H-Tree: An Efficient Index Structure for Event Matching in Content-Based Publish/Subscribe Systems

Shiyou Qian, Jian Cao, Yanmin Zhu, *Member, IEEE*, Minglu Li, and Jie Wang

Abstract—Content-based publish/subscribe systems have been employed to deal with complex distributed information flows in many applications. It is well recognized that event matching is a fundamental component of such large-scale systems. Event matching searches a space which is composed of all subscriptions. As the scale and complexity of a system grows, the efficiency of event matching becomes more critical to system performance. However, most existing methods suffer significant performance degradation when the system has large numbers of both subscriptions and their component constraints. In this paper, we present Hash Tree (H-Tree), a highly efficient index structure for event matching. H-Tree is a hash table in nature that is a combination of hash lists and hash chaining. A hash list is built up on an indexed attribute by realizing novel overlapping divisions of the attribute's value domain, providing more efficient space consumption. Multiple hash lists are then combined into a hash tree. The basic idea behind H-Tree is that matching efficiencies are improved when the search space is substantially reduced by pruning most of the subscriptions that are not matched. We have implemented H-Tree and conducted extensive experiments in different settings. Experimental results demonstrate that H-Tree has better performance than its counterparts by a large margin. In particular, the matching speed is faster by three orders of magnitude than its counterparts when the numbers of both subscriptions and their component constraints are huge.

Index Terms—Content-based publish/subscribe, event matching, index structure, performance, matching time

1 Introduction

Content-based publish/subscribe (pub/sub) systems have been employed in many applications, such as content dissemination, information filtering, e-commerce, and online games. A pub/sub system realizes decoupling of space, time, and synchronization [1], which is desirable for large-scale distributed scenarios. In recent years, many research prototypes have been built, including Siena [2], Hermes [3], Le Subscribe [4], Scribe [5], and JEDI [6]. In addition, some commercial systems have been implemented, such as IBM WebSphere MQ [7], TIBCO Rendezvous [8] and Oracle Streams Advanced Queuing [9].

It is well known that event matching performance is critical to large-scale pub/sub systems because each broker has to perform this task whenever an event is received. Particularly, inner brokers carry heavy loads since they are the junctions of a pub/sub system and most events will pass through them. When the event arrival rate is higher than an inner broker's processing capacity, that broker becomes a performance bottleneck [10]. Therefore, improving the event matching speed is extremely critical for inner brokers. In essence, event matching is searching a space composed of all subscriptions. The space is determined by two factors: the number of subscriptions and the number of component constraints contained in the subscriptions. The search space is extremely huge when the system scale is large in terms of both the numbers of subscriptions and their component constraints. The event matching speed is particularly a concern in this case.

Not surprisingly, many methods have been proposed to improve matching efficiencies in recent years [11], [12], [13], [14], [15], [16]. The basic strategy behind these methods is that satisfied constraints are first identified, and then counting algorithms are utilized to pick out matched subscriptions. However, when the number of constraints in each subscription is large, much time is wasted on checking large unmatched subscriptions containing satisfied constraints. Therefore, the performance of these methods degrades dramatically when there are millions of subscriptions and each subscription contains tens or hundreds of constraints.

In this paper, we present Hash Tree (H-Tree), an efficient index structure for event matching. H-Tree is a hash table in nature that combines hash lists and hash chaining. The value domain of each indexed attribute is divided into overlapping cells on which a hash list is built. Multiple hash lists are then chained into a hash tree. The basic idea behind H-Tree is that matching efficiencies can be improved when the search space is substantially reduced by pruning most of the subscriptions that cannot be matched.

H-Tree is theoretically analyzed in terms of time and space complexities. In addition, extensive experiments are conducted to evaluate the performance of H-Tree. In the experiments, the number of subscriptions is up to 5 million and the number of component constraints in subscriptions

- *S. Qian, J. Cao, Y. Zhu, and M. Li are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {qshiyou, cao-jian, yzhu, mlli}@sjtu.edu.cn.*
- *J. Wang is with the Department of Civil and Environmental Engineering, Stanford University, Palo Alto, CA. E-mail: jiewang@stanford.edu.*

Manuscript received 4 Aug. 2013; revised 22 Apr. 2014; accepted 4 May 2014. Date of publication 12 May 2014; date of current version 8 May 2015. Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2323262

is up to 200. The experimental results show that H-Tree outperforms its counterparts by a large margin. In particular, the matching speed is faster by three orders of magnitude than its counterparts when the numbers of both subscriptions and their component constraints are large.

The rest of the paper is organized as follows. Section 2 introduces the pub/sub system and its data model. Section 3 presents the design of H-Tree. Section 4 analyzes the performance of H-Tree. Section 5 illustrates the results of the experimental performance evaluation. Section 6 discusses related work. Section 7 concludes the paper. H-Tree was first described in [17]. Compared with the previous version, H-Tree has been improved in multiple aspects, including the analysis of tradeoff between time and space, the design of two algorithms and the proof of two theorems.

2 Pub/Sub Systems

A pub/sub system is composed of publishers, subscribers, and middleware [1]. The middleware is composed of brokers (servers or proxies) that process matching and routing tasks. An example of the pub/sub system is shown in Fig. 1. Note that event matching is performed at each broker. For example, when E_4 receives an event from I_3, event matching is immediately performed at E_4 to forward the event to the matched subscribers.

It can be easily observed that inner brokers have to process more events because they act as dispatchers in the network. When an inner broker receives an event from one of its neighboring brokers, event matching is performed to decide whether the event can be forwarded to other neighboring brokers. When the event arrival speed is greater than an inner broker's processing capacity, the inner broker inevitably becomes a performance bottleneck. This case is just like what happens at a toll station where hundreds of thousands of cars are waiting to pass at rush hour. Therefore, improving event matching speed is critical to the performance of large-scale pub/sub systems.

In the context of content-based pub/sub systems, a $message$ is also called an $event$, which can be represented as a conjunction of attribute-value pairs. For example, $\{(current, 10.4), (voltage, 223), (power, 2300)\}$ is an event which describes the electrical status of a device. Traditionally, each attribute can appear only once in an event. An attribute has its value domain specified by minimum and maximum values. The value domain is assumed to be normalized on [0, 1.0].

A subscriber expresses its interests as a $subscription$ that is identified by a unique $subID$. Generally, a subscription is a conjunction of multiple constraints. Since different forms of constraints can be converted into range constraints, only range constraints are considered in this paper. Following the convention of the pub/sub system Siena [2], a $range\ constraint$ can be represented as a four-tuple of $\{name, lowValue, highValue, type\}$. $Name$ is one of the attribute names appeared in the events. The value $type$ can be either one of the simple types or named complex data types, such as $integer, double, string$. Range constraints are assumed to be in inclusive form, which is equivalent to a conjunction of two simple constraints. For example, the range constraint $\{tem, 10, 20, integer\}$ is equal to $\{tem, \geqslant, 10, integer\} \wedge \{tem, \leqslant, 20, integer\}$. By analyzing the properties of range constraints, we design an efficient index structure for range constraints.

