A fast and anti-matchability matching algorithm for content-based publish/subscribe systems

Shiyu Qian a,b, Jian Cao a,b, Weichao Mao a,b, Yanmin Zhu b, Jiadi Yu b, Minglu Li b, Jie Wang c

a Shanghai Institute for Advanced Communication and Data Science, Shanghai Jiao Tong University, China
b Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
c Department of Civil and Environmental Engineering, Stanford University, USA

A R T I C L E   I N F O

Article history:
Received 3 May 2018
Revised 23 October 2018
Accepted 3 December 2018
Available online 4 December 2018

Keywords:
Matching algorithm
Matchability
Performance
Publish/Subscribe system.

A B S T R A C T

The content-based publish/subscribe system is a flexible many-to-many communication middleware that meets the demands of many large-scale distributed applications. It is well known that event matching is a fundamental component of the content-based publish/subscribe system. When designing matching algorithms, matching speed is a major objective being pursued. Moreover, through theoretical analysis and experimental verification, we discover that the matching speed of most existing matching algorithms is affected by the subscriptions’ matchability which is defined as the matching probability of subscriptions with events. Nevertheless, this problem has not been considered in existing matching algorithms. To address this problem, we propose REIN (REctangle INtersection), a fast and anti-matchability matching algorithm for content-based publish/subscribe systems. REIN is a fast matching algorithm, following the conventional design objective of pursuing a high matching speed. Furthermore, due to the utilization of a negative searching strategy that aims to filter out unmatched subscriptions in the matching process, the matching speed of REIN is not affected by the subscriptions’ matchability, but rather is improved. To evaluate the performance of REIN, comprehensive experiments are conducted. The experiment results show that REIN not only has an excellent matching performance, but also possesses a beneficial anti-matchability feature.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

The content-based publish/subscribe system is a flexible many-to-many communication middleware that meets the demands of many large-scale distributed applications, such as information filtering, selective content dissemination, location-based services, and workload monitoring and management. This communication middleware is attractive in that it achieves full decoupling of the communication parties in time, space and synchronization [1]. In order to distribute workload and be scalable, content-based publish/subscribe systems often use a network of brokers to route subscriptions and forward events. Each broker needs to check the high volume of events against a large number of subscriptions, namely performing event matching, to properly forward events to interested subscribers. It is well known that event matching is a fundamental component of any content-based publish/subscribe system.

Since event matching has an effect on the overall performance of large-scale content-based publish/subscribe systems, different techniques have been proposed to improve the matching speed over the last two decades. For example, efficient data structures have been designed to index subscriptions for the purpose of speeding up event matching [2–7]. The representative data structures include matching tree [8], matching table [3,6], binary decision diagram [9,10], BE-Tree [11], OpIndex [12] and bloom filter [13]. In addition, it is clear that matching speed can be promoted by reducing the number of subscriptions through the covering, subsumption, merging and summarization of subscriptions [10,14–17]. Therefore, it is natural that matching speed is one of the major objectives being pursued when designing matching algorithms. Usually, matching time is used to measure matching speed, which is defined as the running time spent on matching an event against a set of subscriptions.

Generally, there are two searching strategies that can be adopted by matching algorithms to find matching subscriptions from candidates: positive searching and negative searching. Positive searching directly locates matching subscriptions, ignoring unmatched ones in the process of event matching. On the contrary,
negative searching first identifies unmatching subscriptions. With
the availability of all subscriptions and unmatching ones, it is easy
to obtain matching subscriptions indirectly.

Existing matching algorithms usually utilize a positive searching
strategy to pursue a high matching speed, such as SIENA [3], TAM
A [6] and OplIndex [12]. However, one side effect of positive searching
is that the performance of the matching algorithms is affected by
the subscriptions’ matchability which is defined as the matching
probability of subscriptions with events. Through theoretical anal-
ysis and experimental verification, we discover that the matcha-
ibility of subscriptions has a great effect on the matching speed
for matching algorithms that follow a positive searching strategy.
Specifically, given a set of subscriptions, the matching time in-
creases linearly or logarithmically with the number of matching
subscriptions, giving rise to performance variation. To the best of
our knowledge, this problem has not yet been addressed in the lit-
erature.

The performance variation of matching algorithms has two
drawbacks. First, it is difficult to estimate the throughput of match-
ing algorithms with unsteady performance. To deal with workload
spikes during rush hour, optimal resource reservation and schedul-
ing is usually based on precise estimates. Second, performance
variation may cause violent fluctuation in event transfer latency for
subscribers. Theoretically, it would be ideal to find all the matching
subscriptions at the same moment, thus ensuring that there is no
obvious difference in event transfer latency for subscribers. How-
ever, even though subscriptions are partitioned into groups and
parallel matching algorithms are employed, a time gap remains be-
tween the identifying time of matching subscriptions due to the
large number of subscriptions. For matching algorithms that follow
a positive searching strategy, when there are more matching sub-
scriptions, a longer matching time is needed. Obviously, the longer
the matching time, the larger the time gap between the identifying
time of matching subscriptions, as well as the higher the fluctua-
tion of event transfer latency. Thus, while simultaneously pursuing
a high matching speed, one should consider the effect of the sub-
scriptions’ matchability when designing matching algorithms.

In relation to the searching strategy being applied to matching
algorithms, there are two options: positive searching and negative
searching. Negative searching and positive searching differ in the
way they retrieve matching subscriptions. The basic idea of nega-
tive searching is that, given a set of subscriptions, if all unmatch-
ingsubscriptions are known, it is easy to determine the matching
ones. Negative searching is more efficient than positive searching
in terms of predicate evaluations. For example, when matching an
event against a subscription that contains five predicates, all five
predicates should be evaluated to establish the matching relation.
On the contrary, the subscription can be determined as unmatch-
ing as soon as one of the five predicates is found to be false. The
count of predicate evaluations is minimally one and at most five.
Furthermore, another advantage of negative searching is that the
matching time decreases with the number of matching subscrip-
tions. In other words, the matchability of subscriptions improves
the matching speed, instead of degrading it.

