PUBSUB: An Efficient Publish/Subscribe System *

Tania Banerjee Mishra, Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{tmishra, sahni}@cise.ufl.edu

ABSTRACT

PUBSUB is a versatile, efficient, and scalable publish/subscribe system. This paper describes the architecture of PUBSUB together with some of its current capabilities. A version of PUBSUB optimized for event processing was benchmarked against the publish/subscribe systems BE-Tree and Siena, which also are optimized for event processing. PUBSUB processes events faster than Siena and BE-tree. On our tests, the speedup of the fastest version of PUBSUB relative to Siena 98% on an average. The speedup range relative to BE-Tree was from 1.23 to 1.48 and averaged 1.36 on the uniform tests and PUBSUB was comparable to BE-tree on the Zipf tests. The faster times in PUBSUB were a result of very efficient data structures used in PUBSUB to store the subscriptions, and the fast matching algorithms developed to match events to subscriptions.

Keywords

Content based publish/subscribe, Boolean expressions, efficient subscription matching

1. INTRODUCTION

A publish/subscribe (pub/sub) system maintains a database of subscriptions, where each subscription is a Boolean expression. For example, each subscription in the pub/sub system of a diverse online vendor may describe the conditions under which a customer may purchase a product. A customer interested in acquiring a camera may post his/her requirement as a subscription to the vendor's pub/sub system by providing the Boolean expression:

$item =$ "$camera$" $\wedge\, price < \$300 \wedge$
$manufacturer \in \{Sony, Nikon, Panasonic\} \wedge zoom > 4\times$

This subscription uses four attributes of a product, namely, $item$, $price$, $manufacturer$ and $zoom$. An attribute is also referred to as a dimension. A predicate consists of an attribute, an operator and attribute value(s). For each permissible value of an attribute, the predicate evaluates to true or false. In the above example, $price < \$300$ is a predicate that is true whenever the attribute $price$ has a value below \$300 and false otherwise. A subscription is the conjunction of predicates. Our example subscription is the conjunction of 4 predicates. An event specifies the values of some attributes. For example the availability of a new \$199 5x zoom camera from "Sony" or a price change in an existing 5x zoom "Sony" camera to \$199 may be specified by the event:

$item =$ "$camera$" $\wedge color =$ "red" $\wedge weight = 8oz \wedge pixels = 14M \wedge price = \$199 \wedge manufacturer =$ "$Sony$" $\wedge zoom = 5\times$

The above event matches the example subscription as all 4 predicates in the subscription evaluate to true when the attributes used in the subscription are assigned the values specified in the event. The example subscription, however, is not matched by the following events:

$item =$ "$camera$" $\wedge\, pixels = 14M \wedge price =$
$\$399 \wedge manufacturer =$ "$Sony$" $\wedge\, zoom = 5\times$

$item =$ "$camera$" $\wedge price = \$129 \wedge manufacturer =$ "$Sony$"

The first of these events fails to match the subscription because the price is too high and the second fails because it does not specify the value of an attribute (zoom) that occurs in the subscription.

When an event occurs, the pub/sub system reports all subscriptions in its database that are matched (or satisfied by the event). Customers who posted these matching subscriptions may then be notified.

Pub/sub systems are used in diverse applications with varied performance requirements. For example, in some applications events occur at a much higher rate than the posting/removal of subscriptions while in other applications the subscription rate may be much higher than the event rate and in yet other applications the two rates may be comparable. Optimal performance in each of these scenarios may result from deploying a different data structure for the subscriptions or a different tuning of the same structure. Many commercial applications of pub/sub systems have thousands of attributes and millions of subscriptions. So, scalability in terms of number of attributes and number of subscriptions is critical.

