to the discrepancies in execution between the forward and backward matching algorithms, fDI-Tree and bDI-Tree exhibit an overall upward and downward trend respectively, with larger fluctuations as predicate width increases. In contrast, fDI-Tree-o and bDI-Tree-o are more stable in their performance.

C. Metric Experiments

The performance of event matching algorithms is influenced by many parameters. We examine the impact of different parameters on the matching time of six algorithms: DI-Tree, REIN [11], Ada-REIN [13], TAMA [8], OpIndex [9], and fgPEM [17] using the single variable method. In this context, DI-Tree refers to the most effective variant, namely fDI-Tree-o, which is the version optimized with bitset optimization.

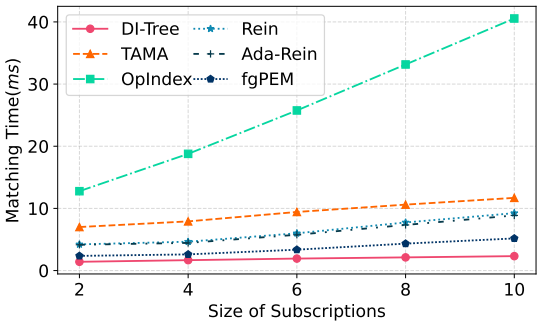

Fig. 4. Effect of subscription size k

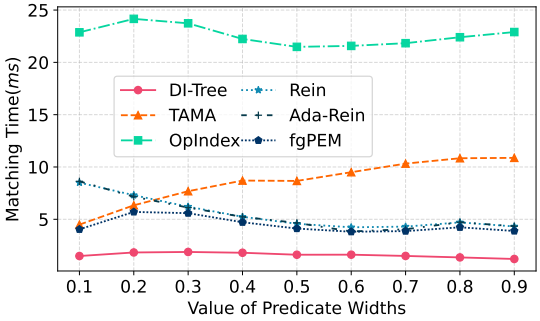

Fig. 5. Effect of predicate width w

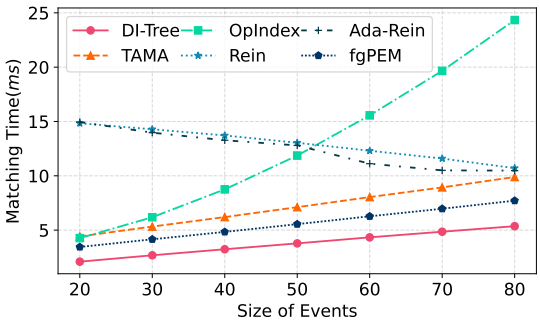

Fig. 6. Effect of event size m

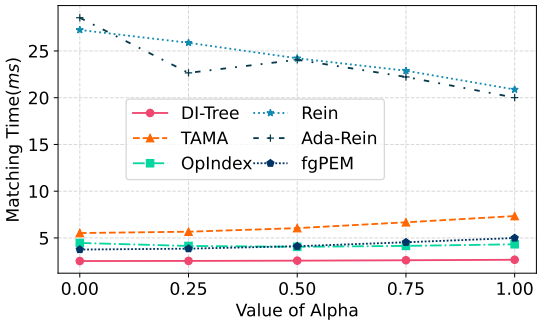

Fig. 7. Effect of attribute Zipf distribution α

1) Effect of Subscription Number n

The number of subscriptions is a crucial metric that significantly impacts matching performance. As shown in Figure 3, we can observe that the matching times for the two forward matching algorithms, OpIndex and TAMA, increase faster than REIN and Ada-Rein as the number of subscriptions grows. DI-Tree stands out as the top performer, with an average performance improvement of 83.7%, 93.7%, 78.4%, 77.8% and 52.0% compared to TAMA, OpIndex, Rein, Ada-Rein and fgPEM respectively. This is because DI-Tree utilizes node bitset optimization, which employs bit operations to handle different nodes on the same attribute, thus avoiding the need to individually mark matching or unmatching predicates. This approach leads to better performance.

2) Effect of Subscription Size k

To assess the impact of subscription size on matching time while ensuring sufficient matching results, we conducted an experiment with n = 1M, m = 30, and w = 0.8. The value of k was gradually increased from 2 to 10. Figure 3 presents the experiment results. All six algorithms exhibit linear growth with respect to k, but at a slower rate compared to the experiment on subscription number. This is due to the fact that in the subscription number experiment, increasing n by 1M results in a 10M increase in total predicates, while in this experiment, each increment of k by 2 only leads to a 2M increase in predicates. Additionally, on average, DI-Tree reduces matching time by 79.6%, 92.0%, 69.1%, 67.9% and 44.9% compared to TAMA, OpIndex, Rein, Ada-Rein and fgPEM respectively.