In this paper, we present REIN (RECTangle INtersection), a fast
and anti-matchability matching algorithm for content-based pub-
lish/subscribe systems. The key idea behind REIN is to employ a
negative searching strategy. For the data model of subscriptions, it
is assumed that each subscription is composed of multiple interval
predicates. An interval predicate is a condition specified on an at-
tribute with a low value and a high value. The attributes appearing
in events form a high-dimensional space. In this space, a subscrip-
tion is a high-dimensional rectangle (for short rectangle), and an
event is a point. Therefore, the event matching problem is equiva-

dent to the point enclosure problem. As proved in [18], the rectan-
gle intersection problem is equivalent to the point enclosure prob-
lem. We utilize the rapid detection method of disjoint rectangles
to quickly find unmatching subscriptions. In addition, an efficient
index structure is designed to address the rectangle intersection
problem by using bit operations.

Compared with matching algorithms that follow a positive
searching strategy, REIN has five attractive features. First, with
the help of a specialized data structure, REIN is highly efficient in
searching unmatching subscriptions, having a high matching speed.
Second, the matching time of REIN decreases with the number of
matching subscriptions, exhibiting a nice anti-matchability feature.
Third, the matching performance of REIN is fairly stable, showing
low standard deviation of matching times. Fourth, REIN is not af-
fected by the distributions of predicate values, maintaining robust-

ness. Fifth, REIN is relatively efficient in updating (insert or delete)
subscriptions in its underlying data structure, making it applicable
to very dynamic environments.

Extensive experiments are conducted to evaluate the perfor-
mance of REIN. First, the parameters that impact the performance
of matching algorithms are identified, including the matchability
of subscriptions, the number of interval predicates, the number
of subscriptions, and the distribution of predicate values. In the
experiments, the number of subscriptions is up to 2 million and the
number of predicates contained in the subscriptions is up to 23.
The experiment results show that REIN strongly outperforms five
reference matching algorithms in terms of matching speed and
performance stability. As discussed in this paper, our main contri-

butions are:

• We discover the effect of the subscriptions’ matchability on the
  performance of matching algorithms and run a series of ex-
  periments to verify this.

• We propose a negative searching strategy to alleviate the effect
  of the subscriptions’ matchability and design an efficient index
  structure to implement this strategy.

• We conduct extensive experiments to evaluate the performance
  of REIN and thoroughly analyze the experimental results.

The remainder of this paper is organized as follows. Section 2
provides background knowledge on publish/subscribe systems. Sec-
tion 3 reviews the related work. Section 4 describes the effect of the
subscriptions’ matchability. Section 5 details the design of REIN. Sec-
tion 6 presents the experiment results. Section 7 discusses two issues relating to REIN. Finally, Section 8 concludes the paper. This paper is an extended version of [19].

2. Background

Some preliminary knowledge on publish/subscribe systems is
given in this section. We first present the data model by defin-
ing the terms, and then describe the architecture and rationale of
publish/subscribe systems.

2.1. Terms

Definition 1. Event

An event is an observable occurrence, also called a message,
publish or notification. An event usually contains multiple at-
tributes, expressed as a conjunction of attribute-value pairs. As a
convention, each attribute appears only once in an event expres-

sion.

For example, \((\text{tem} = 35), \text{(hum} = 15)\) is an event describing
weather conditions. The set of attributes appearing in the event
expression is defined as \(A = \{a_1, a_2, \ldots, a_m\}\) and the number of at-
tributes in \(A\) is denoted by \(m\). The sources of events are called pub-
lishers.
Definition 2. Predicate
A predicate is a condition specified on an attribute selected from \( A \). We consider interval predicates in inclusive form in this paper, which are represented as a 3-tuple of \( \{a, v_1, v_2\} \), where \( a \) is an attribute in \( A \), \( v_1 \) and \( v_2 \) are bounded by the value domain of \( a \), and \( v_1 \) is not larger than \( v_2 \). \( v_1 \) and \( v_2 \) are termed as predicate values in this paper.

Given the value domain of attributes, other forms of predicates can be transformed into interval predicates. For example, if the data type of \( tem \) is an integer with a value domain \([0, 100]\), then the simple predicate \( \{\text{tem} \geq 20\} \) can be transformed into the interval predicate \( \{\text{tem}, 20, 100\} \).

Definition 3. Subscription
A subscription expresses users’ interest in events and is specified as a conjunction of multiple interval predicates.

Users who issue subscriptions are called subscribers. Subscriptions are used to forward events from publishers to subscribers. Each subscription is identified by a unique \( subID \). The number of interval predicates contained in a subscription is not larger than \( m \), where \( m \) is the number of attributes appearing in events. A subscription matches an event if all the interval predicates contained in the subscription are satisfied when they are assigned the corresponding attribute values of the event.

Definition 4. Matchability
The matchability of a subscription is the average probability that the subscription matches events. The matchability of a predicate is the probability that the predicate is satisfied by assigning the corresponding attribute value of events. The matchability of a subscription is determined by the matchabilities of predicates contained in the subscription.

Definition 5. Event Matching
Given a set of \( n \) subscriptions \( S = \{s_1, s_2, \ldots, s_n\} \) and an event \( e \), the problem of event matching is to retrieve all subscriptions that match \( e \) from \( S \). The set of the matching subscriptions \( S_e \) is a subset of \( S \), \( S_e \subseteq S \).

\[
S_e = \{s_i \mid s_i \in S \cap s_i \text{ matches } e\}
\]  

(1)

2.2. Publish/Subscribe system

A typical publish/subscribe system consists of subscribers, publishers, and a network of brokers. Publishers inject events into the system while subscribers issue subscriptions for interested events. The core of the publish/subscribe system is to disseminate events from publishers to subscribers as quickly as possible. Usually, subscriptions are broadcasted to all brokers [20,21] or directed to rendezvous brokers [22,23]. In the former case, event matching is performed by each broker on the reverse path that is constructed during the broadcasting process of subscriptions, and events are delivered to subscribers step by step. In the latter case, event matching is carried out only at the rendezvous brokers, and events are directly sent to interested subscribers by those rendezvous brokers. An example of a publish/subscribe system is shown in Fig. 1. Here, the system has 8 subscribers, 2 publishers, and a network of 7 brokers.