In this paper, we describe the architecture of PUBSUB, which is a versatile and scalable pub/sub system that may be tuned to provide high performance for diverse application environments. PUBSUB is versatile because its architecture supports a variety of predicate types (e.g., ranges, regular expressions, string relations) as well as a heterogeneous collection of data structures for the representation of subscriptions in order to achieve high throughput. The performance

*This research was supported, in part, by the US Air Force Research Laboratory, under grant FA8750-11-1-0245.

of a version of PUBSUB that was tuned for applications in which events occur far more frequently than subscription posting/deletion (hence, high-speed event processing coupled with reasonable support for subscription posting/deletion is required) is compared with the performance of the pub/sub systems BE-Tree [1] developed by Sadoghi and Jacobsen and Siena [2, 3], developed by Carzaniga and Wolf. Both of these benchmark systems also are tuned for the same application environment. Our extensive experiments show that PUBSUB processes events up to 95% faster than does BE-Tree and up to 97% faster than Siena.

We have organized the paper as follows. Section 2 describes the related work in this area. We present the PUBSUB architecture in Section 3 and the results of extensive experimental evaluation in Section 4. We conclude in Section 5.

2. RELATED WORK

The problem of rapidly evaluating a large number of predicates against specified events has been studied extensively in the literature. Yan and Garcia-Molina proposed the use of indexes to speed the evaluation of a collection of Boolean expressions and developed SIFT [10], which is a system based on indexing. Later various researchers proposed decision trees and index structures for this problem. The proposed approaches can be divided into two main categories. The first category is counting based while the second category is based on partitioning subscriptions into subsets. Counting based pub/sub systems build an inverted index structure from the subscriptions and minimize the number of predicate evaluations while partitioning-based systems minimize evaluations by recursively eliminating the subscriptions that cannot be satisfied. The propagation algorithm proposed by Fabret et al. [8], the matching algorithm proposed by Carzaniga et al. [2, 3], and the inverted list construction by Whang et al. [9] all result in pub/sub systems that are counting based. Gryphon, developed by Aguilera et al. [7] and BE-tree [1] developed by Sadoghi and Jacobsen are examples of partitioning-based systems. Our pub/sub system, PUBSUB, also is partitioning based.

BE-tree [1] partitions subscriptions defined on a high dimensional space using two phase space cutting technique, space partitioning and space clustering, to group the expressions with respect to the range of values for the various attributes. Experimental results reported in [1] indicate that the BE-tree outperforms state-of-the-art pub/sub systems such as SCAN [5], SIFT [10], Propagation [8], Gryphon [7], and k-index [9]. BE-Tree, however, is limited to attributes whose values are discrete and for which the range in discrete attribute values is pre-specified. So, BE-tree is unable to cope with real-valued attributes, string-valued attributes, and discrete-valued attributes with unknown range. Additionally, BE-tree employs a clustering policy that is ineffective when many subscriptions have a range predicate such as $low \leq a_i \leq high$, where a_i is an attribute and the clustering criterion p that is used lies between low and $high$. In this case, all such subscriptions fall into the same cluster and event processing is considerably slowed as shown in our experiments of Section 4.

Siena [2, 3] is a pub/sub system that uses a counting algorithm to find matching subscriptions. It maintains an index of attribute names and types. This index is implemented using ternary search tries. Unlike BE-Tree, Siena is not limited to discrete valued attributes from a pre-specified finite domain. Further, Siena is able to work with attributes of type $string$ and supports operators such as prefix, suffix, and substring on this datatype. Siena, however, does not support incremental updates (i.e., subscription posting and deletion) and so updates must be done in batch mode. Although the present implementation of PUBSUB does not support the string datatype, its architecture is sufficiently versatile to accommodate this datatype with the inclusion of additional data structures as described in Section 3.

3. PUBSUB

Section 3.1 describes how PUBSUB organizes its database of subscriptions. In Section 3.2 the data structures and algorithms used in PUBSUB are presented.

3.1 Database Organization

Figure 1 gives the organization of the subscription database used in PUBSUB. This database comprises a collection of level-1 attribute structures $A_1, \ldots, A_m$, where m is the number of attributes. We assume that the allowable attributes have been numbered 1 through m and that the attributes in a subscription are ordered using this numbering of attributes. The attribute structure A_i stores all subscriptions that include a predicate on attribute i but not on any attribute $j < i$. We say that the attribute i is associated with the structure A_i. With our assumptions on attribute ordering within subscriptions, A_i contains all subscriptions whose first attribute is i. In practice, many of the A_is will be empty and only non-empty attribute structures are stored in PUBSUB.