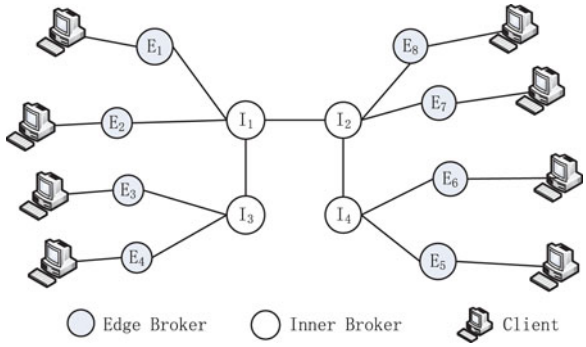

Fig. 1. A distributed pub/sub system where event matching is performed at all brokers.

3 Design of H-Tree

In this section, we first give an overview of H-Tree. The construction procedure of H-Tree is described in detail in each section. The pseudo code for subscription pre-processing and event matching can be found in the online supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2323262.

3.1 Overview

The basic idea behind H-Tree is to filter unmatched subscriptions as much as possible. When matching events, subscriptions can be divided into two categories, matched or unmatched. When all constraints in a subscription are satisfied, we say the subscription is matched. Otherwise, the subscription is unmatched. If parts of constraints in an unmatched subscription are satisfied, we say the subscription is partially matched. Most existing matching algorithms, such as [13], [14], [16], [18], [19], [20], work by first identifying satisfied constraints and then picking out matched subscriptions. One disadvantage of these algorithms is that they waste much time on processing partially matched subscriptions. On the contrary, H-Tree is capable of filtering out most unmatched subscriptions, including partially matched ones.

H-Tree is an index structure for subscriptions that is a combination of hash lists and hash chaining. In H-Tree, there are a number of buckets to store the subIDs of subscriptions. These buckets are identified by bucketIDs. When constructing H-Tree, the value domain of each indexed attribute is divided into cells on which a hash list is built. These cells are identified by cellIDs. Multiple hash lists are then chained into a hash tree. When hashing subscriptions into buckets, a range constraint is mapped to a cellID based on the constraint's center value. The bucketID is determined by the cellIDs of all indexed attributes. When matching an event, a small number of buckets need to be checked. Generally, two or three cellIDs are computed for each indexed attribute and then bucketIDs are calculated according to these cellIDs.

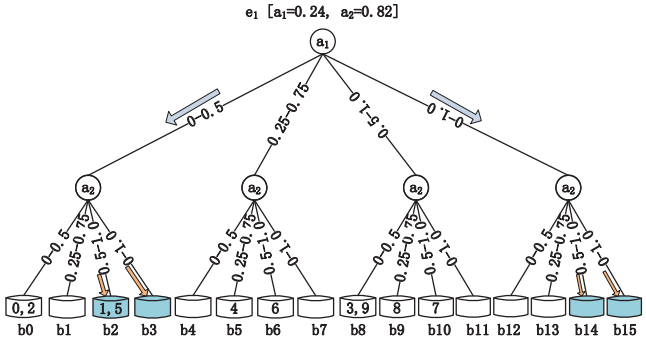

Fig. 2. An illustration of subscription pre-processing and event matching.

TABLE 1
List of Subscriptions

SubID	Content
s0	$0.0 \leqslant a_1 \leqslant 0.1 \wedge 0.2 \leqslant a_2 \leqslant 0.3$
s1	$0.2 \leqslant a_1 \leqslant 0.3 \wedge 0.8 \leqslant a_2 \leqslant 0.9$
s2	$0.2 \leqslant a_1 \leqslant 0.3 \wedge 0.1 \leqslant a_2 \leqslant 0.2$
s3	$0.7 \leqslant a_1 \leqslant 0.8 \wedge 0.3 \leqslant a_2 \leqslant 0.4$
s4	$0.5 \leqslant a_1 \leqslant 0.6 \wedge 0.4 \leqslant a_2 \leqslant 0.5$
s5	$0.1 \leqslant a_1 \leqslant 0.2 \wedge 0.8 \leqslant a_2 \leqslant 0.9$
s6	$0.4 \leqslant a_1 \leqslant 0.5 \wedge 0.6 \leqslant a_2 \leqslant 0.7$
s7	$0.9 \leqslant a_1 \leqslant 1.0 \wedge 0.9 \leqslant a_2 \leqslant 1.0$
s8	$0.6 \leqslant a_1 \leqslant 0.7 \wedge 0.5 \leqslant a_2 \leqslant 0.6$
s9	$0.8 \leqslant a_1 \leqslant 0.9 \wedge 0.3 \leqslant a_2 \leqslant 0.4$

An example is shown in Fig. 2. In the example, there are two indexed attributes, a_1 and a_2. The value domain of each attribute is divided into four cells numbered from 0 to 3. By chaining two indexed attributes, there are 16 buckets identified from $b0$ to $b15$. Ten subscriptions are stored in the system listed in Table 1. When processing a subscription, the cellID of each indexed attribute is computed according to the center of the range constraint. For example, when processing $s3$, the cellID of a_1 is 2 and the cellID of a_2 is 0. So the subID of $s3$ is stored in $b8$. When receiving an event $e_1[a_1 = 0.24,\ a_2 = 0.82]$, the cellIDs of a_1 and a_2 are {0, 3} and {2, 3}, respectively, denoted by arrows in the figure. So the buckets that need to be checked are $b2, b3, b14$ and $b15$. The subIDs in these buckets are used to retrieve the content of subscriptions from the subscription list. Simple matching method is utilized to compute the matched subscriptions. In this example, $s1$ is returned as matched for e_1.

The construction of H-Tree is described as follows. First, attributes to be indexed are selected from the events' attributes. Then, the value domain of each indexed attribute is divided into cells on which a hash list is built. Finally, the hash lists realized on all indexed attributes are chained into a hash tree. The details are described in the following sections.