Whenever a broker receives an event from one of its neighbors (either another broker or a publisher), event matching is carried out to decide whether the event should be forwarded to the next-hop brokers or subscribers. For large-scale distributed publish/subscribe systems, there may be millions of subscriptions maintained by brokers. Brokers with low matching speed are apt to be potential performance bottlenecks. For example, when the event arrival rate is larger than the matching rate of a broker, the broker becomes a performance bottleneck. Therefore, improving matching speed is critical for large-scale contented-based publish/subscribe systems.

3. Related work

Designing highly efficient matching algorithms has drawn a great amount of attention in the last two decades, and many different techniques have been proposed. These techniques can be classified roughly into three categories: (i) proposing new index structures, (ii) reducing the number of subscriptions, and (iii) utilizing the parallel computing capability of hardware.

3.1. Proposing new index structures

New index structures that have been proposed to improve matching efficiency include the work presented in [2,3,5,6,12,13]. Most of these are counting-based matching algorithms that take a positive searching strategy. The matching procedure of these algorithms generally consists of three steps. Step 1, all satisfied predicates are quickly evaluated through the index structures. Step 2, counting algorithms are used to sum up the number of satisfied predicates for each subscription. Step 3, the number of satisfied predicates is compared with the number of predicates contained in the subscription to judge whether the subscription is matching. SIENA [3] and TAMA [6] are two representative counting-based matching algorithms. The index structure of SIENA is a two-level forwarding table, which is applicable to the situation where subscriptions change infrequently, as every modification of subscriptions leads to rebuilding the whole table. The index structure of TAMA is a two-layer matching table used for approximate event matching. H-TREE is not a counting-based algorithm which puts similar subscriptions together and skips subtrees to filter unmatched subscriptions in the process of event matching [7]. When matching events, a lot of unmatched subscriptions are filtered out to improve the matching speed. H-TREE is highly efficient in cases where the width of interval predicates is smaller than the width of cells divided on the attributes’ value domain.

One problem with counting-based matching algorithms is low efficiency. Although an unmatched subscription can ultimately be determined in the matching process, it is counted multiple times which equals the number of satisfied predicates. Therefore, the time complexity of counting-based matching algorithms is linear or logarithmic in the number of satisfied predicates.
3.2. Reducing the number of subscriptions

In addition to proposing new index structures, reducing the number of subscriptions is another way to improve matching speed. The covering and subsumption relations among subscriptions are utilized to reduce the number of subscriptions [4,10,14–17,24–29]. If subscription $s_1$ covers subscription $s_2$, if and only if for any event $e$ that is a match of $s_2$, $e$ must be a match of $s_1$. Given a set of existing subscriptions $S = \{s_1, s_2, \ldots, s_n\}$ and a new subscription $s$, the result of subsumption checking for $s$ is true if and only if $s \subseteq u_{i=1}^n s_i$.

All of the relations of subscriptions, subsumption is the most efficient to reduce the number of subscriptions [4,15,17,26–28]. For example, space-filling curves, such as Hilbert, are used to represent the content space to efficiently check subscription subsumption [4,28]. Subscription covering is less efficient than subsumption, but at a lower cost, such as in [24,25]. Based on the covering and subsumption relations of subscriptions, merging and summarization can be used to reduce the routing table size (the number of subscriptions) [10,14,30]. These techniques are complementary to our proposed matching algorithm, and it is beneficial to incorporate them in REIN. However, checking the subsumption relationships among subscriptions is not trivial, and it is not suitable for dynamic environments.

3.3. Utilizing parallel computing capability

Traditionally, most matching algorithms are sequential. Although these algorithms are effective for improving matching speed and increasing processing throughput, they fail to efficiently utilize the hardware’s parallel computing capability. In recent years, some parallel matching algorithms have been proposed to take advantage of the development of hardware, such as CPUs with multi-cores as well as GPUs. For example, the parallel computing capability of today’s multi-processor chips is exploited to improve matching efficiency in [31,32]. New matching algorithms have been designed to run efficiently on GPUs in [33,34]. In addition, matching algorithms running on FPGAs have been proposed to achieve fine-rate processing by exploring various degrees of parallelism in [35]. Overall, parallel matching algorithms have been developed on the foundation of sequential ones. Yet, the cost of communication and synchronization needs to be carefully considered when designing parallel algorithms.

In summary, REIN differs from existing works in two aspects. Firstly, existing algorithms mainly adhere to the design objective of pursuing a high matching speed, without considering the effect of the subscriptions’ matchability on matching performance. Furthermore, the negative searching strategy is seldom employed in existing matching algorithms. Exploring a different direction, REIN utilizes a negative searching strategy to alleviate the impact of the subscriptions’ matchability.

4. Effect of subscriptions matchability

In this section, we theoretically analyze the impact of the subscriptions’ matchability on the efficiency of matching algorithms. Since the matchability of a subscription is determined by the matchabilities of the subscription’s predicates, we theoretically analyze the impact of predicates’ matchability on the efficiency of matching algorithms that apply a positive searching strategy.

According to the relations between subscriptions and events, subscriptions can be placed into two categories: matching and unmatching. Of the unmatching subscriptions, two subcategories can be divided further, namely partially unmatching and completely unmatching. For a given subscription, if all of its predicates are satisfied, it is matching; if none of the predicates are satisfied, it is completely unmatching; otherwise, the subscription is partially unmatching.

For most matching algorithms applying a positive searching strategy, such as SIENA [3], TAMA [6] and OIndex [12], predicates are the basic units indexed in their underlying data structures. In this way, it is efficient to retrieve satisfied predicates when matching events. When predicates contained in a subscription are independently indexed, the relationship among predicates that consist of the subscription needs to be maintained. A common method is to let predicates contained in the same subscription point to a counter. As discussed in Section 3.1, for counting-based matching algorithms, the number of satisfied predicates is counted for both matching and partially unmatching subscriptions. However, it is useless to process partially unmatching subscriptions, as this degrades the matching efficiency. Therefore, it is time-consuming to obtain the set of matching subscriptions through counting algorithms.

