Networking 2013 1569702283

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

Shiyou Qian †, Jian Cao †, Yanmin Zhu †, Minglu Li †, and Jie Wang ⋆
† Shanghai Jiao Tong University
⋆Stanford University
†{qshiyou, cao-jian, yzhu, mlli}@sjtu.edu.cn
⋆Jiewang@stanford.edu

***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 is to search in a space which is composed of all subscriptions. As the scale and complexity of a system grow, the efficiencies of event matching become more critical to the system performance. Most existing methods suffer performance degradation problem when a system has both large number of subscriptions and large number of constraints. In this paper, we present H-$Tree$ (Hash Tree), a highly efficient index structure for event matching. H-$Tree$ is a hash table in nature which is a combination of hash lists and hash chaining. A hash list is realized on an indexed attribute by dividing the attribute's value domain into cells. Multiple hash lists are chained 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 impossible subscriptions. We have implemented H-$Tree$ and conducted extensive experiments in different settings. Experimental results show that H-$Tree$ outperforms its counterparts to a large degree. In particular, the matching time is faster by three order of magnitude than its counterparts when both the number of subscriptions and the number of constraints are large.**

I. 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. 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, TIBCO Rendezvous and Oracle Streams Advanced Queuing.

It is well known that event matching is critical to large-scale pub/sub systems because each broker performs this task when receiving an event. Particularly, inner brokers are junctions since most events pass through them. When the number of received events is greater than an inner broker's processing capacity, the inner broker becomes a performance bottleneck [7]. Therefore, sustaining fast event matching is extremely necessary. In essence, event matching is to search a space which is composed of all subscriptions. When both the number of subscriptions and the number of constraints contained in subscriptions, the search space is extremely huge. Event matching is challenging when the system scale is large.

Many methods have been proposed to improve matching efficiencies [8–13]. The basic strategy behind these methods is that partially matched subscriptions are first computed and then counting algorithms are utilized to get fully matched subscriptions. However, when the number of constraints is large, almost all subscriptions are partially matched. Therefore, the matching performance of these methods degrades dramatically when there are millions of subscriptions and each subscription contains tens or hundreds of constraints.

In the paper, we present H-$Tree$ (Hash Tree), an efficient index structure for event matching. H-$Tree$ is a hash table in nature which is a combination of hash lists and hash chaining. The value domain of each indexed attribute is divided into cells on which a hash list is realized. Multiple hash lists are then chained 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 impossible matching subscriptions.

H-$Tree$ is theoretically analyzed in terms of the time and space complexities. The time complexity of pre-processing is $O(N_{ind}N_{sub})$, where N_{ind} is the number of indexed attributes and N_{sub} is the number of subscriptions. The time complexity of matching is $O(r_iN_{sub})$, where r_i is index ratio which indicates the index efficiency of an index structure. The space complexity is $O(N_{sub})$. Extensive experiments are also conducted to evaluate the performance of H-$Tree$ and experimental results show that H-$Tree$ outperforms its counterparts to a large degree. In particular, the matching time is faster by three order of magnitude than its counterparts when both the number of subscriptions and the number of constraints are large.

The rest of the paper is organized as follows. Section II introduces the background and the data model. Section III presents the design of H-$Tree$. Section IV analyzes H-$Tree$. Section V illustrates the results of performance evaluation. Section VI discusses related work. Section VII concludes the paper.

II. Background and Model

A. Background of Pub/Sub Systems

A pub/sub system is composed of publishers, subscribers, and a middleware[1]. The middleware is composed of brokers (servers or proxies) that process matching and routing tasks. An example of a pub/sub system is shown in Fig. 1. It can be noted that event matching is performed at all brokers once

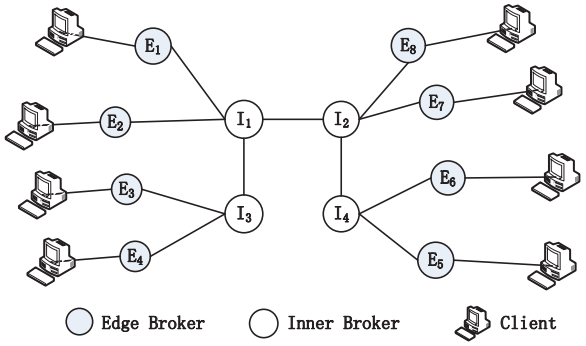

Fig. 1. A distributed pub/sub system where event matching is performed at all brokers.

they receive events. 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 since inner brokers act as dispatchers in the network. When an inner broker receives an event from an interface, event matching is carried out to decide whether the event is forwarded through other interfaces to the next-hop brokers. When the number of received events is greater than an inner broker's processing capacity, the inner broker becomes a performance bottleneck. Therefore, achieving fast matching is critical to large-scale pub/sub systems.

B. Data Model

A $message$ is also called an $event$ in the context of content-based pub/sub systems. An event can be represented as a conjunction of attribute-value pairs. Every attribute name should appear only once in an event. For example, $\{(current, 10.4), (voltage, 223), (power, 2300)\}$ is an event which describes the electrical status of a device. In real applications, each attribute has its value domain which is specified by minimum and maximum values.

Following the naming conventions of $Siena$, a $range\ constraint$ is a 4-tuple of $\{name, lowValue, highValue, type\}$. $Name$ is one of the attribute names appeared in events. The value $type$ can be any of simple or complex data types, such as $integer$, $double\ or\ string$. It is assumed that range constraints are in inclusive form and are normalized to [0,1.0]. A range constraint is equivalent to a conjunction of two simple constraints. For example, the range constraint $\{tem, 10, 20, integer\}$ is equal to $\{tem, \geqslant, 10, integer\} \wedge \{tem, \leqslant, 20, integer\}$. Other forms of constraints can be converted into range constraints. For example, if $power$ is of $integer$ type and has a minimum value of 10 and a maximum value of 100, then a simple constraint $\{power, \leqslant, 50, integer\}$ can be transformed to a range constraint $\{power, 10, 50, integer\}$. A $range\ subscription$ is a conjunction of multiple range constraints. Each subscription is identified by a unique $subID$. In the paper, only range constraints are considered. By analyzing the properties of range constraints, we design an efficient index structure for range subscriptions.