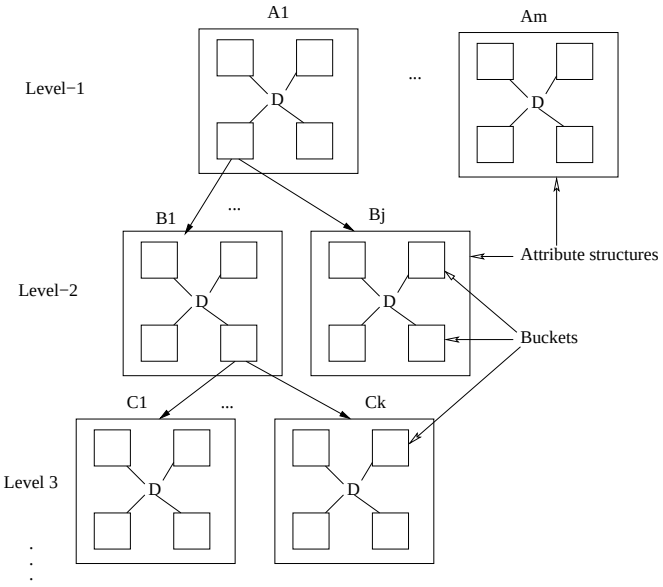

Figure 1: Organization of PUBSUB subscription database

A level-k, $k > 0$, attribute structure A_i comprises 0 or more buckets that contain subscriptions. The distribution of subscriptions across these buckets is determined by the attribute i predicates in these subscriptions and the data structure D used to keep track of the buckets. The data structure D, when given a value v_i for attribute i, is able to efficiently locate the buckets that contain all subscriptions (and possibly others) whose predicate on attribute i is satisfied by v_i. Different attribute structures may use different data structures D to keep track of their buckets. Individual buckets of a level-k attribute structure may have higher level (i.e., larger k) attribute structures associated with them. The attribute associated with a level-k attribute structure is the kth attribute of the subscriptions stored in that structure. For uniformity, level-1 attribute structures are associated with a header bucket that is always empty.

To provide a better understanding of the organization of the subscription database, we describe how events are processed as well as

Algorithm: search(Event e, Bucket b)
Input:
e: event having attributes $e_1, e_2, ..., e_j$
b: current bucket, initially the header bucket
Output:
list of matching subscriptions

```
1:    foreach e_i, 1 ≤ i ≤ j
2:        // A_{e_i} is the attribute structure for e_i associated with b
3:        if (A_{e_i} exists) then
4:            B = buckets in A_{e_i} determined by D and
5:            e_i to possibly have matching subscriptions
6:            foreach b ∈ B
7:                add matching subscriptions in b to output list
10:               search (e, b);
11:           endfor
12:       endif
13:   endfor
```

Figure 2: Search algorithm

how subscriptions are posted and deleted.

Figure 2 gives a high level description of the algorithm to process an event. To search for all subscriptions that match an event that specifies a value for the attributes $e_1 < e_2 < ... < e_j$, we search the level-1 attribute structures A_{e_i}, $1 \leq i \leq j$. Note that the remaining attribute structures contain subscriptions that have at least one attribute (i.e, the first attribute) whose value is not specified by the event and so these subscriptions are not matched by the event. To search A_{e_i} for matching subscriptions, we use the associated data structure D to locate the buckets that may possibly contain matching subscriptions. The subscriptions stored in these buckets are examined to determine those that match the event. Additionally, level-2 attribute structures associated with these buckets and whose associated attribute has a value specified in the event (i.e., the associated attribute is one of the e_is) are recursively searched for matching subscriptions. Note that only those attribute structures (regardless of level) whose associated attribute is one of the e_is may be examined when processing an event; the D structures determine which of these are actually examined.