One advantage of negative searching over positive searching is that the relationship of predicates contained in a subscription does not need to be maintained in the data structures. For a subscription, as soon as a predicate contained in the subscription is evaluated as false, the subscription is determined as unmatching. In terms of predicate evaluations, negative searching is more efficient than positive searching when the matchability of predicates is relatively high or the number of predicates contained in the subscription is large.

Given a subscription $s$ containing $k$ independent predicates, and the matchability of predicates is $p$, it is assumed that the data structures for both positive searching and negative searching cost the same to evaluate the predicates. For simplicity, we assume that $p$ is the matchability of all predicates. In real use cases, predicates can have different matchability.

Lemma 1. For matching algorithms applying a positive searching strategy, the expectation of predicate evaluations is $pk$.

Proof. The proof of this lemma is very straightforward. For positive searching, the number of satisfied predicates should be counted for each subscription. The count of predicate evaluations equals the number of predicates that are evaluated as true. Given the number of predicates $k$ and the matchability of predicates $p$, $pk$ predicates are evaluated to be true on average. □

Lemma 2. For matching algorithms applying a negative searching strategy, the expectation of predicate evaluations is $\frac{1-p^k}{1-p}$.

Proof. For negative searching, since we can assert that a subscription is unmatched as long as we find one unsatisfied predicate, the expected number of predicates that needs to be checked in a subscription is

$$\sum_{i=1}^{k} (i * p^{i-1} * (1-p)) + k * p^{k-1} * p = \frac{1-p^k}{1-p}.$$  \(2\)

□

Lemma 3. Given the number of predicates $k (k > 3)$ contained in subscriptions, there is a turning point of $p$ that makes negative searching more efficient than positive searching in terms of predicate evaluations.

Proof. When $\frac{1-p^k}{1-p} < pk$, negative searching is more efficient than positive searching in terms of predicate evaluations. When $k \leq 3$, there are no real solutions to the inequality. The Abel—Ruffini Theorem states that there is no algebraic solution when $k > 5$. However, these solutions can be computed to any desired degree of accuracy using numerical methods such as the Newton—Raphson method or the Laguerre method. For example, when $k = 5$, for all $p$
5. Design of REIN

In this section, we detail the design of REIN and analyze its time complexity. An example is also provided to illustrate the data structure and matching procedure of REIN.

5.1. Overview

The design objective of REIN is twofold. First, since matching performance is critical to both matching algorithms and publish/subscribe systems, naturally, the clear aim of REIN is to pursue a high matching speed. Second, the performance of REIN should be stable, exhibiting an anti-matchability feature. As analyzed and verified in Section 4, matching algorithms following a positive searching strategy do not possess an anti-matchability feature. To pursue a high matching speed and to alleviate the effect of the subscriptions’ matchability, REIN uses a negative searching strategy. The basic idea of negative searching is that, given a set of subscriptions, if the unmatching subscriptions can be identified quickly, it will be easy to obtain the matching ones.

After establishing the design objective of REIN, the next step is to investigate this method and gauge its results. As defined in Definition 5, given a set of subscriptions S and an event e, the problem of event matching is to find all subscriptions from S that match e. The set of attributes $A = \{a_1, a_2, \ldots, a_m\}$ in events forms a $m$-dimensional space. In the space, events are points and subscriptions are rectangles. Therefore, the event matching problem is equivalent to the point enclosure problem [36], that is, finding all rectangles that contain a given point. Points can be seen as special rectangles. As proved in [18], the rectangle intersection problem is equivalent to the point enclosure problem. We utilize the rapid detection method of disjoint rectangles to quickly search unmatching subscriptions.

5.2. Index structure

In order to pursue a high matching speed without sacrificing performance stability, REIN uses a negative searching strategy that searches unmatching subscriptions during the process of event matching. Therefore, when matching events against subscriptions, the challenge of designing REIN is to realize a fast searching method that is able to find all unmatching subscriptions. To tackle this challenge, it is necessary to design an index structure.

The index structure of REIN consists of a collection of bucket lists. The number of bucket lists is $2m$, where $m$ is the number of attributes appearing in events. For each attribute, two bucket lists are constructed. One bucket list is for the low values of the interval predicates specified on the attribute and the other is for the high values of the interval predicates. A bucket list is constructed by dividing the value domain of an attribute into cells and realizing the mapping from the cells to the buckets. All values belonging to a cell map to the corresponding bucket. When a predicate is indexed in a bucket, the predicate value and the corresponding subID are inserted into the bucket. An example of the index structure is shown in Fig. 3, where four bucket lists are created for two attributes $a_1$ and $a_2$. The value domain of $a_1$ and $a_2$ is $[0, 20]$ from which four cells are divided. Each cell maps to a bucket in which the predicate values and subIDs are stored.

The number of cells divided on a value domain is determined by multiple factors. One is the stability of subscriptions. For a publish/subscribe system, if the subscriptions are relatively static, fewer cells can be created, and the items in each bucket can be sorted on the predicate values to obtain better matching performance. Otherwise, more cells are needed to reduce the cost of
subscription modifications. Another factor is the number of subscriptions. In order to improve matching efficiency, the size of the buckets, represented by the number of items in the buckets, is critical. Given the number of subscriptions, there is a turning point for the number of buckets. After reaching the turning point, the performance of REIN degrades with the addition of more buckets.

5.3. Event matching

The event matching procedure of REIN is straightforward, as described by Algorithm 1 in Fig. 4. When matching an event, a bit-set is initialized in which the number of bits equals the number of subscriptions (line 3). All unmatching subscriptions are marked in the bitset. Given an event that is specified by \( \{ a_1 = v_1, a_2 = v_2, \ldots, a_m = v_m \} \), for each attribute, the values (low value and high value) of the predicates are compared with the values of the event, finding and marking all subscriptions where (i) the high value of the interval predicates is less than the value of the event (lines 4 – 11), and (ii) the low value of the interval predicates is larger than the value of the event (lines 12 – 18). The unmarked bits in the bitset represent the matching subscriptions (lines 20 – 24).