III. Design of H-$Tree$

A. Overview

In order to achieve fast event matching, subscription pre-processing is necessary. It is common to organize subscriptions into an index structure, which is applied equally to H-$Tree$. If an index structure is efficient, then most unrelated items can be filtered out when searching. In order to evaluate the index efficiency of H-$Tree$, index ratio r_i is defined as

$$r_i = \frac{n_r}{n_a}, \tag{1}$$

where n_r is the number of remained subscriptions after filtering and n_a is the number of all subscriptions.

H-$Tree$ is a combination of hash lists and hash chaining. The value domain of each indexed attribute is divided into cells on which a hash list is realized. Multiple hash lists are then chained into a hash tree. When pre-processing a subscription, the subID of the subscription is stored into one bucket. The cellID of each indexed attribute is computed first and the bucketID is calculated according to the cellIDs. When matching an event, a small number of buckets need to be checked. 2 or 3 cellIDs are computed for each indexed attribute and bucketIDs are calculated according to these cellIDs.

The construction of H-$Tree$ is described as follows. First, indexed attributes are selected from the event attributes. Then, the value domain of each indexed attribute is divided into cells on which a hash list is realized. Finally, the hash lists realized on all indexed attributes are chained into a hash tree. The details of domain division and attributes chaining are described in the following subsections.

B. Selection of indexed attributes

Indexed attributes are first selected from the event attributes. The selection has obvious impacts on the performance of H-$Tree$. It is more beneficial to select popular attributes than infrequent ones because a hash tree built on infrequent attributes is more likely to be skewed. Some heuristic rules are given for the selection of indexed attributes. Firstly, popular attributes are prioritized because a hash tree built on popular attributes is less skewed. Secondly, attributes with smaller range widths are taken first. Finally, the appropriate number of indexed attributes is between 5 and 10 because the number of buckets grows exponentially with the number of indexed attributes.

C. Division of value domains

A range constraint is specified by a low value and a high value. Center and width are two indispensable information to describe a range constraint. Given an attribute value, any range constraint is possible to match it. It is not feasible to hash range constraints on one characteristic because the width of a range constraint is unlimited. By putting a limitation on the constraint width, range constraints that need to be checked can be reduced. In reality, the width of range constraints specified by users is usually a small proportion to the whole attribute domain. By restricting the width of range constraints under an upper bound, hashing is realized on the center of range constraints.

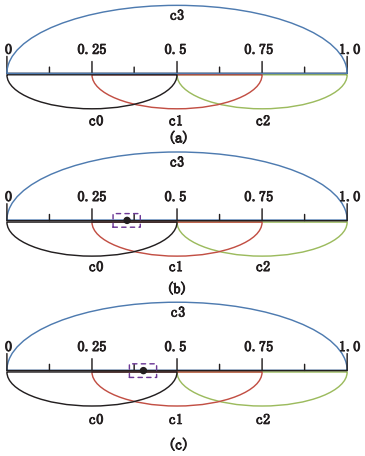

Fig. 2. A novel division of attribute spaces.

Suppose that the width of range constraints is smaller than an upper bound $\mathcal{W}$. (How to relax this bound is discussed later.) After the selection of indexed attributes, the number of cells N_{cel} that each attribute domain is divided into is determined. The ceiling of N_{cel} is limited by $\mathcal{W}$, which is $\lfloor \frac{1}{\mathcal{W}} \rfloor$. By imposing this restriction, it is ensured that the width of cells is not smaller than $\mathcal{W}$. After dividing an attribute domain into N_{cel} segments, two neighboring segments are composed into a $cell$. In this way, two neighboring cells are overlapped with a width of $\frac{1}{N_{cel}}$. The overlapped division of attribute domains complies with the property of range constraints.

The illustration is shown in Fig. 2. The upper bound $\mathcal{W}$ in the example is 0.25, and 4 segments are divided. 3 overlapped cells $\{c0, c1, c2\}$ are composed with their centers at $\{0.25, 0.5, 0.75\}$ respectively, as shown in Fig. 2 (a). The last cell, namely $c3$, is specialized for the case where no range constraint is specified on an attribute. When processing a subscription, the cellID of each indexed attribute is computed. When computing the cellID for indexed attributes, the rules are listed as follows.

(1) If no range constraint is specified on the indexed attribute, the cellID is $c3$.
(2) If the center of the range constraint is located in interval [0, 0.25] or [0.75, 1.0], the cellID is $c0$ or $c2$, respectively.
(3) 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 overlapped cells.

In Fig. 2 (b), 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. 2 (c), 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 $c1$.

The above domain division assumes the uniform distribution of the constraint values, so the attribute domains are evenly divided. It is obvious that this will not work perfectly when the distribution of the constraint values is skewed, which are normal for real-world applications. However, given the distribution function of the constraint values $F(x)$, non-uniform distribution can be converted into the uniform distribution. According to the Probability Integral Transformation theorem [14], 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]. For example, when X is a random variable with a standard normal distribution $N(0, 1)$. Then its CDF is

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

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

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt. \quad (3)$$

As for pareto distribution, its CDF is

$$F(x) = \begin{cases} 1 - (\frac{x_m}{x})^{\alpha} & for\ x \geqslant x_m \\ 0 & for\ x < x_m \end{cases} \quad (4)$$

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

D. Chaining of hash lists

After the overlapped division of attribute domains, the hash lists of all indexed attributes are chained into a hash tree, just like shown in Fig. 3. The chaining of indexed attributes naturally corresponds to the conjunction relationship of constraints in subscriptions. In nature, $H\text{-}Tree$ is a hash table by putting similar subscriptions in the same 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 bucket to store the subscription is easy to calculate. Please note, only the subID of a subscription is stored in the buckets, not the content of the subscription, so the storage consumption is very small. When matching events, 2 or 3 cellIDs is figured out for each indexed attribute. The bucketIDs that need to be checked are computed from these cellIDs. The subID stored in these bucketIDs is used to retrieve the content of subscriptions from the subscription list. Simple matching method is employed to get the matched subscriptions.