A high level description of the algorithm to post/insert a subscription is given in Figure 3. Using the attributes in the subscription, the attributes associated with attribute structures, and the D structures, we follow a path that begins at the level-1 attribute structure for the first attribute in the subscription, goes to the appropriate level-2 structure for the second attribute, and so on. If no non-empty attribute structure is encountered, then a new level-1 attribute structure with a single bucket is created for this subscription. The attribute associated with this newly created structure is the first attribute of the new subscription. If non-empty attribute structures are encountered, let k be the lowest level at which this happens and let Z_i be the attribute structure encountered at this level. To insert into a level-k attribute structure Z_i, the data structure D for this structure is used to determine the appropriate bucket b' of Z_i for insertion. If this bucket is full its subscriptions along with the new subscription are split into 2 or more buckets in accordance with the data structure D. In case such a split is not possible (this happens when D is unable to distinguish among the attribute i predicates of the subscriptions in the bucket), the next attribute in the new subscription is used to create a new attribute structure that includes the new subscription and all subscriptions in the full bucket that have a predicate on this attribute. When the new subscription doesn't have a next attribute, we use instead a subscription in the full bucket that has a next attribute. When no subscription has a next attribute we expand the full bucket beyond its designed maximum capacity.

To delete a subscription, we use a procedure that is the inverse

Algorithm: insert(Subscription s, Attribute s_k, Bucket b)
Input:
s: subscription with ordered attributes $s_1 < s_2 < ... < s_j$
s_k: current attribute, initially s_1
b: current bucket, initially the header

```
1:  if (b has no associated attribute structure Z_i for s_k) then
2:      create an attribute structure Z_i for s_k associated with b
3:      move subscriptions (if any), with attribute s_k in b, to Z_i
4:          in accordance with data structure D.
5:      add s in Z_i in accordance with data structure D.
6:  else
7:      Use D and predicate on s_k to find an
8:          appropriate bucket b' in Z_i for s.
9:      if b' has space then
10:         add s to b'
11:     else
12:         split b' and s into 2 or more buckets as per D.
13:         if such a split is not possiblethen
14:             if (k < j) then
15:                 insert(s, s_{k+1}, b');
16:             else
17:                 if there is a subscription s' in b' with a
18:                 k + 1st attribute a' then
19:                     replace s' by s in b';
20:                     insert(s', a', b');
21:                 else
22:                     expand b' to include s
23:                 endif
24:             endif
25:         endif
26:     endif
27: endif
```

Figure 3: Insert algorithm

of that used to insert a subscription.

3.2 PUBSUB Data Structures

3.2.1 Global Hash Table

A single global hash table is used to keep track of all attribute structures regardless of their level and which bucket they may be associated with. The use of a hash table enables faster branching to a next level bucket than when each bucket stores links to next level buckets. The hash key for an attribute structure Z_i associated with bucket b is the pair (b, i). Each Z_i is kept track of using some characteristic of Z_i such as the header (if any) of the data structure D used in Z_i.

3.2.2 Bucket

A bucket is used to store subscriptions. The organization of a bucket is application dependent and we describe exemplar organizations for small and large buckets. Small buckets store few subscriptions, while large ones may store over a thousand subscriptions. Small buckets are useful in applications where the rate at which subscriptions are posted/deleted is high while large ones are useful when we are concerned primarily with the time to process an event.

Subscriptions in a small bucket may be stored as an unordered list. Subscriptions in a large bucket are sorted on the first attribute not associated with the attribute structures on the path from the header to the current bucket. Each group of subscriptions with the same first unused attribute is further sorted based on the predicates of this common attribute. For example, consider a subscription that has a predicate $0 \leq a \leq 10$. Then, the predicate range of attribute a is [0, 10]. Subscriptions in a group are sorted by the starting point of the predicate range for the common attribute.