The matching efficiency of REIN is manifested in three aspects. First, the divide-and-conquer strategy is applied to the index structure, dividing the value domain of attributes into multiple cells that map to buckets. Thus, the size of the buckets is much smaller compared with the number of subscriptions. Second, comparison operations are only executed in two buckets for each attribute when matching an event, rapidly traversing the remainder of the buckets. Third, the index structure of REIN is concise, embedding the principle of simplicity.
5.4. Complexity analysis

We now analyze the time complexity of REIN. To facilitate the analysis, it is assumed that the distribution of predicate values and event values are uniform and the predicate attributes are uniformly selected from the event attributes. For the following lemmas, the number of subscriptions is \( n \), the number of predicates in the subscriptions is \( k \), the matchability of predicates is \( p \), the number of attributes in the events is \( m \), and the number of buckets is \( b \).

**Lemma 4.** The number of predicates in a bucket is \( e = \frac{nk}{mb} \).

**Proof.** The total number of buckets is \( m^b \), since for each one of the \( m \) event attributes, the value domain is divided into \( b \) cells and each cell is mapped to one bucket. Since the \( n^k \) predicates are evenly inserted into the \( m^b \) buckets, the number of predicates in a bucket is \( e = \frac{nk}{mb} \). \( \square \)

**Lemma 5.** The time complexity of event matching is \( O(mb) \).

**Proof.** According to the matching procedure of REIN, the main operations performed by REIN are comparison, traversing and switching. For each attribute, the comparison cost is \( O(e) \) since the comparison operation is performed in 2 buckets. On average, half of the buckets are traversed, causing the predicates traversing cost and bucket switching cost both to be \( O(be) \). Since there are \( m \) attributes in events, the time complexity of event matching is \( O(mb) \). \( \square \)

**Lemma 6.** The time complexity of inserting a subscription is \( O(k) \).

**Proof.** For a subscription containing \( k \) predicates, the subID of the subscription should be inserted into \( 2k \) buckets. For each of the \( k \) predicates, computing the mapped bucket can be done in \( O(1) \). Since sorting is not implemented in the buckets, appending an item in a bucket is \( O(1) \). Therefore, the time complexity of inserting a subscription is \( O(k) \). \( \square \)

**Lemma 7.** The time complexity of deleting a subscription is \( O(ke) \).

**Proof.** When deleting a subscription containing \( k \) predicates, the subID of the subscription should be removed from \( 2k \) buckets. Since each bucket has \( e \) predicates and sorting is not implemented in buckets, locating the subID of the subscription costs \( O(e) \). Therefore, the time complexity of deleting a subscription is \( O(ke) \). \( \square \)

**Theorem 2.** Given the number of subscriptions \( n \), the number of predicates \( k \) contained in the subscriptions, the optimal number of buckets is \( b^* = \sqrt{\frac{nk}{2mp}} \).

**Proof.** Given the number of buckets \( b \) and the number of attributes \( m \) in events, let \( \alpha \) be the unit time to compare a predicate in the bucket, \( \beta \) be the unit time to traverse a predicate in the bucket, and \( \gamma \) be the unit time to switch buckets when traversing, then the total matching time can be denoted as:

\[
T(b) = 2\alpha e + b(\beta e + \gamma) = 2\alpha \frac{nk}{mb} + \frac{\beta nk}{m} + by.
\]

Taking the derivative of the total cost \( T(b) \) with respect to \( b \), setting it to 0 and solving the equation, we get \( b^* = \sqrt{\frac{2nk}{mp}} \). \( \square \)

5.5. Example

To illustrate the index structure and the matching procedure of REIN, we present an example in Fig. 3. There are two attributes in the index structure. For each attribute, two bucket lists are created, one for the low values of the interval predicates and the other for the high values. The value domain of each attribute is \([0, 20]\). 10 subscriptions listed in Table 1 are stored in the index structure, which is represented in Fig. 3(a). When indexing a subscription, the interval predicates contained in the subscription are processed one by one. For each interval predicate, the low value and the high value are used to determine the corresponding bucket to store the subID of the subscription. Please note that if there are \( k \) interval predicates in a subscription, the subID of the subscription is stored \( 2k \) times. For example, when indexing \( s_1 \), the low value on \( a_1 \) is 4 which is located in the cell mapping to bucket \( b_0 \) as shown in Fig. 3(b). The high value specified on \( a_1 \) is 12, which is in the cell mapping to bucket \( b_2 \) as shown in Fig. 3(c). The same is true for processing the interval predicate specified on \( a_2 \), indexing \( s_1 \) in \( b_1 \) and \( b_2 \) as shown in Fig. 3(d) and Fig. 3(e), respectively.

The rectangles that represent the 10 subscriptions are shown in Fig. 5(a). Given an event \( e \) \( \{a_1 = 6, a_2 = 10\} \) (denoted as a red point), the matching subscriptions of the event \( e \) are those rectangles that intersect with the point. When the rectangles that disjoin with the point are determined, it is easy to obtain those that intersect with the point. The rectangles that have their right sides on the left side of the point are marked, namely \( s_0 \). The resulting rectangles for the right side of the point are \( s_4, s_5, s_6, s_7, s_8 \) and \( s_9 \). For the top side of the point, \( s_0, s_2 \) and \( s_3 \) are the resulting rectangles. Additionally, \( s_4, s_8 \) and \( s_9 \) are the rectangles for the bottom side. Obviously, some rectangles are marked multiple times. By checking the bitset, we see \( s_1 \) and \( s_2 \) are the matching subscriptions for the event \( e \), as shown in Fig. 5(b).
system of the server is Ubuntu 11.10 with Linux kernel 3.0.0–12. Parallelism is not used in the experiments. In each experiment, 400 events are matched.

6.1.4. Metric

To comprehensively evaluate the performance of the six tested algorithms, three time metrics are measured: matching time, construction time and deletion time. Matching time is the most important metric used to evaluate the matching speed of matching algorithms. Construction time and deletion time represent the maintenance cost of matching algorithms. In addition, the memory consumption of all tested matching algorithms is compared.