3.2 Selection of Indexed Attributes

Indexed attributes are first selected from the events' attributes. Obviously, the selection impacts the performance of H-Tree. Some heuristic rules are given as follows. First, popular attributes have higher priority because a hash tree built on popular attributes is less skewed. Let M be the set of attributes in events and S_i be the set of attributes in a subscription. S_i is a subset of M. Given the number of subscriptions n and the number of indexed attributes l, $S = \{x : x \in S_i, 1 \leqslant i \leqslant n\}$, S is a multiset. The discovery of popular attributes is to find the top l most frequent elements in S. The frequencies of elements can be calculated using counting algorithms based on $Map < String, Integer >$. The time complexity of counting algorithms is $O(|S|)$ and the space complexity is $O(|M|)$. We can randomly select the indexed attributes at the initiation stage. Then, it is feasible to discover popular attributes with enough subscriptions using counting algorithms. H-Tree can be reconstructed based on these discovered popular attributes. Second, attributes with more narrow range constraints are preferred since more cells can be divided on these attributes to achieve better matching performance. Given the upper bound of range widths $\mathcal{W}$, constraints with smaller range width than $\mathcal{W}$ are called narrow-range constraints. The upper bound $\mathcal{W}$ should be at least twice of the width of narrow-range constraints. For example, the upper bound $\mathcal{W}$ in Fig. 2 is 0.25 because the width of range constraints specified on a_1 and a_2 is 0.1. Third, the order of indexed attributes has no effect on the performance of H-Tree. For a given subscription, changing the order of indexed attributes just means hashing the subscription into a different bucket. All similar subscriptions are still hashed into a bucket, just changing the ID of the bucket. Finally, the number of indexed attributes is determined by the memory size. Generally, the performance of H-Tree improves with more indexed attributes, which has been verified in the experiments. On average, matching time is reduced 35 percent by adding one additional index attribute. However, the number of buckets grows exponentially with the number of indexed attributes.

3.3 Division of Value Domains

A range constraint is specified by a low value and a high value, which can also be represented by its center and width. Generally, a range constraint cannot be hashed based on a single value because the width of each range constraint is not equal. However, if the widths of range constraints are finite, the number of range constraints that needs to be checked can be reduced when performing matching. In reality, the width of any range constraint specified by users is usually relatively small compared with the whole width of the value domain, such as in [21]. Narrow-range constraints are widely used in real-world applications, such as stock trading, CAD, GIS, and VLSI. For example, a stock trader may define the conditions on buying a stock in the form of range constraints, $\{3.24 \leqslant price \leqslant 3.33 \wedge 50000 \leqslant volume \leqslant 80000\}$. By restricting the width of range constraints under an upper bound, hashing can be built up on the centers of range constraints.

Suppose that the width of range constraints is smaller than an upper bound $\mathcal{W}$ (how to relax this bound is discussed later.) Given the indexed attributes, the number of cells c divided on each indexed attribute's value domain is determined. The maximum of c is limited by $\mathcal{W}$, which is $\lfloor \frac{1}{\mathcal{W}} \rfloor$. By imposing this restriction, we ensure that the width of each cell is not smaller than $\mathcal{W}$. After dividing an indexed attribute's domain into c segments, two neighboring segments are composed into a *cell*. In this way, two neighboring cells overlap with a width of $\frac{1}{c}$. The overlapping division

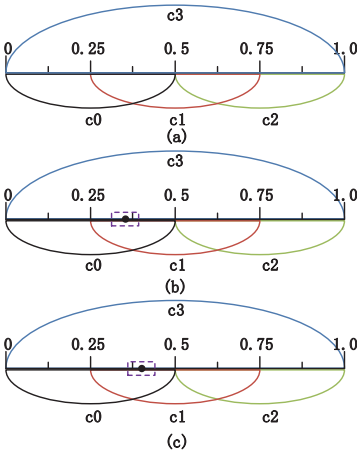

Fig. 3. A novel overlapping division of attribute spaces.

of the indexed attributes' value domain complies with the property of range constraints.

Fig. 3 shows an example in which the upper bound $\mathcal{W}$ is 0.25, and four segments are divided. Three overlapping cells $\{c0, c1, c2\}$ are composed with their centers at {0.25, 0.5, 0.75} respectively, as shown in Fig. 3a. Specifically, the last cell, $c3$, corresponds to the case where no range constraint is specified on the attribute. When processing a subscription, the cellID of each indexed attribute is computed. The rules for computing the cellID of indexed attributes are listed as follows.

- If no range constraint is specified on the indexed attribute, the cellID is $c3$.
- If the center of the range constraint is located in interval [0, 0.25] or [0.75, 1.0], the cellID is $c0$ or $c2$.
- Otherwise, the range constraint is located in two cells. The cellID is determined by the distance from the center of the range constraint to the center of the two overlapping cells.

In Fig. 3b, a range constraint, denoted as a dashed rectangle, is located in two cells, $c0$ and $c1$. The cellID of the constraint is $c0$, because the center of the range constraint, denoted as a solid point, is closer to the center of $c0$ than to the center of $c1$. In Fig. 3c, another range constraint with the same width but at a different center is located in the same cells, $c0$ and $c1$. But the cellID of the constraint is $c1$ because the center of the range constraint is closer to the center of $c1$.

Given the upper bound of range widths, overlapping division of range constraints is more efficient in memory consumption than non-overlapping division. First, in the case of non-overlapping division, a range constraint may span two cells. Therefore, the subID of a subscription should be stored in 2^l buckets where l is the number of indexed attributes. On the contrary, a range constraint is totally contained by an overlapping cell. The subID of a subscription is stored in only one bucket. Second, under the same conditions, overlapping division requires fewer buckets than non-overlapping division. The number of overlapping cells is equal to the number of non-overlapping cells minus one. Nevertheless, the number of buckets is increased exponentially with the number of cells. For example, when the upper bound $\mathcal{W}$ is 0.2 and the number of indexed attributes is 8, the number of buckets created by the non-overlapping method is almost six times the number of buckets created by the overlapping method.

The above domain division assumes uniform distribution of the constraints' attribute values, so the value domain of indexed attributes is evenly divided. It is obvious that this will not work perfectly when the distribution of the constraints' attribute values is skewed, which is normal for real-world applications. However, given the distribution function of the constraints' attribute values $F(x)$, non-uniform distribution can be converted into uniform distribution.

The conversion of non-uniform distribution to uniform distribution is solved by the Probability Integral Transformation theorem [22]. Let X be any continuous random variable with a probability density $p(x)$, and let $F(x)$ be its cumulative distribution function (CDF). A new random variable Y is defined as $Y = F(X)$. Then the Probability Integral Transform theorem states that $Y = F(X)$ has a uniform distribution on [0, 1.0]. For example, when X is a random variable with a standard normal distribution $N(0, 1)$, then its CDF is

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt = \frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right], x \in \mathcal{R}, \quad (1)$$

where $erf()$ is the error function which is defined as

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt. \quad (2)$$

As for pareto distribution, its CDF is

$$F(x) = \begin{cases} 1 - \left(\frac{x_m}{x}\right)^{\alpha} & for\ x \geq x_m \\ 0 & for\ x < x_m. \end{cases} \quad (3)$$

Without loss of generality, the constraints' and events' attribute values are generated according to these two distributions to evaluate the performance of H-Tree in the experiments. The transformation is very simple without any costly computation. Therefore, the cost of transformation is negligible.