This kind of dividing and chaining has many merits. First, an attribute value is at most located in two cells by putting an upper bound on the range width. Our aim is to design an index structure which is capable of filtering out unrelated subscriptions when matching events. When the number of cells divided in each attribute domain is larger than 5, at least 40% subscriptions are filtered out at each level in $H\text{-}Tree$. By chaining multiple hash lists, subscriptions that finally need to be checked are exponentially decreased with the number of chained attributes.

Second, similar subscriptions are hashed into the same bucket based on the center of multiple range constraints. Subscriptions that need to be checked have high probability to match events because most unrelated subscriptions are already filtered out level by level. Therefore, $H\text{-}Tree$ has the ability to return the matched subscriptions earlier than traditional

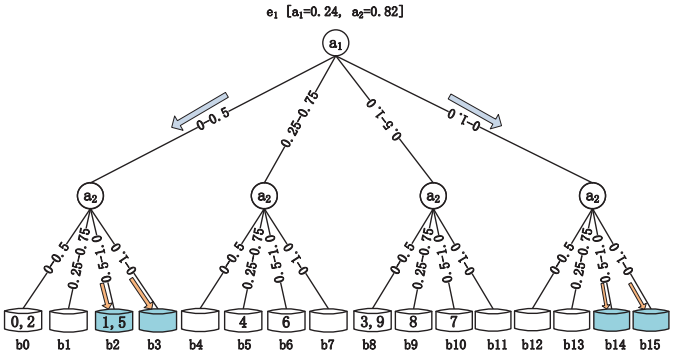

Fig. 3. An illustration of subscription pre-processing and event matching

TABLE I. THE 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	$0.8 \leqslant a_1 \leqslant 0.9 \wedge 0.3 \leqslant a_2 \leqslant 0.4$

matching algorithms. One advantage of this ability is that events are forwarded as soon as possible at inner brokers and the end-to-end latency is reduced in turn.

E. Example

An example is shown in Fig. 3. In the example, there are 2 attributes, a_1 and a_2. Each attribute is divided into 4 cells numbered from 0 to 3. By chaining 2 attributes, there are 16 buckets from $b0$ to $b15$. 10 subscriptions are stored in the system listed in Table I. 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.

IV. ANALYSIS

$H\text{-}Tree$ is analyzed in terms of the time and space complexities. A subscription is called a small-range subscription when all of its range widths are smaller than the upper bound $\mathcal{W}$. Otherwise, a subscription is called a large-range subscription.

A. Analysis for Small-range Subscriptions

1) Pre-processing Time: For a small-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 $O(N_{ind}N_{sub})$, where N_{ind} is the number of indexed attributes and N_{sub} is the number of subscriptions.

2) Matching Time: The matching time has direct relation with the index ratio r_i. In the case of the uniform distribution, all subscriptions are evenly stored in the buckets. The number of buckets is ${N_{cel}}^{N_{ind}}$. When matching events, 2 or 3 cellIDs are figured out for each indexed attribute, so the number of buckets that need to be checked is at most $3^{N_{ind}}$. The index ratio r_i is

$$r_i = \frac{3^{N_{ind}}}{{N_{cel}}^{N_{ind}}} = \frac{3}{N_{cel}}^{N_{ind}}, \tag{5}$$

which is exponentially decreased with the number of indexed attributes because N_{cel} is larger than 3. The time complexity of matching is $O(r_i N_{sub})$. When N_{cel} is 8 and N_{ind} is 8, the value of r_i is 0.00039, which means that only 0.39 ‰ of total subscriptions are remained to be checked after the filtering of unrelated subscriptions.

3) Insertion time: Inserting a small-range subscription involves computing the bucketID and adding the subID of the subscription to the bucket. The time complexity of insertion is $O(N_{ind})$.

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 the average, the number of subIDs in a bucket is $\frac{N_{sub}}{{N_{cel}}^{N_{ind}}}$, so the time complexity of deletion is $O(\frac{N_{sub}}{{N_{cel}}^{N_{ind}}})$.

5) Storage Consumption: The subID of a small-range subscription is stored only once. The space complexity is $O(N_{sub})$.

B. Analysis for Large-range Subscriptions

The upper bound of the range width is addressed by splitting a large-range subscription into multiple small-range subscriptions. One defect of $H\text{-}Tree$ is that the subID of large-range subscriptions is stored in multiple buckets. Several assumptions are made to analyze the time and space complexities for large-range subscriptions, which are listed as follows:

(1) The generation of the event values and constraint values, and the selection of constraint attributes fit the uniform distribution.
(2) The average width of range constraints is w.
(3) "80/20 rule" applies to the distribution of the range width, which means that 80% of the range width of the indexed attributes are smaller than the upper bound and 20% are larger than the upper bound.
(4) The number of cells is N_{cel}.

Under these assumptions, a large-range subscription is split into N_{spl} small-range subscriptions, which is computed by

$$N_{spl} = \lceil \frac{w}{1/N_{cel}} \rceil^{0.2N_{ind}} = \lceil (wN_{cel}) \rceil^{0.2N_{ind}}. \tag{6}$$

1) Pre-processing Time: The pre-processing time is affected when processing large-range subscriptions. The number of small-range subscriptions after splitting is N_{spl}. The time complexity of pre-processing is $O(N_{spl}N_{ind}N_{sub})$.

2) Matching Time: Since the subID of a large-range subscription is stored in multiple buckets, the average number of subIDs in a bucket is increased. When matching events, the index ratio r_i is $\frac{3}{N_{cel}}^{N_{ind}}$. The time complexity of matching is $O(r_i N_{spl} N_{sub})$.