6.2. Effect of subscriptions’ matchability

We first conduct an experiment to confirm that the effect of the subscriptions’ matchability on the performance of matching algorithms does exist, verifying the analysis in Section 4. In the experiment, the parameters are set as follows: $n = 2,000,000$, $m = 20$, $k = 10$ and $w = 0.5$. The average matchability of subscriptions is almost 0.001 ($= 0.5^{10}$). For each of 400 events, the number of matching subscriptions and the corresponding matching time are recorded. The correlation between the matching time and the number of matching subscriptions of the six algorithms are depicted in Fig. 6. By regression analysis, the goodness of these fits is all above 0.7 in terms of $R^2$ with 95% confidence.

Overall, we find that the matching time of the five compared matching algorithms increases either logarithmically or linearly with the number of matching subscriptions. Specifically, the matching time of SIENA, TAMA and OplIndex increases logarithmically with the number of matching subscriptions as shown in Fig. 6(a), (b) and (c), respectively. H-TREE and BE-TREE exhibit linear correlation, which is depicted in Fig. 6(d) and (e) respectively. On the contrary, the matching time of REIN decreases logarithmically with the number of matching subscriptions, which is verified in Fig. 6(f).

SIENA, TAMA and OplIndex are based on counting algorithms to positively search matching subscriptions. Counting-based matching algorithms have an obvious performance drawback that is an unmaching subscription may be counted multiple times because parts of its predicates are satisfied. When the number of predicates in subscriptions is large, much time is wasted checking the vast number of partially unmatching subscriptions that contain one or more satisfied predicates. Therefore, the performance of counting-based matching algorithms degrades dramatically when there are millions of subscriptions and each subscription contains tens or even hundreds of predicates, as shown in Fig. 6(a), (b) and (c).

H-TREE and BE-TREE are tree-based matching algorithms. The basic idea of tree-based matching algorithms is that matching speed can be improved when the search space is substantially reduced by pruning most of the unmatching subscriptions. To this end, subtrees are skipped according to the coverage relationship between subscriptions, just checking the subtrees that contain subscriptions with a high matching probability. After identifying these subtrees, a naive matching method is used to determine the matching subscriptions in H-TREE and BE-TREE, which causes the matching time to increase linearly with the matchability of subscriptions, as depicted in Fig. 6(d) and (e).

REIN exhibits an excellent anti-matchability feature in that the matching time of REIN decreases logarithmically with the number of matching subscriptions, as shown in Fig. 6(f). This can be explained in that when more subscriptions match an event, fewer bits in the bitset are marked in the course of matching, thus reducing the matching time.

Table 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>the number of subscriptions</td>
</tr>
<tr>
<td>$m$</td>
<td>the number of attributes contained in events</td>
</tr>
<tr>
<td>$k$</td>
<td>the number of predicates contained in subscriptions</td>
</tr>
<tr>
<td>$w$</td>
<td>the matchability of interval predicates</td>
</tr>
<tr>
<td>$d$</td>
<td>the discretization level in TAMA</td>
</tr>
<tr>
<td>$c$</td>
<td>the number of cells in H-TREE</td>
</tr>
<tr>
<td>$l$</td>
<td>the number of indexed attributes in H-TREE</td>
</tr>
<tr>
<td>$s$</td>
<td>the number of segments in OplIndex</td>
</tr>
<tr>
<td>$g$</td>
<td>the bits of signature in OplIndex</td>
</tr>
<tr>
<td>$b$</td>
<td>the number of buckets in REIN</td>
</tr>
</tbody>
</table>

6.3. Matching time

The matching time of REIN is affected by multiple parameters, including the number of subscriptions, the number of predicates contained in subscriptions, the matchability of predicates, the distribution of predicate values, and the number of buckets. We thoroughly evaluate the effects of these parameters in this section.

6.3.1. Number of subscriptions

In this experiment, we evaluate the effect of the number of subscriptions. The parameters set in the experiment are as follows: \( m = 20 \), \( k = 10 \) and \( w = 0.5 \). The experiment results shown in Fig. 7 reveal two more attractive features of REIN, namely rapidity and stability.

Overall, the matching time increases with the number of subscriptions for all the tested algorithms, including REIN. In the experiment, the matchability of subscriptions is deterministic. With more subscriptions, the number of matching subscriptions for the events increases accordingly, resulting in a longer matching time. Compared with the other five matching algorithms, the performance of REIN is least affected by the number of subscriptions. When the number of subscriptions is 2,000,000, REIN is almost 2.3, 2.5, 2.9, 8.8 and 4.5 times faster than SIENA, TAMA, H-TREE, BE-TREE and OpIndex, respectively.

Besides OpIndex, REIN performs more stably than other four matching algorithms. The minimum (Min), maximum (Max) and standard deviation (Std) of the matching time are listed in Table 3, where the number of subscriptions is 2,000,000. By observing the standard deviation, the matching performance of OpIndex is the most stable, a little better than REIN. However, the standard deviation of REIN is almost 1.1, 1.9, 17.0 and 85.1 times smaller than the standard deviations of SIENA, TAMA, H-TREE and BE-TREE, respectively. In addition, the stability of REIN is also manifested in the difference between the maximum and the minimum of matching time. The difference of REIN is 26.74 ms, which is on the same scale of OpIndex which has the smallest difference of 24.31 ms. However, the difference of SIENA, TAMA, H-TREE and BE-TREE is 27.77, 53.65, 614.45 and 1660.48 ms, respectively.