3.4 Chaining of Hash Lists

Based on overlapping division, the hash lists of all indexed attributes are chained into a hash tree, just like that shown in Fig. 2. The chaining of indexed attributes naturally corresponds to the conjunction relationships of constraints in subscriptions. In nature, H-Tree is a hash table created by putting similar subscriptions in a bucket. The similarity of subscriptions is measured by the centers of the range constraints. When the cellID of each indexed attribute in a subscription is known, the appropriate bucket in which to store the subscription is easy to calculate. Only the subIDs of subscriptions are stored in the buckets, not the content of the subscription, so storage consumption is very small.

This division and chaining approach has many advantages. First, an attribute value is at most located in two cells after imposing an upper bound onto the range width. Our aim is to design an index structure which is capable of filtering out most unmatched subscriptions. When matching events, at most three cells are selected for each indexed attribute. Let c be the number of cells divided, so the percentage of filtered subscriptions for each indexed attribute

is $\frac{c-3}{c}$. For example, when c is 5, at least 40 percent of subscriptions are filtered out at each level in H-Tree. By chaining multiple hash lists, subscriptions that finally need to be checked are exponentially decreased with the number of chained indexed attributes.

Second, similar subscriptions are hashed into a bucket based on the centers of multiple range constraints. Subscriptions that need to be checked have a high matching probability because most unrelated subscriptions have been filtered out level by level. Therefore, H-Tree has the ability to return matched subscriptions earlier than traditional matching algorithms. One advantage of this ability is that events can be forwarded as soon as possible at inner brokers, and the end-to-end latency is reduced in turn.

3.5 Tradeoff between Time and Space

In order to evaluate the index efficiency of a structure, the filtration ratio f is defined as

$$f = \frac{n_r}{n_a}, \tag{4}$$

where n_r is the number of remained subscriptions after filtering and n_a is the number of all subscriptions. The matching time directly relates to the filtration ratio f. In the case of uniform distribution, all subscriptions are evenly stored in the buckets. The number of buckets is c^l. When matching events, two or three cellIDs are determined for each indexed attribute, so the number of buckets that needs to be checked is at most 3^l. The filtration ratio f is

$$f = \frac{3^l}{c^l} = \left(\frac{3}{c}\right)^l, \tag{5}$$

which is exponentially decreased with the number of indexed attributes since c is larger than 3.

The performance of H-Tree can be evaluated in terms of matching time and memory consumption. The matching time of H-Tree is reduced with the growth of l and c. However, memory consumption increases exponentially with l and c. Given the number of indexed attributes l, the number of cells can be determined by taking into consideration both matching efficiency and memory consumption. As defined above, the filtration ratio f depicts the index efficiency of H-Tree, which can also be used to represent matching efficiency. The memory consumption of H-Tree can be represented by the number of buckets, which is c^l.

Suppose the value domain of c is $[L, R]$, and the minimum and maximum of f are $(\frac{3}{R})^l$ and $(\frac{3}{L})^l$. The minimum and maximum of memory consumption are L^l and R^l. A unity function is established by normalization, which is shown as:

$$\alpha \frac{\left(\frac{3}{c}\right)^l - \left(\frac{3}{R}\right)^l}{\left(\frac{3}{L}\right)^l - \left(\frac{3}{R}\right)^l} + \beta \frac{c^l - L^l}{R^l - L^l}, \tag{6}$$

where α and β are weights of matching efficiency and memory consumption. The sum of α and β is equal to 1. The values of α and β reflect the importance of matching efficiency and memory consumption. Their values can be adjusted in terms of application requirements. By obtaining the first derivative and setting it to 0, the equation is:

$$c^{2l} = 3^l \frac{\alpha(R^l - L^l)}{\beta((\frac{3}{L})^l - (\frac{3}{R})^l)}. \tag{7}$$

Solving the equation, the extreme point of c is:

$$c = \sqrt[2l]{3^l \frac{\alpha(R^l - L^l)}{\beta((\frac{3}{L})^l - (\frac{3}{R})^l)}}. \tag{8}$$

Given the values of α, β and l, the value computed in Equation (8) is the optimal selected from the value domain $[L, R]$. Similarly, given the values of α, β and c, the optimal value for l can also be computed in the same way.

TABLE 2
Parameters Used in the Experiments

Parameter name	Meaning
n	the number of subscriptions
m	the number of attributes
k	the number of range constraints
c	the number of cells
l	the number of indexed attributes
w	width of range constraints
a	parameter of Zipf distribution

4 Theoretical Performance Analysis

H-Tree is analyzed in terms of time and space complexities. Some other properties are discussed in the online supplementary file, available online.

A subscription is called a narrow-range subscription when all of its range widths are smaller than the upper bound $\mathcal{W}$. Otherwise, a subscription is called a wide-range subscription. The upper bound of the range width is addressed by splitting a wide-range subscription into multiple narrow-range subscriptions. One defect of H-Tree is that the subID of a wide-range subscription is stored in multiple buckets. Let u be the number of wide range constraints whose widths are larger than $\mathcal{W}$, where $u \leqslant k$. We assume that each attribute has the same probability of being selected as an indexed attribute. The parameters used in the following analysis are defined in Table 2. Based on these parameters, a wide-range subscription is split into s narrow-range subscriptions, which is computed by:

$$s = \left\lceil \frac{w}{1/c} \right\rceil^{\frac{lu}{mk}} = \lceil (wc) \rceil^{\frac{lu}{mk}}. \tag{9}$$

4.1 Pre-Processing Time

For a narrow-range subscription, its subID is stored only once. When computing the bucketID for a subscription, the cellIDs of indexed attributes are computed. The time complexity of pre-processing is $O(ln)$. The time complexity of pre-processing wide-range subscriptions is $O(sln)$.

4.2 Matching Time

The time complexity of matching is $O(fn)$, where f is the filtration ratio. When c is 8 and l is 8, the value of f is 0.00039,

which means that only 0.39‰ of total subscriptions remain to be checked after the filtering of unrelated subscriptions. The time complexity of matching wide-range subscriptions is $O(sfn)$.

4.3 Insertion Time

Inserting a narrow-range subscription involves computing the bucketID and adding the subID of the subscription to the bucket. The time complexity of inserting a narrow-range subscription and a wide-range subscription is $O(l)$ and $O(sl)$, respectively.

4.4 Deletion Time

When giving the subID of a subscription, the content of the subscription is first retrieved from the subscription list. The bucketID storing the subscription is determined. Then the subID of the subscription is removed from the bucket by simple searching method. Only the subIDs in one bucket are compared when deleting a subscription. On average, the number of subIDs in a bucket is $\frac{n}{c^l}$, so the time complexity of deleting a narrow-range subscription is $O(\frac{n}{c^l})$. The time complexity of deleting a wide-range subscription is $O(\frac{sn}{c^l})$.