3) Effect of Predicate Width w

As shown in Figure 5, TAMA and REIN reveal opposite impacts of predicate width on them. In comparison, DI-Tree outperforms REIN's backward marking for most predicate intervals. On average, the performance of DI-Tree is improved by 79.8%, 92.9%, 69.7%, 69.0% and 48.7% compared to TAMA, OpIndex, Rein, Ada-Rein and fgPEM respectively. Moreover, the variance of matching time for REIN, TAMA, DI-Tree is 1.49, 1.01, and 0.05 respectively, indicating that DI-Tree is the most stable algorithm among the three.

4) Effect of Event Size m

In this experiment, the event size m is varied from 20 to 80. The results are shown in Figure 6. As m increases, the matching load also increases for forward matching algorithms such as TAMA and OpIndex. On the other hand, backward matching algorithms like REIN and Ada-REIN show an inverse relationship with m, as a larger m implies fewer empty attributes in events, and therefore reduces the number of unmatching predicates to be marked. Overall, DI-Tree's matching time is 47.7%, 66.2%, 69.6%, 68.0%, and 23.8% less than TAMA, Rein, OpIndex, Ada-Rein and fgPEM respectively.

5) Effect of Attribute Distribution α

The skewness of attribute distributions is configured by a parameter α, with a larger α resulting in more concentration on a few popular attributes. The results of this experiment are shown in Figure 7. DI-Tree effectively addresses the issue of skewness and demonstrates the most stable performance. On the other hand, REIN and Ada-REIN exhibit significant fluctuations as the value of α changes. In terms of matching time, DI-Tree achieves an average improvement of 58.2%, 38.7%, 89.2%, 88.8% and 38.6% compared to TAMA, OpIndex, Rein, Ada-Rein and fgPEM respectively.

D. Maintainability

Maintainability primarily refers to the performance of the matching algorithm in terms of insertion time, deletion time,

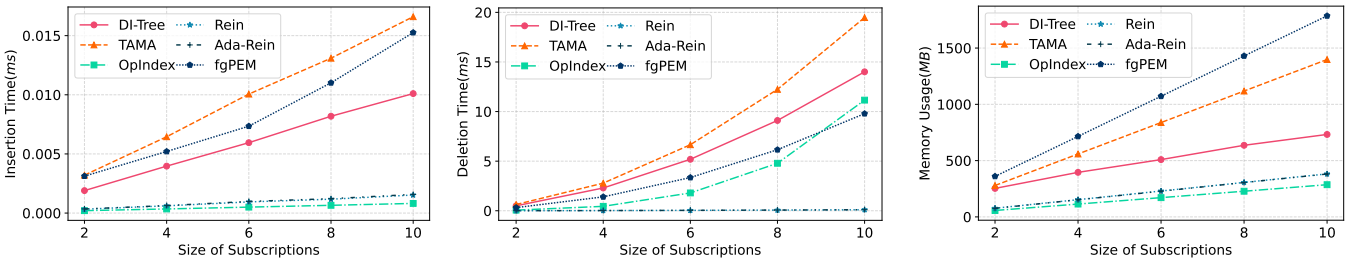

Fig. 8. Maintainability of DI-Tree with different subscription sizes k

and memory usage. The maintainability of DI-Tree is mainly dependent on the subscription size k. Therefore, the experiments focus on k.

The average insertion time of DI-Tree is moderate among the six algorithms, as shown in Figure 8a. This can be attributed to its tree structure. Specifically, OpIndex has the lowest insertion time, followed closely by REIN. On the other hand, TAMA has the highest insertion time, as it needs to insert subscription IDs into multiple buckets for each attribute with defined predicates in a subscription. The insertion time of fgPEM is similar to that of TAMA.

Figure 8b shows that the deletion time of DI-Tree is comparable to that of TAMA. This is due to the fact that the process of searching and deleting the target (predicate value, subscription ID) in buckets is time-consuming, while maintaining a bitset incurs minimal costs. As a result, the deletion times for DI-Tree, REIN, Ada-REIN, TAMA, OpIndex and fg-PEM are 1k, 49, 50, 850, 7k and 500 times their respective insertion times.