6.3.2. Number of predicates

An experiment is conducted to evaluate the effect of the number of predicates. In the experiment, the number of attributes \( m \) is variable, and the number of predicates \( k \) is half of \( m \). The results of the experiment are shown in Fig. 8, where \( n = 1,000,000 \), \( w = 0.5 \), and the y-axis represents the matching time in log scale. As previously mentioned, the matching time of H-TREE is linear to the number of matching subscriptions. When the number of predicates increases, the matchability of subscriptions decreases. Therefore, the matching time of H-TREE first drops quickly with the number of predicates. Then, the matching time of H-TREE is mainly affected by the number of predicates, which is similar to the other five algorithms. In general, the matching time of the six algorithms increases with the number of predicates. When the number of predicates is 23, REIN is 2.7, 1.9, 2.2, 10.2 and 5.2 times faster than SIENA, TAMA, H-TREE, BE-TREE and OpIndex, respectively.

| Table 3 The maximum, minimum, and standard deviation of matching time (ms). |
|------------------|----------------|------------------|------------------|-----------------|------------------|
|                  | SIENA           | TAMA             | H-TREE           | BE-TREE          | OpIndex          |
| Min              | 87.930          | 87.109           | 17.348           | 70.286           | 190.137          |
| Max              | 115.697         | 140.756          | 631.795          | 1730.762         | 214.444          |
| Std              | 5.539           | 10.122           | 88.352           | 442.972          | 4.736            |

Fig. 7. Effect of number of subscriptions.

Fig. 6. Relationship between matching time and the number of matching subscriptions.
faster than SIENA, TAMA, H-TREE, BE-TREE and OpIndex, respectively.

6.3.3. Matchability of predicates

Given the distribution of event values, the width of the inter-
val predicates determines the matchability of subscriptions. In gen-
eral, the wider the width, the larger the matchability. An experi-
ment is conducted to evaluate the effect of interval width (predic-
tes’ matchability) on the matching time. The results are shown in
Fig. 9, where \( n = 1,000,000, m = 20, k = 10 \), and the y-axis repre-
sents the matching time in log scale. The matching time of SIENA
increases with the matchability of interval predicates. With the in-
crease of \( w \), TAMA behaves asymptotically, as does SIENA. Of the six tested algorithms, H-TREE behaves best when \( w \leq 0.3 \). When
\( w > \frac{1}{2}, \) a subscription is split into \( \left\lceil \frac{k}{w} \right\rceil \) subscriptions with nar-
row interval predicates, where \( c \) is the number of cells and \( l \) is
the number of indexed attributes in H-TREE. When \( w \geq 0.7, \) 32GB
memory is used up due to the exponential growth. Therefore, there
are no results shown in the figure when \( w = 0.7 \) and \( w = 0.8 \) for H-
TREE. The matching time of REIN decreases with \( w \), exhibiting an
anti-matchability feature. When \( w = 0.6, \) REIN is 3.0, 2.3, 26.3,
50.9 and 5.4 times faster than SIENA, TAMA, H-TREE, BE-TREE and
OpIndex, respectively.

6.3.4. Distribution of predicate values

Ideally, the subIDs of subscriptions should be stored evenly in
the buckets of REIN, but this is impractical in reality. REIN is nearly
unaffected by the distribution of the predicate values, which is
the fourth feature of REIN. When the number of buckets is large
enough, the impact of the distribution of the predicate values can
be eliminated by decreasing the size of the buckets and the corre-
spending comparison operations executed in the buckets. To evalu-
ate the effect of the distributions on the matching time, we use our
own data generator to generate predicate values according to three
different distributions: uniform, normal and Pareto. For the normal
distribution, the mean and the variance are set to 0.5 and 0.02, re-
spectively. For the Pareto distribution, the mean and the scale are
set to 0.5 and 2, respectively. Event values and predicate attributes
are generated randomly. Compared with the uniform distribution,
the other two distributions have nearly the same results, as shown in
Fig. 10.

6.3.5. Number of buckets

For REIN, the comparison operations are executed in two buck-
ets for each attribute. Obviously, the size of the buckets affects
the performance of REIN. When the number of subscriptions is large,
more buckets are needed to improve matching efficiency. However,
when the number of buckets reaches a turning point, the perfor-
ance of REIN degrades. This can be explained by the following:
although the size of the buckets is reduced with more buckets,
the cost to switch between buckets increases, which offsets
the benefits obtained from the reduction of comparison operations. As
shown in Fig. 11, when there are 2,000,000 subscriptions, the turn-
ning point of buckets is about 1,000. The matching time first de-
creases with the number of buckets before reaching the turning
point. After the turning point, the performance of REIN decreases
with more buckets. Theorem 2 gives an equation to compute the
optimal number of buckets.

6.4. Construction time

Each of the six tested matching algorithms has its own spe-
cialized index structure. We conduct experiments to measure the
time spent on constructing the index structures for these algo-
rithms.

For SIENA, because an interval predicate is converted into two
simple predicates, the subID of a subscription is stored 2k times in
the index structure, where \( k \) is the number of interval predicates
contained in subscriptions. As for TAMA, the width of the inter-
val predicates determines the times to store the subID of a subscrip-
tion. When \( w = 0.5, \) the subID is stored at least 10k times. For H-
TREE, the subID is stored \( \left\lceil \frac{k}{w} \right\rceil \) times. For BE-TREE, the underly-
ring tree structure needs to split and merge nodes with the insertion
of subscription, so its construction operation is costly. For OpIndex,
an interval predicate is also converted into two simple predicates, just
like SIENA. The times to store the subID in REIN is 2k. These
results are shown in Fig. 12, where \( m = 20, k = 10, w = 0.5, \) and
the y-axis represents the construction time in log scale. As shown in
Fig. 12, SIENA spends the least amount of time constructing its
index structure. REIN and OpIndex have the same scale construc-
tion time. When \( n = 2,000,000 \), the construction time of TAMA,
H-TREE and BE-TREE is, respectively, 14.8, 3.6 and 11.5 times larger
than SIENA. Please note the construction time of SIENA, BE-TREE,
OpIndex and REIN is not affected by the width of interval predic-
tives. However, the construction time of TAMA and H-TREE in-
creases with the width of the interval predicates.