4.5 Storage Consumption

The subID of a narrow-range subscription is stored only once. The space complexity is $O(n)$. For a wide-range subscription, the subID is stored in multiple buckets after splitting. The space complexity is $O(sn)$.

5 Experimental Performance Evaluation

Experimental evaluation results are presented in this section. We measure matching time, insertion time, deletion time, and memory consumption to compare the performance of H-Tree with its counterparts. All experiments are conducted on a Dell PowerEdge T710 with eight 2 GHz cores and 32 GB memory running Ubuntu 11.10 with Linux kernel 3.0.0-12. Parallelism is not utilized in all experiments. All code is written in C++ language. More evaluation results can be found in the online supplementary file, available online.

5.1 Experimental Settings

The performance of an index structure is influenced by many parameters. In order to extensively evaluate the performance of H-Tree, these parameters are identified in Table 2. Different experiments are conducted to observe the impacts of these parameters on the matching performance.

Two methods are chosen to compare with H-Tree, namely Simple and Tama. Simple utilizes a naive matching policy which compares an event with all subscriptions. This policy is further optimized in two aspects. First, attributes are numbered so that a range constraint is compared with the corresponding attribute value quickly. Second, when one range constraint in a subscription is not satisfied, it is not processed any longer and the next subscription is checked immediately. Tama is an approximate matching and forwarding engine which was recently proposed [16]. Tama trades matching accuracy for improvement on matching time. Tama uses a hierarchical matching table to store

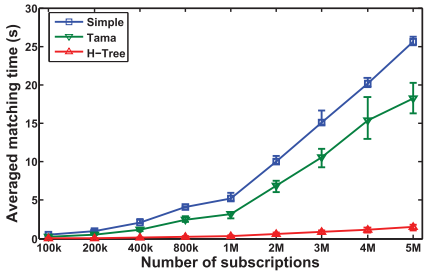

Fig. 4. Matching time with the number of subscriptions under uniform distribution, where $m = 50$, $k = 20$, $w = 0.1$, $c = 8$, $l = 8$.

subscriptions. The first two layers are linear tables, indexed on the attributes and the discretization levels, respectively; the last layer is a hash map. Event matching in Tama becomes the query to this table. The discretization level of Tama is set to 13 in the experiments.

5.2 Matching Time

Matching time is the most important metric for evaluating a matching algorithm. Matching time is influenced by many parameters. Comprehensive experiments have been conducted to observe the impacts of these parameters in different settings. Ten thousand events are input to the system and the average matching time per event is measured.

5.2.1 *Matching Time with the Number of Subscriptions*

In general, matching time is linear to the number of subscriptions. The results are shown in Fig. 4. H-Tree is 12 times faster than Tama when the number of subscriptions is 5M. The matching time of H-Tree fluctuates least compared with the other two methods. The standard deviation of Simple, Tama and H-Tree is 0.265, 0.692 and 0.164, respectively, when the number of subscriptions is 5M. Matching time fluctuation is influenced by the distribution of the constraints' attribute values and events' attribute values.

5.2.2 *Matching Time with the Distribution*

To evaluate the impact of distribution on matching time, two experiments are conducted. In the first experiment, we generate the constraints' attribute values according to three different distributions, uniform, normal, and pareto. For the normal distribution, the mean is 0.5 and the variance is 0.02. For the pareto distribution, the mean is 0.5 and the scale is 2. The events' attribute values and constraint attributes are generated uniformly. Compared with uniform distribution, these two other distributions may cause H-Tree to be skewed. As mentioned in Section 3, converting other forms of distribution into uniform is beneficial in balancing H-Tree. We compare the benefits obtained by the distribution conversion. The results are shown in Fig. 5. Before the distribution conversion, the matching time of H-Tree under normal and pareto distributions is larger than the one under uniform distribution. The benefit of conversion is obvious. The improved matching performance is 12 and 25 percent for the normal and the pareto distributions, respectively. The second experiment has similar results which can be found in the online supplementary file, available online.

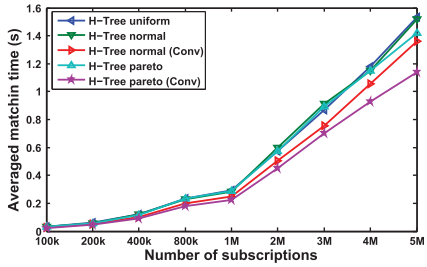

Fig. 5. Matching time with the number of subscriptions under different distribution of constraints' attribute values, where $m = 50$, $k = 20$, $w = 0.02$, $c = 8$, $l = 8$.

5.2.3 Matching Time with the Number of Constraints Contained in Subscriptions

We present H-Tree for large-scale pub/sub systems in terms of numbers of both subscriptions and component constraints in each subscription. H-Tree is especially capable of handling matching tasks where the number of range constraints is at the scale of tens or hundreds. The performance of Simple degrades markedly with the number of range constraints. With increasing numbers of range constraints, partially matched subscriptions computed by Tama are large, which causes more time spent on counting. When the number of constraints is at the scale of tens, almost every subscription is partially matched. Therefore, counting algorithms need to sum up the satisfied constraints for each subscription and compare that with the number of all constraints contained in the subscription to judge whether the subscription is matched. By chaining multiple hash lists into a hash tree, H-Tree handles it more efficiently. The results are shown in Fig. 6. On the average, H-Tree is almost 30 times faster than Tama.

One prominent merit of H-Tree is matching performance improves with the number of range constraints in subscriptions, given the number of attributes. The ratio of the number of constraints to the number of attributes (RCA) is defined as:

$$RCA = \frac{k}{m}. \tag{10}$$

The impact of RCA on the matching time is shown in Fig. 7. In the experiment, the number of attributes is fixed at 100 and the number of constraints is variant. Two observations are found in the results. First, when the number of constraints is large, Tama behaves similarly to Simple since almost every subscription is partially matched. Second, H-Tree performs perfectly with larger numbers of constraints, which can be explained as subscriptions are more evenly hashed into the buckets with higher RCA. For an

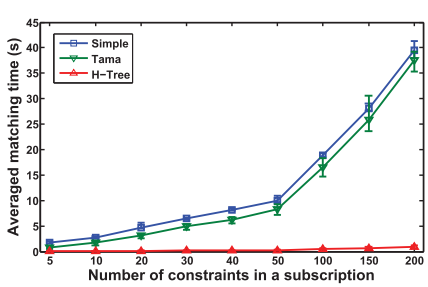

Fig. 6. Matching time with the number of constraints, where $n = 1M$, $m = k * 2$, $w = 0.1$, $c = 8$, $l = 8$.