3) Insertion time: Just as discussed above, the complexity of inserting a small-range subscription is $O(N_{ind})$. A large-range subscription is split into N_{spl} small-range subscriptions. The time complexity of inserting a large-range subscription is $O(N_{spl} N_{ind})$

4) Deletion time: When deleting a large-range subscription, the subID is removed from multiple buckets. In each bucket, the subID is searched by testing it against all subIDs. The time complexity of deleting a large-range subscription is $O(\frac{N_{spl} N_{sub}}{N_{cel}^{N_{ind}}})$.

5) Storage consumption: For a large-range subscription, the subID is stored in multiple buckets after splitting. The space complexity is $O(N_{spl} N_{sub})$.

V. Performance Evaluation

Evaluation results are presented in this section. We measure the matching time, insertion time, deletion time, and memory consumption to compare the performance of $H\text{-}Tree$ with its counterparts. All experiments are conducted on a Dell PowerEdge T710 with 8 2GHz cores and 32GB 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.

A. Experimental Settings

The performance of an index structure is influenced by many parameters. In order to extensively evaluate the performance of $H\text{-}Tree$, these parameters are identified as shown in Table II. Experiments are conducted to to observe the impacts of these parameters on the matching performance.

Three methods are compared with $H\text{-}Tree$, namely $Simple$, $Siena$, and $Tama$. $Simple$ utilizes naive matching policy which compares an event with all subscriptions. This policy is optimized in two aspects. First, attributes are numbered, so a range constraint compares quickly with the corresponding attribute value, not needing a loop. Second, when one range constraint in a subscription is not satisfied, matching is turned to the next subscription. $Siena$ is an exact matching algorithm which is referenced as baseline in many literatures [2, 10]. $Tama$ is an approximate matching and forwarding engine which is the fastest matching algorithm so far to the best of our knowledge [13]. The discretization level of $Tama$ is set to 13 in the experiments.

B. Matching Time

The matching time is the most important metric to evaluate a matching algorithm. The matching time is influenced by many parameters. Comprehensive experiments are conducted to observe the impacts of these parameters. 10000 events are sent to measure the matching time per event in each experiment. The average is plotted in figures and the standard deviation is presented in tables. In the experiments, the constraint values, event values and constraint attributes are generated uniformly unless stated clearly.

TABLE II. THE PARAMETERS USED IN THE EXPERIMENTS

Parameter name	Meaning
N_{sub}	the number of subscriptions
N_{att}	the number of attributes
N_{con}	the number of range constraints
N_{cel}	the number of cells
N_{ind}	the number of indexed attributes
$width$	width of range constraints
a	parameter of Zipf distribution

1) Matching time with the number of subscriptions: In general, the matching time is linear with the number of subscriptions. One observation found in the experiment is that $Simple$ is faster than $Siena$ because $Siena$ spends large time on counting the partially matched subscriptions. When the number of constrains in a subscription is large, the partial matching probability of the subscription is high. Suppose that the matching probability of a range constraint is 10% and range constraints are independent. The full matching probability of a subscription composed of 2 range constraints is 1% and the partial matching probability of the subscription is 20%. When the number of range constraints is 10, the probability of full match is 10^{-10} and the probability of partial match is 100%. $H\text{-}Tree$ is 12 times faster than $Tama$ when the number of subscriptions is 5M. The results are shown in Fig. 4 where y-axis represents the matching time in log-scale. The standard deviation of the matching time is given in Table III(a). The matching time of $H\text{-}Tree$ is least fluctuated compared with other three methods. The fluctuation of the matching time is influenced by the distribution of the constraint values.

2) Matching time with the distribution of constraints values: We generate the constraint 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 event values and constraint attributes are generated uniformly. Compared with the uniform distribution, other two distributions may cause $H\text{-}Tree$ skewed. As mentioned in Section III, converting non-uniform distribution into the uniform is beneficial to balance $H\text{-}Tree$. We compare the benefits obtained by distribution conversion. The results are shown in Fig.5. Before distribution conversion, the matching time of $H\text{-}Tree$ under the normal and the pareto distribution is larger than the one under the uniform distribution. The benefit of conversion is obvious. The improved matching performance is 12% and 25% for the normal and the pareto distribution, respectively. The standard deviation of the matching time is given in Table III(b). After distribution conversion, the standard deviation under the normal and the pareto distribution is reduced to the scale under the uniform distribution.

When subscriptions are evenly hashed into the buckets, the distribution of event values has no effects on the matching time. Another experiment is conducted to evaluate the impacts of the distribution of the event values when the constraint values are generated uniformly. There is no difference in the matching time under the uniform, normal and pareto distributions. The results are omitted due to the space limit.

3) Matching time with the number of constraints: We present $H\text{-}Tree$ for large-scale pub/sub systems in terms of both the number of subscriptions and the number of constraints

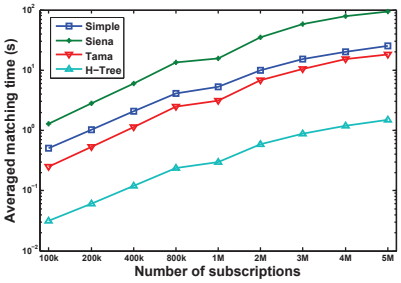

Fig. 4. The matching time with the number of subscriptions, where $N_{att} = 50$, $N_{con} = 20$, $width = 0.1$, $N_{cel} = 8$, $N_{ind} = 8$.

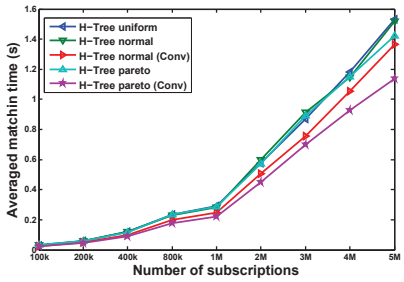

Fig. 5. The matching time under different distribution of constraint values, where $N_{att} = 50$, $N_{con} = 20$, $width = 0.02$, $N_{cel} = 8$, $N_{ind} = 8$.