6.5. Deletion time

For TAMA, since the subID of a subscription is stored in multi-
ple buckets, deletion is very time-consuming. The index structure
of SIENA is a two-level matching table, with the first level indexed
on attributes and the second level on operators. There are only
two operators for each attribute, namely \( \leq \) and \( \geq \). Each operator
maps to a bucket, and the number of buckets is \( 2m \), where \( m \) is the
number of attributes appearing in events. For each bucket, the size
is \( n \), where \( n \) is the number of subscriptions. The index structure
of OpIndex is similar to SIENA. When the number of subscriptions
is large, deleting a subscription is also costly for SIENA and OpIndex. Here, we delete 1000 subscriptions from different numbers of subscriptions and compute the average deletion time of one subscription. The results are shown in Fig. 13, where $m = 20$, $k = 10$, $w = 0.5$, and the y-axis represents the deletion time in log scale. By observing the construction time and the deletion time, the fifth feature of REIN is that REIN is applicable to dynamic environments where subscriptions update frequently. Since the binary executable of BE-TREE does not provide an interface to delete subscriptions, BE-TREE is not included in Fig. 13.

6.6. Memory consumption

We also measure the memory consumption of the six matching algorithms with different number of subscriptions. The results are shown in Fig. 14, where $m = 20$, $k = 10$, $w = 0.5$ and the y-axis in log scale. As shown in this figure, BE-TREE occupies the largest memory because each clustering directory contains a large number of levels. This construction method is also adopted by TAMA but with a limited level of discretization. On the contrary, the index structures of the other four matching algorithms are relatively concise. H-TREE consumes the least amount of memory. The memory consumption of REIN is moderate, like SIENA.

7. Discussion

7.1. Dependency of predicates

In this paper, the predicates in a subscription are assumed to be independent. However, this may not hold true in reality. In
practice, methods of machine learning, such as primary component analysis (PCA) [38], can be used to transform the subscriptions and events that may include dependent predicates to the predicates that are independent. PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, i.e., PCA maps a data vector from an original space of \( m \) variables to a new space of \( n \) variables which is uncorrelated over the dataset. Therefore, before constructing the index structure of REIN, subscriptions and events can be transformed by PCA-like methods to guarantee the dependency of predicates.

7.2. Drawback of REIN

The main advantages of REIN over the compared matching algorithms are the improved matching speed and the beneficial anti-matchability feature. One disadvantage of REIN is the delay of the determining times of matching subscriptions when matching events. The matching procedure of REIN can be partitioned into two stages: marking and outputting. In the marking stage, all unmatching subscriptions are marked in the bitset. In the outputting stage, all unmarked bits in the bitset are outputted as matching subscriptions. So, the determining times of matching subscriptions are delayed in REIN, compared with counting-based matching algorithms, such as Oplex-index [12].

8. Conclusion

In this paper, we present REIN, a fast and anti-matchability matching algorithm for large-scale content-based publish/subscribe systems. By convention, pursuing a high matching speed is one of the major objectives when designing matching algorithms, usually without taking into account the effect of the subscriptions’ matchability. One problem caused by ignoring the subscriptions’ matchability is the resulting performance variation of matching algorithms. To tackle this problem, the design objective of REIN is established as pursuing a high matching speed while keeping performance stable. To conquer the impact of the subscriptions’ matchability, REIN utilizes a negative searching strategy, rather than a positive searching strategy which is widely used by existing matching algorithms. Therefore, REIN has five attractive features, namely rapidity, anti-matchability, stability, robustness and dynamism. To evaluate the performance of REIN, comprehensive experiments are conducted. The experiment results show that REIN strongly outperforms its counterparts in terms of matching speed and performance stability.

Acknowledgement

This work was supported by National Key R&D Program of China (2018YFB1003800), the Joint Key Project of the National Natural Science Foundation of China (U1736207), the National Science Foundation of China (61772334, 61702151, 61702320, 61572324), and Shanghai Talent Development Fund, Shanghai Jiao Tong Arts and science inter-project (15JCMY08).

References

Shiyou Qian received the PhD degree from Shanghai Jiao Tong University in 2015. He is currently a research assistant at the Department of Computer Science and Engineering in Shanghai Jiao Tong University. His research interests include event matching for content-based publish/subscribe systems, resource scheduling for Hybrid-Cloud, and driving recommendation with vehicular networks.

Jian Cao received the PhD degree from the Nanjing University of Science and Technology in 2000. He is currently a professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His main research topics include service computing, network computing, and intelligent data analytics. He is a member of the IEEE.

Weichao Mao is currently pursuing the B.S. degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests lie in the interdisciplinary topics between computer science and economics.

Yanmin Zhu is a professor with the Department of Computer Science and Engineering at Shanghai Jiao Tong University. He obtained his PhD in computer science from Hong Kong University of Science and Technology in 2007, and BEng from Xian Jiao Tong University in 2002. Prior to joining Shanghai Jiao Tong University, he was a Research Associate in the Department of Computing in Imperial College London. His research interests include ad hoc sensor networks, vehicular networks, and mobile computing and systems. He is a member of the IEEE, the IEEE Communication Society and the IEEE Computer Society.

Jadi Yu received the PhD degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2007. He is currently an associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. Prior to joining Shanghai Jiao Tong University, he was a postdoctoral fellow in the Data Analysis and Information Security (DASy) Laboratory, Stevens Institute of Technology from 2009 to 2011. His research interests include cyber security and privacy, mobile and pervasive computing, cloud computing, and wireless sensor networks. He is a member of the IEEE and the IEEE Communication Society.

Minglu Li received the PhD degree from Shanghai Jiao Tong University in 1996. He is currently a professor at the Department of Computer and Engineering in Shanghai Jiao Tong University. He is the director of the IBM-SJTU Grid Research Center at Shanghai Jiao Tong University. His main research topics include cloud computing, image processing, and e-commerce.

Jie Wang received the BS degree from Shanghai Jiao Tong University, two MS degrees from Stanford University and the University of Miami, and the PhD degree from Stanford University. He is a consulting professor with Stanford University focusing on interdisciplinary research in adaptive computational learning and reasoning for complex physical and social systems. He has conducted researches at Stanford, worked at several startups in Silicon Valley, and consulted for multi-national companies and for government agencies. Currently, he is the executive director of the Stanford Center for Sustainable Development and Global Competitiveness.