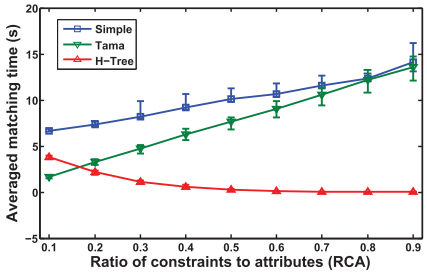

Fig. 7. Matching time with RCA, where $n = 1M$, $m = 100$, $w = 0.1$, $c = 8$, $l = 8$.

indexed attribute, if there is no constraint specified on it in a subscription, it is hashed into the last cell specialized for range [0, 1.0]. When there are a large number of subscriptions without specifying constraints on the indexed attributes, the resulting H-Tree is skewed. When RCA is 0.1, the standard deviation of H-Tree is 0.078. However, when RCA is 0.9, the standard deviation is reduced to 0.005. The standard deviation of H-Tree decreases with RCA.

5.2.4 Matching Time with the Number of Indexed Attributes

Index efficiencies are improved when there are more attributes to be indexed. This is verified by the experimental results shown in Fig. 8. However, more indexed attributes mean more buckets, given the number of cells. The number of indexed attributes is influenced by the number of subscriptions and the number of cells. The number of subscriptions per bucket (SPB) is used to indicate the size of the buckets. In theory, lower SPB means higher index efficiencies but at the cost of more memory consumption. The results of experiments show that SPBs between 100 and 500 are sufficient to provide efficient index performance. As shown in the figure, the matching time is reduced at least 29 percent by adding one indexed attribute. When the number of subscriptions is 5M and the number of indexed attributes is 4, H-Tree is almost five times faster than Tama. However, when the number of indexed attributes is 8, H-Tree is 35 times faster than Tama.

5.2.5 Matching Time with the Number of Cells

The number of cells is related to the range width. When constructing H-Tree, at least four segments are divided to compose three cells. If the width of range constraints is small, more cells can be divided to get better index effects. In general, 5-10 cells are modest, which is verified in the experiment shown in Fig. 9. When the number of subscriptions is

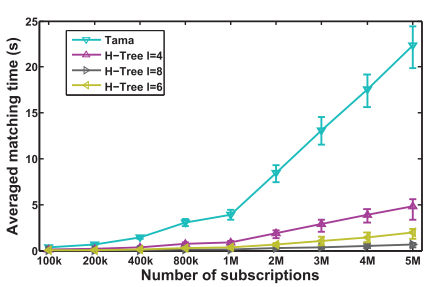

Fig. 8. The impacts of the number of indexed attributes on matching time, where $m = 50$, $k = 25$, $w = 0.1$, $c = 8$.

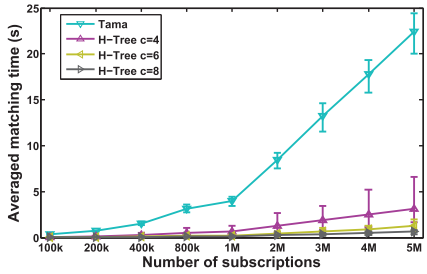

Fig. 9. The impacts of the number of cells on matching time, where $m = 50$, $k = 25$, $w = 0.1$, $l = 8$.

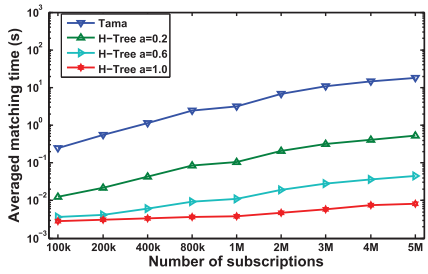

Fig. 11. Matching time with the distribution of attributes selection, where $m = 50$, $k = 20$, $w = 0.1$, $c = 8$, $l = 8$.

5M, the matching time is reduced 45 percent by increasing the number of cells from 4 to 5. The effect of the reduction is alleviated with larger numbers of cells. The reduction is 20% when the number of cells is increased from 7 to 8. H-Tree is seven times faster than Tama when the number of cells is 4. The difference is magnified with larger numbers of cells. When the number of cells is 8, H-Tree is almost 35 times faster than Tama.

5.2.6 Matching Time with Range Width

Range width has no impact on Simple. As for Tama, large range width means more matching time because the subID of a subscription is stored in more buckets. Meanwhile, a higher discretization level is needed to realize a low false positive rate due to the overlapping of range constraints. When the width of range constraints is smaller than the upper bound $\mathcal{W}$, the performance of H-Tree is not impacted. Two experiments are conducted to evaluate the impacts of the range width on matching time. The first experiment tests 1M subscriptions with different fixed range widths smaller than $\mathcal{W}$. The results are plotted in Fig. 10. Just as discussed above, Simple and H-Tree are not influenced by the width. However, the matching time of Tama is increased linearly with the width. When the width is 0.1, H-Tree is 44 and 34 times faster than Simple and Tama, respectively. The matching time of H-Tree fluctuates least compared with the other two methods. The second experiment has similar results which can be found in the online supplementary file, available online.

5.2.7 Matching Time with the Distribution of Attributes Selection

In some cases where the constraint attributes are not uniformly selected from the events' attributes, some attributes appear more frequently than others. Mathematically, when something is shared among a sufficiently large set of participants, there must be a number k between 50 and 100 such that k percent is taken by $(100 - k)$ percent of the participants [23]. This phenomenon is called "Zipf distribution" which was found in web request [24]. "Zipf distribution" is beneficial to H-Tree because when H-Tree is built on popular attributes, rather than unpopular, it is less skewed. We simulate the distribution of attributes selection as Zipf. The value of a is varied. Fig. 11 shows the results with the y-axis representing matching time in log-scale. When a is 1.0 and the number of subscriptions is 5M, H-Tree is at most 2,190 times faster than Tama. The standard deviation of H-Tree decreases with the value of a, which verifies our analysis that H-Tree built on popular attributes is less skewed.

5.3 Maintenance Cost

The insertion time, deletion time and memory consumption are measured for the algorithms.

5.3.1 Insertion Time

For Simple, inserting a subscription just means adding it to the list of subscriptions. Tama and H-Tree have their specialized index structures. We measure the time when inserting narrow-range subscriptions for each algorithms. Ten runs of experiments are conducted. The average insertion time is computed and the results are shown in Fig. 12. Tama uses more insertion time than its counterparts because the subID of subscriptions is stored in multiple buckets. H-Tree spends a little more time, less than 10 percent, than Simple. However, the insertion time of Tama is almost two times that of Simple.