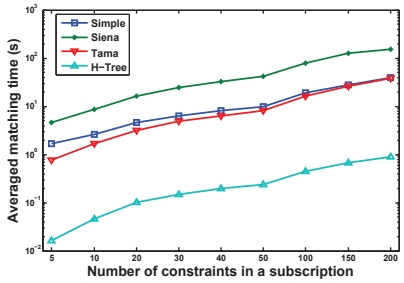

Fig. 6. The matching time with the number of constraints, where $N_{sub} = 1M$, $N_{att} = N_{con} * 2$, $width = 0.1$, $N_{cel} = 8$, $N_{ind} = 8$.

TABLE III.

(a) the standard deviation of fig. 4

	Simple	*Siena*	*Tama*	*HTree*
100k	0.010	0.013	0.010	0.006
200k	0.016	0.025	0.021	0.008
400k	0.082	0.069	0.041	0.014
800k	0.051	0.478	0.097	0.027
1M	0.123	0.198	0.119	0.032
2M	0.165	1.191	0.256	0.064
3M	0.262	1.404	0.409	0.096
4M	0.254	3.538	1.373	0.132
5M	0.265	6.024	0.692	0.164

(b) the standard deviation in fig. 5

	Uni	Nor	Nor(C)	Par	Par(C)
100k	0.006	0.013	0.005	0.015	0.006
200k	0.007	0.024	0.007	0.023	0.008
400k	0.012	0.049	0.011	0.052	0.015
800k	0.021	0.096	0.021	0.106	0.026
1M	0.027	0.120	0.025	0.140	0.034
2M	0.053	0.258	0.051	0.267	0.068
3M	0.078	0.398	0.078	0.426	0.109
4M	0.104	0.523	0.106	0.518	0.135
5M	0.138	0.644	0.144	0.610	0.168

(c) the standard deviation in fig. 6

	Simple	*Siena*	*Tama*	*HTree*
5	0.081	0.133	0.083	0.007
10	0.066	0.220	0.128	0.010
20	0.111	0.614	0.133	0.017
30	0.080	0.633	0.231	0.022
40	0.089	0.512	0.181	0.029
50	0.218	0.633	0.400	0.034
100	0.111	4.038	0.884	0.061
150	0.167	2.918	1.240	0.090
200	0.578	2.714	0.878	0.118

in each subscription. $H\text{-}Tree$ is especially capable of handling matching tasks where the number of range constraints is at the scale of tens or hundreds. The experimental results confirm the advantage of $H\text{-}Tree$, which are shown in Fig. 6 where y-axis represents the matching time in log-scale. On the average, $H\text{-}Tree$ is almost 30 times faster than $Tama$. The standard deviation is presented in Table III(c).

One prominent merit of $H\text{-}Tree$ is that the matching performance is improved with the number of range constraints consisted in subscriptions given the number of attributes. Ratio of the number of constraints to the number of attributes (RCA) is defined as

$$RCA = \frac{N_{con}}{N_{att}}. \tag{7}$$

The impacts of RCA on the matching time are shown in Fig. 7 where y-axis represents the matching time in log-scale. In the experiment, the number of attributes is fixed to 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 $Simple$ because the index ratio of $Tama$ is almost 100%. Second, $H\text{-}Tree$ performs perfectly with larger number of constraints, which can be explained that subscriptions are more evenly hashed into the buckets with higher RCA. The standard deviation of $H\text{-}Tree$ decreases with RCA. The standard deviation is shown is Table IV(a).

4) Matching time with the number of indexed attributes: Index efficiencies are improved with more indexed attributes. This is verified by the experimental results shown in Fig. 8 where y-axis represents the matching time in log-scale. However, more indexed attributes mean more buckets given the number of cells. As shown in the figure, the matching time is reduced at least 29% by adding one indexed attribute. When the number of subscriptions is 5M and the number of indexed attributes is 4, $H\text{-}Tree$ is almost 5 times faster than $Tama$. However, when the number of indexed attributes is 8, $H\text{-}Tree$ is 35 times faster than $Tama$. The standard deviation reduces with the number of indexed attributes, which is shown in Table IV(b). The impacts of the number of cells are similar to the number of indexed attributes. Due to the space limit, the figure and the standard deviation are omitted.

5) Matching time with the distribution of attributes selection: In some cases where the constraint attributes are not uniformly selected from the event 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 $k\%$ is taken by $(100 - k)\%$ of the participants[15]. This phenomenon is called "80/20 rule" which is beneficial to $H\text{-}Tree$ because $H\text{-}Tree$ built on popular attributes is less skewed than on infrequent ones. We simulate the distribution of attributes selection as Zipf. The value of a is changed. Fig. 9 shows the results where y-axis represents the matching time in log-scale. When a is 1.0 and the number of subscriptions is 5M, $H\text{-}Tree$ is at most 2190 times faster than $Tama$. The standard deviation of $H\text{-}Tree$ decreases with the value of a which verifies our discussion. The standard deviation is shown in Table IV(c).

6) Matching time with range width: Range width has no impacts on $Simple$ and $Siena$. As for $Tama$, large range width means more matching time because the subID of a subscription is stored in more buckets. When the width of range constraints is smaller than the upper bound $\mathcal{W}$, the performance of $H\text{-}Tree$ is not impacted. Two experiments are conducted to evaluate the impacts of the range width on the matching time. The first experiment tests 1M subscriptions with different fixed range width smaller than $\mathcal{W}$. The results are plotted in Fig. 10. Just as discussed above, $Simepe$, $Siena$ and $H\text{-}Tree$ are not influenced by the width. However, the matching time of $Tama$ is increased linearly with the width. When the width is 0.1, $H\text{-}Tree$ is 44, 167 and 34 times faster than $Simple$, $Siena$ and $Tama$, respectively. The matching time of $H\text{-}Tree$ is least fluctuated compared with its counterparts, which is shown in Table V(a).