Figure 4 describes the algorithm to find matching subscriptions

```
Algorithm: Bucket::match(Event e)
Input:
e: event
Output:
matching subscriptions

1:      j = 0;
2:      for i // iterate over the subscription groups, increment i by 2
3:          attr = group[i];
4:          groupEndIndex = group[i+1];
5:          if (attr exists in e) then
6:              endIndex = binarySearch(attr, j, groupEndIndex);
7:              for j up to endIndex, incremented by 1
8:                  match jth subscription in bucket with event
9:                  if (matched) then
10:                     append jth subscription to output list
11:                 endif
12:             endfor
13:         else
14:             j = groupEndIndex + 1;
15:         endif
16:     endfor
```

Figure 4: Search algorithm for a large bucket

in a large bucket. In this algorithm, we check if the common attribute, which is the first unused attribute for a group of subscriptions in a bucket, is present in the event (line 5). If the common attribute is not present in the event, then the whole group of subscriptions is skipped (line 14). If the common attribute is present, we match subscriptions from the beginning of the group up to a certain subscription given by endIndex (lines 6-7) in the group, thereby skipping the rest of the subscriptions (from endIndex+1 up to groupEndIndex) in that group.

The processed subscriptions have the start points of predicate ranges to the left of the event value, whereas those that are skipped have their start points to the right, which completely eliminates the possibility that the event value will be included in the predicate ranges of the skipped subscriptions. Figure 5 shows a group of 5 predicate ranges and an event value corresponding to the common attribute. The three subscriptions with predicate ranges marked as 1, 2, 3 are matched with the event, whereas those with ranges marked as 4 and 5 are skipped, since the start point of ranges 4 and 5 are to the right of the event value.

In the following, we use the term *bucket size* to mean the maximum number of subscriptions permitted in a bucket. The actual size (both the number of subscriptions currently in a bucket as well as the number of subscription slots presently available in the bucket; when all subscription slots are occupied, an implementation may expand the bucket using a technique such as array doubling [11]) of a bucket may vary dynamically.

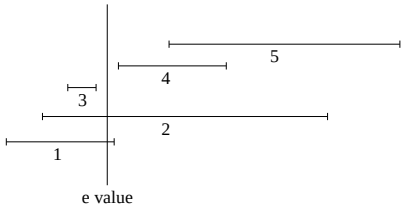

Figure 5: Predicate ranges are ordered by starting points

3.2.3 D Structures

Priority Search Trees [6], Interval Trees [11], Suffix Trees [18], and Aho-Corasick Tries [17] are all examples of structures that can be used for D, depending on the type of attribute being partitioned by D and the operators being supported in the predicates on this attribute. For example, priority search trees and interval trees could be good choices for attributes whose predicates specify a range of values while suffix trees could be good choices for string attributes whose predicates use the substring and suffix operators. Our current implementation of PUBSUB includes Red-Black Priority Search Trees (RBPST) and Interval Trees (IT). These data structures are well suited to determine which of a set of range predicates are satisfied by a specified attribute value. The current implementation of PUBSUB is readily extended to support additional attribute data structures such as red-black trees for exact match attributes and suffix trees and Aho-Corasick tries for string attributes.

An RBPST helps us perform exact-match searches, inserts, and deletes in $O(\log n)$ time and rectangle searches in $O(s + \log n)$ time, where n is the number of points in the RBPST. RBPSTs place no restriction on the domain or the keys.

With an IT, a point v in a range $[L, R)$ may be found in $O(s + logK)$ time, where K is the cardinality of the range $[L, R)$.

3.3 Comparison with BE-Tree

BE-Tree [1] and PUBSUB have many similarities. For example both use clustering on a set of subscriptions that have a common attribute. This is a standard approach for multidimensional data with common attributes and has been used earlier in range trees [20] and multidimensional tries [19], for example. Like BE-Tree, both range trees and multidimensional tries use the same clustering strategy at all levels and for all attributes (range trees use the median attribute value while multidimensional tries use a bit of the attribute to cluster). PUBSUB, on the other hand, allows for a heterogeneous selection of clustering strategies (i.e., the data structure D). Both BE-Tree and PUBSUB partition a set of subscriptions into subsets that have a common attribute so that clustering may be applied to these subsets. BE-Tree selects the partitioning attribute by analyzing the subscriptions in the bucket to be partitioned while PUBSUB does this using a pre-specified attribute ordering.

Besides superior performance (see Section 4), PUBSUB offers the following advantages relative to BE-Tree:

1. BE-Tree uses the same clustering strategy for all attributes resulting in a homogeneous system. PUBSUB, which is a heterogeneous system, offers a variety of data structures to keep track of the buckets in an attribute structure enabling the user to select data structures best suited for each attribute.