The performance of H-Tree degrades when inserting wide-range subscriptions because the subID of a wide-range subscription is stored in multiple buckets. We measure the time when inserting subscriptions with random range width. The $lowValue$ and $highValue$ of each range

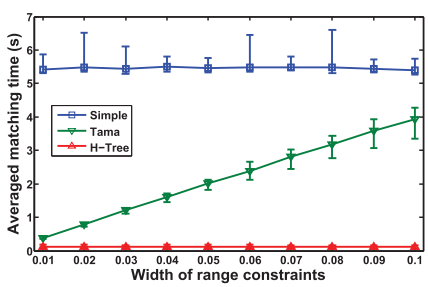

Fig. 10. The impacts of fixed range width on matching time, where $n = 1M$, $m = 50$, $k = 25$, $c = 8$, $l = 8$.

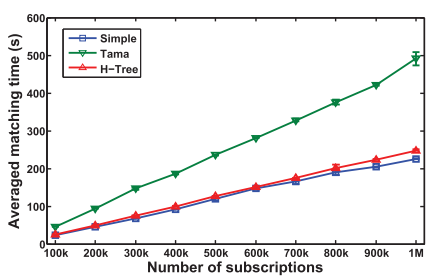

Fig. 12. Insertion time with fixed width subscriptions, where $m = 50$, $k = 20$, $w = 0.1$, $c = 8$, $l = 8$.

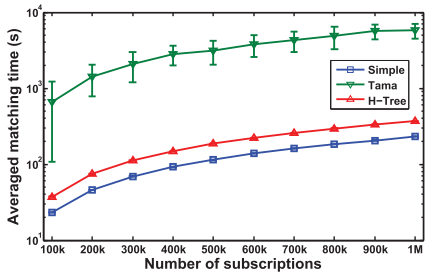

Fig. 13. Insertion time with random width subscriptions, where $m = 50$, $k = 20$, $w = random$, $c = 8$, $l = 8$.

constraint are randomly generated. The results are shown in Fig. 13 where the y-axis represents insertion time in log-scale. H-Tree spends 60 percent more time than Simple when inserting 1M subscriptions. The matching time of H-Tree is 200 percent faster than Simple. Compared with the results of fixed width, the insertion times of Simple and H-Tree fluctuate only a little. However, the insertion time of Tama fluctuates greatly.

5.3.2 Deletion Time

For Tama, deletion operation is time-consuming since the subID of a subscription is stored in multiple buckets. We measure the deletion time per subscription by deleting 1,000 subscriptions from different numbers of subscriptions. The number of indexed attributes impacts the deletion time of H-Tree. H-Tree is evaluated under different values of l. The results are shown in Fig. 14 where the y-axis represents deletion time in log-scale. Given the number of subscriptions, the size of the buckets decreases with more indexed attributes. Therefore, the deletion time of H-Tree decreases with the number of indexed attributes.

5.3.3 Memory Consumption

A range constraint is specified by a low value and a high value. The raw storage of a range constraint is 16 bytes, two *double* for values. A subscription needs one *integer* to store its subID. There is no additional storage requirement for Simple. We analyze Tama in the case where the range width is 0.1 and the discretization level is 13. The subID of a subscription is stored at least nine times, which consumes 36 bytes. The number of constraints in each subscription needs to be stored, which occupies 2 bytes. When the range width is 0.1 and the number of cells is less than 10, H-Tree just stores the subID of a subscription in one bucket. Additional storage for H-Tree is 4 bytes for each subscription. The storage consumption is compared in Table 3 for a single range constraint and a subscription composed of 10 constraints. The memory consumption with the number of subscriptions is plotted in Fig. 15. As shown in the figure, H-Tree consumes a little more memory than Simple, but much less than Tama.

TABLE 3
Storage Consumption of a Constraint and a Subscription

type	Simple	Tama	H-Tree
a constraint	16B	52B	16B
a subscription	164B	526B	168B

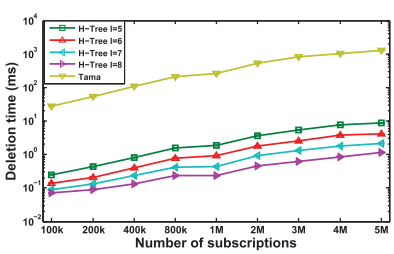

Fig. 14. Deletion time per subscription, where $m = 100$, $k = 30$, $w = 0.05$, $c = 10$.

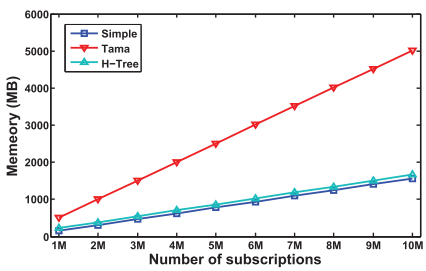

Fig. 15. Memory consumption counted in memory space needed to store subscriptions and subIDs.

6 Related Work

We review related work in this section. Additional discussion on other related work including tree-like structures and parallel algorithms is provided in the online supplementary file, available online.

6.1 Counting Algorithms Based

Different index structures have been designed to improve matching efficiencies based on counting algorithms [13], [14], [16], [18], [19], [20]. The matching procedure of these methods is separated into two phases. In the first phrase, satisfied constraints are computed. In the second phase, matched subscriptions are returned by utilizing counting algorithms. [13] and [16] are two representative ones. In [25], bloom filters are used to store matched primitive constraints. One disadvantage of these methods is that a subscription may be counted multiple times as a partially matched subscription, which wastes much time. Therefore, the time complexity of counting algorithms is linear with the number of partially matched subscriptions. Compared with these methods, H-Tree is more efficient by filtering out most partially matched subscriptions.

6.2 Reducing Routing Table Size

Matching time is decreased when the number of subscriptions is reduced. Therefore, reducing the routing table size is another way to improve matching efficiencies. In [26], subscriptions are summarized to reduce the routing table size by using imperfect merging. Similar approaches are subscription subsumption and covering, such as [27], [28], [29], [30], [31]. The order of subscriptions, cover relation, matching history, and routing destination are considered simultaneously to provide efficient event matching in [32]. These methods are orthogonal to our proposed method. However, the time complexity of subscription covering and

subsumption is relatively high and one drawback of imperfect merging is the waste of bandwidth due to the existence of false positives. Moreover, merging, covering and subsumption are not applicable to the dynamic environments where subscriptions are frequently updated.

7 Conclusion

In this paper, we present H-Tree, an efficient index structure for event matching in large-scale content-based publish/subscribe systems. The basic idea behind H-Tree is that matching efficiencies can be improved when unmatched subscriptions can be filtered out in the matching process. H-Tree is a hash table in nature that combines hash lists and hash chaining. The novelty is that hash lists are built up on the overlapping cells which are divisions of the value domain of indexed attributes. Extensive experiments are conducted to evaluate the performance of H-Tree and experimental results show that H-Tree outperforms its counterparts to a large degree, especially in the cases where both the numbers of subscriptions and their component constraints are large.