The DI-Tree has a moderate memory consumption compared to the other five algorithms, as shown in Figure 8c. As a space-time tradeoff algorithm, the memory consumption of DI-Tree is acceptable, which is approximately 66.3% of TAMA and 51.72% of fgPEM. However, it achieves a performance improvement of 69.4% and 39.8% on average compared to TAMA and fgPEM respectively. This demonstrates the excellent overall performance of DI-Tree.

VI. Conclusion

Building upon existing event matching algorithms, this study aims to further enhance performance and stability to meet real-world demands. We propose DI-Tree, a data structure based on predicate distribution splitting. Furthermore, by counting matching predicates and marking unmatching predicates, we design forward matching algorithm and backward matching algorithms based on DI-Tree. Additionally, optimized versions are proposed from the perspective of bitset optimization and node balance to further enhance performance and reduce memory usage. The experiment results demonstrate the superior performance of DI-Tree. In future endeavors, we intend to propose optimization on DI-Tree from multiple aspects, such as attribute filtering and multi-fanout, to further enhance its overall performance.

References

[1] A. Adadi, "A survey on data-efficient algorithms in big data era," *Journal of Big Data*, vol. 8, no. 1, p. 24, 2021.

[2] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011.

[3] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 421–434, 2015.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM CSUR*, vol. 35, no. 2, pp. 114–131, 2003.

[5] S. C. L. Hernandes, M. E. Pellenz, A. Calsavara, and M. C. Penna, "An efficient event-based protocol for emergency situations in smart cities," in *Proceedings of the 33rd International Conference on Advanced Information Networking and Applications (AINA)*, 2019, pp. 523–534.

[6] T. Ding, S. Qian, J. Cao, G. Xue, and M. Li, "SCSL: optimizing matching algorithms to improve real-time for content-based pub/sub systems," in *IEEE IPDPS*, 2020, pp. 148–157.

[7] S. Slimani and K. Zhang, "Selective auctioning using publish/subscribe for real-time bidding," in *Proceedings of the 16th International Conference on Web Information Systems and Technologies (WEBIST)*, 2020, pp. 26–37.

[8] Y. Zhao and J. Wu, "Towards approximate event processing in a large-scale content-based network," in *ICDCS*, 2011, pp. 790–799.

[9] D. Zhang, C.-Y. Chan, and K.-L. Tan, "An efficient publish/subscribe index for e-commerce databases," *VLDB*, vol. 7, no. 8, pp. 613–624, 2014.

[10] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni, "Indexing boolean expressions," *VLDB*, vol. 2, no. 1, pp. 37–48, 2009.

[11] S. Qian, J. Cao, Y. Zhu, and M. Li, "Rein: A fast event matching approach for content-based publish/subscribe systems," in *IEEE INFOCOM*, 2014, pp. 2058–2066.

[12] W. Fan, Y. Liu, and B. Tang, "Gem: An analytic geometrical approach to fast event matching for multi-dimensional content-based publish/subscribe services," in *IEEE INFOCOM*, 2016, pp. 1–9.

[13] S. Qian, W. Mao, J. Cao, F. L. Mouël, and M. Li, "Adjusting matching algorithm to adapt to workload fluctuations in content-based publish/subscribe systems," in *IEEE INFOCOM*, 2019, pp. 1936–1944.

[14] W. Shi and S. Qian, "HEM: A hardware-aware event matching algorithm for content-based pub/sub systems," in *DASFAA*, 2022, pp. 277–292.

[15] Y. Dong, S. Qian, W. Shi, J. Cao, and G. Xue, "Eem: An elastic event matching framework for content-based publish/subscribe systems," *Computer Networks*, vol. 232, p. 109837, 2023.

[16] W. Zhu, Y. Deng, S. Qian, J. Cao, and G. Xue, "PEM: A parallel ensemble matching framework for content-based publish/subscribe systems," in *The 34th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2022, pp. 106–111.

[17] J. Li, Y. Deng, S. Qian, J. Cao, and G. Xue, "Parallel ensemble matching based on subscription partitioning for content-based publish/subscribe systems," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 32, no. 11&12, pp. 1733–1752, 2022.

[18] T. Ding, S. Qian, W. Zhu, J. Cao, G. Xue, Y. Zhu, and W. Li, "Comat: An effective composite matching framework for content-based pub/sub systems," *ISPA*, 2020.

[19] H. Edelsbrunner, "A new approach to rectangle intersections part i," *International Journal of Computer Mathematics*, vol. 13, no. 3-4, pp. 209–219, 1983.