The second experiment tests random range width. The

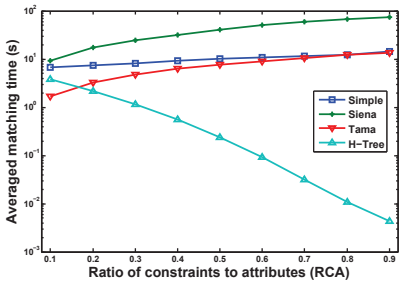

Fig. 7. The matching time with RCA, where $N_{sub} = 1M$, $N_{att} = 100$, $width = 0.1$, $N_{cel} = 8$, $N_{ind} = 8$.

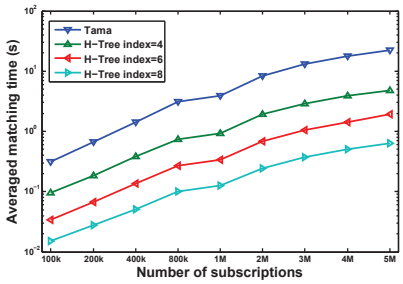

Fig. 8. The matching time with the number of indexed attributes, where $N_{att} = 50$, $N_{con} = 25$, $width = 0.1$, $N_{cel} = 8$.

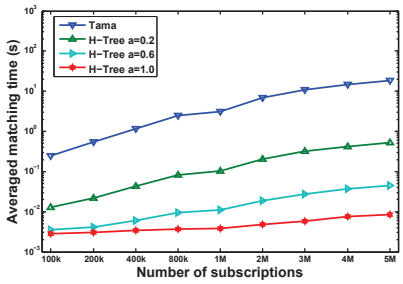

Fig. 9. The matching time with the distribution of attributes selection, where $N_{att} = 50$, $N_{con} = 20$, $width = 0.1$, $N_{cel} = 8$, $N_{ind} = 8$.

TABLE IV.

(a) the standard deviation in fig. 7

	Simple	*Siena*	*Tama*	*HTree*
0.1	0.072	0.084	0.047	0.078
0.2	0.088	0.206	0.090	0.092
0.3	0.120	0.534	0.132	0.078
0.4	0.175	0.732	0.185	0.058
0.5	0.248	1.205	0.208	0.034
0.6	0.138	0.507	0.254	0.019
0.7	0.130	2.682	0.301	0.009
0.8	0.107	2.167	0.375	0.006
0.9	0.835	0.643	0.395	0.005

(b) the standard deviation in fig. 8

	Tama	ind=4	ind=6	ind=8
100k	0.012	0.012	0.006	0.006
200k	0.026	0.019	0.009	0.007
400k	0.050	0.038	0.018	0.009
800k	0.118	0.072	0.033	0.017
1M	0.153	0.092	0.043	0.020
2M	0.312	0.190	0.082	0.038
3M	0.500	0.288	0.136	0.058
4M	0.625	0.392	0.180	0.081
5M	0.845	0.461	0.243	0.097

(c) the standard deviation in fig. 9

	Tama	HT a=0.2	HT a=0.6	HT a=1.0
100k	0.010	0.005	0.005	0.005
200k	0.023	0.006	0.005	0.005
400k	0.041	0.009	0.005	0.005
800k	0.093	0.015	0.005	0.005
1M	0.121	0.017	0.006	0.005
2M	0.253	0.033	0.008	0.005
3M	0.471	0.051	0.011	0.006
4M	0.529	0.068	0.013	0.006
5M	0.683	0.085	0.016	0.006

$lowValue$ and $highValue$ of each range constraint are randomly generated in the experiment. Large-range subscriptions are split into multiple small-range subscriptions for $H\text{-}Tree$. The results are shown in Fig. 11. The matching time of $H\text{-}Tree$ is slightly affected by large-range subscriptions because the subID of large-range subscriptions is stored in multiple buckets. The performance of $Tama$ degrades quickly with large number of wide range constraints, even slower than $Simple$. The fluctuation of $H\text{-}Tree$ is magnified because large-range subscriptions are split and stored in multiple buckets, which causes the size of buckets uneven, middle buckets with more subscriptions. The standard deviation is shown in Table V(b).

C. Maintenance Cost

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

1) Insertion time: For $Simple$, inserting a subscription just means adding it to the list of subscriptions. $Siena$, $Tama$ and $H\text{-}Tree$ have specialized index structures. We measure the time when inserting small-range subscriptions into the index structures for these three algorithms. 10 runs of experiment are conducted. The averaged insertion time is presented. The results are shown in Fig. 12. $H\text{-}Tree$ spends a little more time, less than 10%, than $Simple$. However, the insertion time of $Tama$ is almost 2 times of $Simple$. The standard deviation is small which is omitted.

The performance of $H\text{-}Tree$ degrades when inserting large-range subscriptions. We measure the time when inserting subscriptions with random range width. The $lowValue$ and $highValue$ of each range constraint are randomly generated. The results are shown in Fig. 13 where y-axis represents the matching time in log-scale. $H\text{-}Tree$ spends 60% more time than $Simple$ when inserting 1M subscriptions. The matching time of $H\text{-}Tree$ is 200% faster than $Simple$. Compared with the results of fixed width, the insertion time of $Simple$, $Siena$ and $H\text{-}Tree$ changes little. However, the insertion time of $Tama$ fluctuates greatly. The standard deviation is listed in Table V(c).

2) Deletion time: In $Siena$, the number of buckets to store constraints is equal to the number of event attributes. When the number of subscriptions is large, the size of the buckets is also large. Deleting a subscription is very costly. As for $Tama$, the subID of a subscription is stored in multiple buckets. Deletion is also time-consuming as shown in Fig. 11 in [13]. Therefore, the deletion time is not measured for $Siena$ and $Tama$. We measure the deletion time per event by deleting 1000 subscriptions from different number of subscriptions for $H\text{-}Tree$. The number of indexed attributes has impacts on the deletion time. $H\text{-}Tree$ is evaluated under different value of N_{ind}. The results are shown in Fig. 14. Given the number of subscriptions, the size of the buckets decreases with more indexed attributes. The deletion time of $H\text{-}Tree$ is very stable with little fluctuation. The standard deviation is omitted due to the space limit.