Acknowledgments

The work was supported by Morgan Stanley (No. CIP-A20110324-2). This work was also partially supported by NSFC (No. 61272438, and 61170238), Research Funds of Science and Technology Commission of Shanghai Municipality (No. 12511502704), China 973 Program (2014CB340303), Singapore NRF (CREATE E2S2), National 863 Program (2013AA01A601), and Program for Changjiang Scholars and Innovative Research Team in University(IRT1158, PCSIRT), China. Prof. Jian Cao is the corresponding author.

References

[1] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[2] A. Carzaniga, D. Rosenblum, and A. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, no. 3, pp. 332–383, 2001.

[3] P. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *Proc. 22nd Int. Conf. Distrib. Comput. Syst. Workshops*, 2002, pp. 611–618.

[4] F. Fabret, F. Llirbat, J. A. Pereira, I. Rocquencourt, and D. Shasha, "Efficient matching for content-based publish/subscribe systems," Tech. Rep., INRIA, pp.1–20, 2000.

[5] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," *Netw. Group Commun.*, vol. 2243, pp. 30–43, 2001.

[6] G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Trans. Softw. Eng.*, vol. 27, no. 9, pp. 827–850, Sep. 2001.

[7] IBM WebSphere MQ 7.5. IBM Corporation. (2012). [Online]. Available: http://www-01.ibm.com/software/integration/wmq/

[8] TIBCO Rendezvous 8.3.0. (2010). [Online]. Available: http://www.tibco.com/products/automation/messaging/low-latency/rendezvo us/default.jsp

[9] D. Raphaely, N. Bhatt, and C. Hall . (2007). Streams advanced queuing user's guide 11g release 1 (11.1). [Online]. Available: http://docs.oracle.com/cd/B28359_01/server.111/b28420.pdf

[10] A. Cheung and H. Jacobsen, "Load balancing content-based publish/subscribe systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, p. 9, 2010.

[11] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching events in a content-based subscription system," in *Proc. ACM Symp. Principles Distrib. Comput.*, 1999, pp. 53–61.

[12] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," in *Proc. 23rd Int. Conf. Softw. Eng.*, 2001, pp. 443–452.

[13] A. Carzaniga and A. Wolf, "Forwarding in a content-based network," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2003, pp. 163–174.

[14] F. Fabret, H. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2001, pp. 115–126.

[15] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien, "Efficiently evaluating complex boolean expressions," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2010, pp. 3–14.

[16] Y. Zhao and J. Wu, "Towards approximate event processing in a large-scale content-based network," in *Proc. 22nd Int. Conf. Distrib. Comput. Syst. Workshops*, 2011, pp. 790–799.

[17] S. Qian, J. Cao, Y. Zhu, M. Li, and J. Wang, "H-tree: An efficient index structure for event matching in publish/subscribe systems," in *Proc. IEEE IFIP Netw. Conf.*, 2013, pp. 1–9.

[18] H. Jafarpour, S. Mehrotra, N. Venkatasubramanian, and M. Montanari, "MICS: An efficient content space representation model for publish/subscribe systems," in *Proc. 3rd ACM Int. Conf. Distrib. Event-Based Syst.*, 2009, p. 7.

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

[20] G. Ashayer, H. Leung, and H. Jacobsen, "Predicate matching and subscription matching in publish/subscribe systems," in *Proc. 22nd Int. Conf. Distrib. Comput. Syst. Workshops*, 2002, pp. 539–546.

[21] M. Krátkỳ, V. Snášel, P. Zezula, J. Pokornỳ, and T. Skopal, "Efficient processing of narrow range queries in the R-tree," Amphora Research Group, Dept. Comput. Sci., VSB-Tech. Univ. Ostrava, Ostrava, Czech Republic, Tech. Rep. ARG-TR-01-2004, 2004.

[22] R. Fisher, *Statistical methods for research workers*, Guwahati, India: Genesis Publishing Pvt Ltd, 1925.

[23] G. Zipf, "Relative frequency as a determinant of phonetic change," *Harvard Studies Classical Philology*, vol. 40, pp. 1–95, 1929.

[24] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE Conf. Comput. Commun.*, 1999, pp. 126–134.

[25] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in *Proc. 2nd Int. Conf. Distrib. Event-Based Syst.*, 2008, pp. 71–81.

[26] P. Triantafillou and A. Economides, "Subscription summarization: A new paradigm for efficient publish/subscribe systems," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2004, pp. 562–571.

[27] G. Li, S. Hou, and H. Jacobsen, "A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams," in *Proc. 25th IEEE Int. Conf. Distrib. Comput. Syst.*, 2005, pp. 447–457.

[28] A. Ouksel, O. Jurca, I. Podnar, and K. Aberer, "Efficient probabilistic subsumption checking for content-based publish/subscribe systems," in *Proc. 7th ACM/IFIP/USENIX Int. Conf. Middleware*, 2006, pp. 121–140.

[29] K. Jayaram and P. Eugster, "Split and subsume: Subscription normalization for effective content-based messaging," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2011, pp. 824–835.

[30] Z. Shen and S. Tirthapura, "Approximate covering detection among content-based subscriptions using space filling curves," *J. Parallel Distrib. Comput.*, vol. 72, no. 12, pp. 1591–1602, 2012.

[31] Z. Liu, S. Parthasarthy, A. Ranganathan, and H. Yang, "Scalable event matching for overlapping subscriptions in pub/sub systems," in *Proc. Int. Conf. Distrib. Event-Based Syst.*, 2007, pp. 250–261.

[32] X. Guo, J. Wei, and D. Han, "Efficient event matching in publish/subscribe: Based on routing destination and matching history," in *Proc. Int. Conf. Netw., Archit., Storage*, pp. 129–136 2008.

Shiyou Qian received the master's degree in computer science from the Department of Computer Science in China University of Mining and Technology. He is currently working toward the PhD degree at the Department of Computer Science and Engineering in Shanghai Jiao Tong University. His research interests include matching in pub/sub systems 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 at the Department of Computer Science and Engineering in Shanghai Jiao Tong University. His main research topics include service computing, cooperative information systems, and software engineering. He has published more than 50 papers.

Yanmin Zhu received the PhD degree in computer science from the Hong Kong University of Science and Technology in 2007, and BEng degree from Xian Jiao Tong University in 2002. He is an associate professor with the Department of Computer Science and Engineering at Shanghai Jiao Tong University. 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.

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 grid computing, image processing, and e-commerce.

Jie Wang received the PhD degree from Stanford University, and he is currently a consulting professor at Stanford University. He conducts research in engineering and environmental informatics and knowledge management for sustainable development. He has also worked for a number of companies and government agencies including Collation, Inc., EDS, Loudcloud, Inc, Stanford University ITSS, Instantis, Inc., First Union Bank, Department of Energy, and NOAA.