2. The clustering strategy employed in BE-Tree limits us to attributes whose values are discrete and for which the range of values is known in advance (i.e., at the time the attribute is created). So, for example, a non-negative integer valued attribute can be used only if we know, in advance, what its maximum value is. Because of PUBSUB's heterogeneity in data structures for each attribute, PUBSUB permits all attribute data types. So, for example, we may set the attribute data structure D to RBPST for all attributes whose values are ordered (i.e., two attribute values may be compared to determine whether one is less than the other or whether both are equal), to IT for discrete valued attributes whose range is known in advance, to suffix tree or Aho-Corasick trie for attributes of string type (although the current implementation of PUBSUB doesn't support these structures, they are easily added to PUBSUB).

3. The clustering strategy employed in BE-Tree results in performance degradation when many subscriptions specify a range for the clustering attribute that spans the clustering criterion p. So, for example, if we are clustering on attribute 6 and us-

Parameter \ Experiment	4.1	4.2	4.3
Size	1M	100K-5M	1M
Number of Dimensions	1000	400	100-1500
Average Sub Size	7	7	7
Average Event Size	15	15	15
% Equality Predicates	30	30	30
Matching Probability %	0.01-15	1	1

Figure 6: Parameters used with BEGen for generating datasets

ing the criterion $p = 30$, then all subscriptions with a predicate on attribute 6 that is satisfied by the value 30 are assigned to the same cluster. Suppose that many of these predicates are range predicates of the form $low_i \leq a_6 \leq high_i$. To determine which of these actually match the event value (say) 20, we must examine each of the a_6 ranges in the cluster. This takes time linear in the cluster size, which could be fairly large. PUBSUB overcomes this type of performance degradation by using data structures D that can quickly extract matching subscriptions even from large clusters.

4. EXPERIMENTS

The current version of PUBSUB is implemented in C++ and supports, for D, the data structures interval tree (IT), and red-black priority search tree (RBPST). For our experiments, we required PUBSUB to use the same data structure D for every attribute structure. As mentioned earlier, users may specify which data structure D should be used for which attribute and, in general, we expect the use of a heterogeneous set of data structures. The terms PS-IT, and PS-RBPST refer to PUBSUB with all data structures D set to IT, and RBPST, respectively. For our experiments, we compiled our code on a 64 bit Linux box with a 1.2GHz CPU. We benchmarked the performance of PUBSUB against the pub/sub systems BE-Tree [1] (July 28, 2012 release) and Siena [2, 3]. The BE-tree release used by us has improved search times over the original version used in [1]. On our platform we got about 10x improvement in search performance of BE-tree with respect to the numbers reported in [1]. We note that the times reported in [1] are in milliseconds while those reported in this paper are in microseconds. Our experiments, like those of Sadoghi and Jacobsen [1], are for an application environment where the event rate far exceeds the rate at which subscriptions are inserted/deleted. Hence the focus is on event processing time. As a result, the experiments first initialize the subscription database and then measure the time needed to process events. For the application environment considered in this section, Sadoghi and Jacobsen [1] have established the superiority of BE-Tree over other Pub/sub systems such as k-index [9], Propagation [8], Gryphon [7], SIFT [10], and SCAN [5]. So, we did not include these other systems in our experiments.

The test data (synthetic as well as real) for our experiments were generated using $BEGen$ [1] and our experiments were modeled after those reported in [1]. As in [1], we used two kinds of distributions, namely, uniform and Zipf, for selecting the predicates of a subscription.

For our experiments, the attributes in a subscription were ordered based on the frequency of occurrence of the attributes in the entire set of subscriptions in the system. The ordering was from the least frequent attribute to the most frequent one. This ordering improved PUBSUB performance, particularly for tests on Zipf distribution.