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, 2 *double* for values. A subscription needs 1 *integer* to store the subID. There is no additional storage requirement for $Simple$. As for $Siena$, the number of range constraints in the subscription should be stored for event matching. Additional storage for $Siena$ is 2 bytes for each subscription. 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 at least stored 9 times which consumes 36 bytes. Just like $Siena$, 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, $H\text{-}Tree$ just stores the subID of a subscription in one bucket. Additional storage for $H\text{-}Tree$ is 4 bytes for each subscription. The memory consumption with the number of subscriptions is plotted in Fig. 15. As shown in the figure, $H\text{-}Tree$ consumes a little more memory than $Simple$, but much smaller than $Tama$.

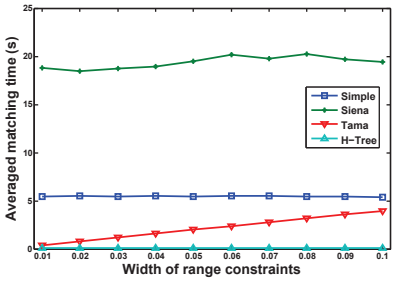

Fig. 10. The matching time with fixed range width, where $N_{sub} = 1M$, $N_{att} = 50$, $N_{con} = 25$, $N_{cel} = 8$, $N_{ind} = 8$.

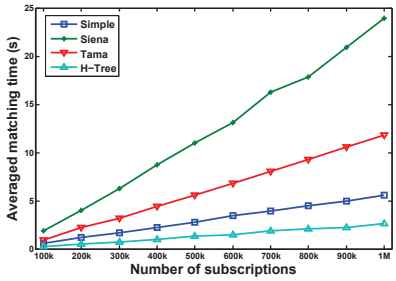

Fig. 11. The matching time with random range width, where $N_{att} = 50$, $N_{con} = 25$, $N_{cel} = 8$, $N_{ind} = 8$.

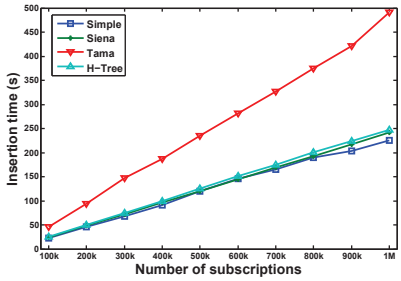

Fig. 12. The insertion time with fixed width subscriptions, where $N_{att} = 50$, $N_{con} = 20$, $width = 0.1$, $N_{cel} = 8$, $N_{ind} = 8$.

TABLE V.

(a) the standard deviation in fig. 10

	Simple	*Siena*	*Tama*	*HTree*
0.01	0.074	0.639	0.008	0.017
0.02	0.101	0.191	0.015	0.016
0.03	0.086	0.180	0.026	0.017
0.04	0.074	0.402	0.042	0.017
0.05	0.077	0.654	0.054	0.018
0.06	0.109	0.415	0.075	0.017
0.07	0.074	0.700	0.093	0.018
0.08	0.149	0.743	0.108	0.018
0.09	0.062	0.605	0.129	0.019
0.1	0.093	0.197	0.146	0.019

(b) the standard deviation in fig. 11

	Simple	*Siena*	*Tama*	*HTree*
100k	0.010	0.028	0.062	0.147
200k	0.063	0.080	0.149	0.299
300k	0.043	0.114	0.200	0.426
400k	0.056	0.172	0.287	0.587
500k	0.044	0.280	0.338	0.772
600k	0.087	0.232	0.421	0.829
700k	0.067	0.258	0.520	1.075
800k	0.070	0.355	0.656	1.231
900k	0.076	0.751	0.677	1.355
1M	0.083	0.884	0.728	1.580

(c) the standard deviation in fig. 13

	Simple	*Siena*	*Tama*	*HTree*
100k	0.055	0.132	554.285	0.441
200k	0.202	0.168	632.224	0.442
300k	0.203	0.148	900.780	0.441
400k	0.227	0.282	811.669	0.593
500k	0.566	0.435	1095.445	0.709
600k	0.470	0.572	1194.138	0.594
700k	0.426	0.297	1270.297	0.982
800k	0.408	1.361	1593.709	1.141
900k	1.073	0.391	1247.413	1.132
1M	0.589	0.676	1241.538	1.501

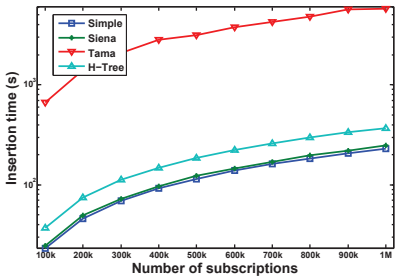

Fig. 13. The insertion time with random width subscriptions, where $N_{att} = 50$, $N_{con} = 20$, $width = random$, $N_{cel} = 8$, $N_{ind} = 8$.

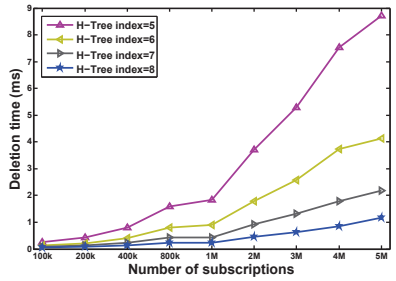

Fig. 14. The deletion time per subscription, where $N_{att} = 100$, $N_{con} = 30$, $width = 0.05$, $N_{cel} = 10$.

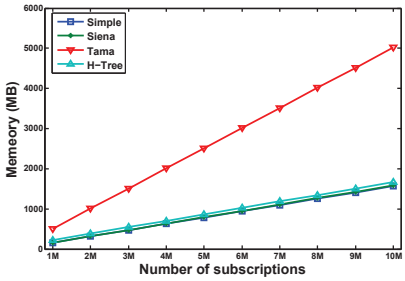

Fig. 15. The memory consumption counted in memory space needed to store subscriptions and subIDs.

VI. Related Work

Event matching is a critical component of large-scale content-based pub/sub systems. Many methods have been proposed to improve matching efficiencies. These methods can be classified into three categories.

A. Counting Algorithm based

Different index structures have been proposed to improve matching efficiencies based on counting algorithms [10, 11, 13, 16, 17]. The matching procedure of these methods is separated into two phases. In the first phrase, partially matched subscriptions are computed. Then fully matched subscriptions are returned by utilizing counting algorithms. [10] and [13] are two representative ones. In [18], bloom filters are used to store matched primitive constraints. One disadvantage of these methods is that a subscription may be returned multiple times as partially matched subscription. The time complexity of counting algorithms is linear with the number of partially matched subscriptions.

B. Reducing Routing Table Size

The 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 [19], subscriptions are summarized to reduce the routing table size by using imperfect merging. Similar approaches are subscription subsumption and covering, such as [20] [21] [22]. The order of subscriptions, cover relation, matching history, and routing destination are considered simultaneously to provide efficient event matching in [23]. These methods are orthogonal to our proposed method. However, the time complexity of subscription subsumption is relatively high and one drawback of imperfect merging is the waste of bandwidth due to the existence of false positives.

C. Tree-like structures

Range subscriptions are composed of range constraints. Therefore, subscriptions can be viewed geographically as rectangular regions and publications are viewed as points in the multi-dimensional space composed of event attributes. Event matching problem equals to point enclosure problem, which tries to check whether a point is contained by some rectangular regions. Some traditional tree-like structures can be used to index subscriptions, including R-Tree [24], and Interval Binary Search Tree (IBS-Tree) [25]. In [8], subscriptions were pre-processed into a matching tree which had the matching complexity of $O(N^{1-\lambda})$ at the cost of pre-processing time

complexity $O(NK)$ and space complexity $O(NK)$, where N is the number of subscriptions and K is the number of attributes. Binary Decision Diagrams were utilized as a filtering engine to speed up event matching in [9]. The performance of these methods degrades when the number of subscriptions and the number of constraints are both large. On the contrary, $H\text{-}Tree$ handles it more efficiently.

VII. Conclusion

In this paper, we present $H\text{-}Tree$, an efficient index structure for event matching in large-scale content-based publish/subscribe systems. The basic idea behind $H\text{-}Tree$ is that matching efficiencies can be improved when the search space is substantially reduced. $H\text{-}Tree$ is a hash table in nature which is a combination of hash lists and hash chaining. The novelty is that hash lists are realized on the cells which are overlapped. Extensive experiments are conducted to evaluate the performance of $H\text{-}Tree$ and experimental results show that $H\text{-}Tree$ outperforms its counterparts to a large degree, especially in the case where the number of subscriptions and the number of constraints are both large.

Acknowledgment

This work is partially supported by China National Science Foundation (Granted Number 61073021, 61272438,61170238, 60903190, 61027009), Research Funds of Science and Technology Commission of Shanghai Municipality (Granted Number 11511500102, 12511502704), Cross Research Fund of Biomedical Engineering of Shanghai Jiaotong University (YG2011MS38), Doctoral Fund of Ministry of Education of China (Granted Number 20100073120021), National 863 Program (Granted Number 2009AA012201, 2011AA010500), Singapore NRF (CREATE E2S2).

References

[1] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *Computing Surveys (CSUR)*, 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," *TOCS*, vol. 19, no. 3, pp. 332–383, 2001.

[3] P. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *ICDCS Workshops*. IEEE, 2002, pp. 611–618.

[4] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha, "Efficient matching for content-based publish/subscribe systems," in *Proc. CoopIS*, 2000.

[5] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," *Networked Group Communication*, 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," *Software Engineering, IEEE Transactions on*, vol. 27, no. 9, pp. 827–850, 2001.

[7] A. Cheung and H. Jacobsen, "Load balancing content-based publish/subscribe systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 9, 2010.

[8] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching events in a content-based subscription system," in *PODC*. ACM, 1999, pp. 53–61.

[9] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," in *ICSE*. IEEE, 2001, pp. 443–452.

[10] A. Carzaniga and A. Wolf, "Forwarding in a content-based network," in *SIGCOMM*. ACM, 2003, pp. 163–174.

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

[12] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien, "Efficiently evaluating complex boolean expressions," in *SIGMOD*. ACM, 2010, pp. 3–14.

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

[14] R. Fisher, *Statistical methods for research workers*. Genesis Publishing Pvt Ltd, 1925.

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

[16] H. Jafarpour, S. Mehrotra, N. Venkatasubramanian, and M. Montanari, "Mics: an efficient content space representation model for publish/subscribe systems," in *DEBS*. ACM, 2009, p. 7.

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

[18] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in *DEBS*. ACM, 2008, pp. 71–81.

[19] P. Triantafillou and A. Economides, "Subscription summarization: A new paradigm for efficient publish/subscribe systems," in *ICDCS*. IEEE, 2004, pp. 562–571.

[20] 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 *ICDCS*. IEEE, 2005, pp. 447–457.

[21] A. Ouksel, O. Jurca, I. Podnar, and K. Aberer, "Efficient probabilistic subsumption checking for content-based publish/subscribe systems," *Middleware 2006*, pp. 121–140, 2006.

[22] K. Jayaram and P. Eugster, "Split and subsume: Subscription normalization for effective content-based messaging," in *ICDCS*. IEEE, 2011, pp. 824–835.

[23] X. Guo, J. Wei, and D. Han, "Efficient event matching in publish/subscribe: based on routing destination and matching history," in *NAS*. IEEE, 2008, pp. 129–136.

[24] A. Guttman, *R-trees: a dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[25] E. Hanson, M. Chaabouni, C. Kim, and Y. Wang, "A predicate matching algorithm for database rule systems," *SIGMOD*, vol. 19, no. 2, pp. 271–280, 1990.