We first ran an experiment (Section 4.1) to determine an appropriate bucket size for PUBSUB. This experiment was followed by several experiments to compare the event processing performance of PS-IT, PS-RBPST, BE-Tree, and Siena. The various parameters used to generate the test data used in each of sections IV-B through IV-J are shown in Figure 6. The parameters are those supported by BEGen [1] and have the following meaning:
Number of Dimensions: The total number of attributes in the system.
Average Sub Size: Average number of attributes in a subscription
Average Event Size: Average number of attributes in an event
% Equality Predicates: Total number of predicates in the subscription that involve the equality operator.
Matching Probability %: Probability that an event will match a subscription.

In the following, the reported event processing time is the average time (microseconds per event) to process an event. This does not include the time needed to process the subscriptions and create the data structure in which the subscriptions are stored (i.e., for example, the time to create the collection of attribute structures used by PUBSUB).

4.1 Determining maximum bucket size

Figure 7 shows how the event processing time varies with maximum bucket size and matching probability.

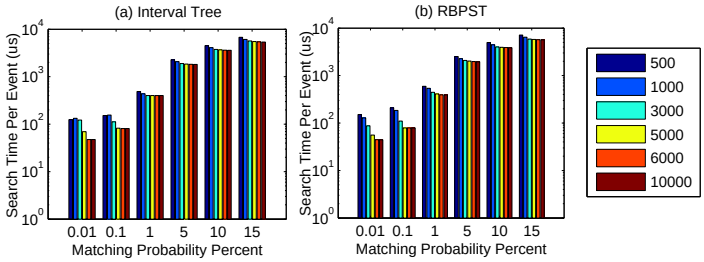

Figure 7: How search time depends (microseconds/event) on bucket size

Bucket sizes ≥ 5000 result in the best performance for the different matching probabilities as well as for both choices of the data structure D. So, for the remaining experiments, we set the maximum bucket size to 5000. We note that in application environments where the subscription insert/delete rate is not low, a smaller bucket size will, most likely, result in overall best performance.

4.2 How search time varies with the number of subscriptions

Figure 8 gives the variation in event processing time as we increase the number of subscriptions. For the uniform tests the reduction in event processing time using any of the PUBSUB schemes compared to BE-Tree is between 19 to 31%. The improvement in search time compared to Siena is between 97 to 99% for the uniform tests. The results for the Zipf tests is comparable with respect to BE-Tree. The performance speedup of PUBSUB with respect to Siena is between 38 to 147 for the Zipf tests. The subscriptions in Zipf tests have a large number of common attributes, which results in deep trees for both PS-RBPST and PS-IT. The search performance of PUBSUB degrades as a results, since a large number of buckets are visited and the subscriptions stored in these buckets are all compared with the event.

The relative performance of the two PUBSUB schemes is comparable. The performance of PS-RBPST is slightly better than that of PS-IT especially when the number of subscriptions exceeded a million and the degree of overlap between subscriptions is high (as

in the Zipf tests).

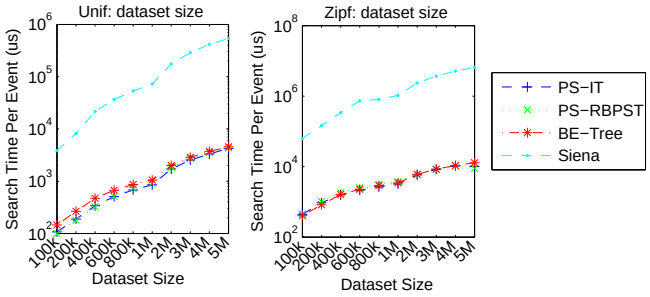

Figure 8: Search time with varying dataset size (microseconds/event)

4.3 How search time varies with the number of dimensions (or attributes) in the system

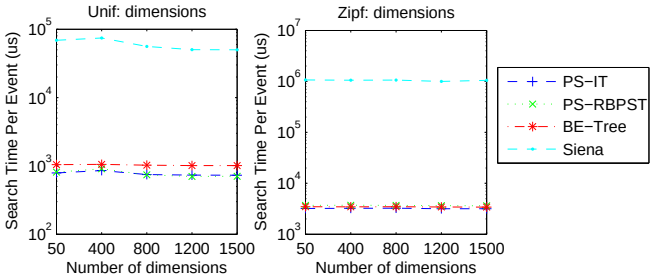

Figure 9: Search time with varying number of dimensions (microseconds/event)

All of the pub/sub systems being studied display the same trend in search time as the number of attributes is increased. The search time decreased slightly with an increase in the number of dimensions. As the dimensions in a system are increased, the degree of overlap among subscriptions tend to decrease if the average number of attributes in subscriptions remain the same. This translates into the observed reduction in search times. On the tests based on uniform distribution, PUBSUB is faster than BE-Tree by 24 to 30%, while on the Zipf tests PUBSUB is 6.2 to 7.5% faster than BE-tree. BE-Tree is faster than Siena for the uniform and Zipf tests.

5. CONCLUSION

PUBSUB is a versatile, scalable, and efficient publish/subscribe system. Although the present implementation includes only 2 choices (interval tree, and red black priority search tree) for the data structure D that is used to partition subscriptions based on the predicates of a single attribute, the set of available data structures for D is readily extendable to include structures such as Aho-Corasick trees [17] and suffix trees [18] for string type attributes and operators. Our selection for the initial data structures was motivated by their suitability for predicates that specify a range of values.

We compared, experimentally, the performance of PUBSUB with that of BE-Tree [1] and Siena [2, 3] in an environment where event processing dominates subscription insert/delete. The same settings were used to generate our datasets as were used in [1]. Additionally, we used very large data sets containing over a million subscriptions. In general, there were two different types of datasets – those based on predicates selected from the attributes' pool using uniform distribution, and those based on predicate selection using Zipf distribution. PUBSUB performed the best on the uniform tests, while on the Zipf tests, the performance of PUBSUB is comparable to BE-Tree. On our tests, the speedup, in event processing, of the fastest version of PUBSUB relative to Siena ranged from a low of 38 to a high of 330 and averaged 201 for the uniform and the Zipf tests. The speedup range relative to BE-Tree was between 1.23 to 1.48 and averaged 1.36 for the uniform tests and was comparable to BE-tree on the Zipf tests.

It should be emphasized that although our experiments used the same data structure for all attribute structures, we expect that in real-world applications optimal performance will be achieved with a heterogeneous selection of data structures with interval trees being used in some attribute structures, red black priority search trees in others, and so on. The architecture of PUBSUB readily supports this heterogeneity.

6. REFERENCES

[1] M. Sadoghi and H.-A. Jacobsen, BE-Tree: An Index Structure to Efficiently Match Boolean Expressions over High-dimensional Discrete Space, *SIGMOD 2011*.

[2] A. Carzaniga, D. Rosenblum, and A. Wolf, Design and evaluation of wide-area event notification service. *ACM Trans. on Computer Systems*, 19, 3, 2001, 332–383.

[3] A. Carzaniga and A. L. Wolf, Forwarding in a Content-Based Network, *ACM SIGCOMM 2003*.

[4] H. Lu and S. Sahni, O(log n) Dynamic Router-Tables for Prefixes and Ranges, *IEEE Transactions of Computers* Vol. 53, No. 10, 2004, 1217–1230.

[5] T. W. Yan and H. Garcia-Molina, Index Structures for Selective Dissemination of Information Under the Boolean Model, *ACM TODS 1994*.

[6] E. M. McCreight, Priority Search Trees, *Siam J. Comput.* Vol. 14, No. 2, May 1985, 257–276.

[7] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, Matching events in a content-based subscription system, *PODC 1999*.

[8] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, Filtering algorithms and implementation for fast pub/sub systems, *SIGMOD 2001*.

[9] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina, Indexing Boolean Expressions, *VLDB, 2009*.

[10] T. W. Yan and H. Garcia-Molina, The SIFT Information Dissemination System. *ACM TODS, 1999*.

[11] D. Mehta and S. Sahni, Handbook of Data Structures and Applications, *Chapman & Hall/CRC*, 2005.

[12] A. Mitra, M. Maheswaran and J. A. Rueda, Wide-Area Content-based Routing Mechanism, *IPDPS, 2003*.

[13] W. Rao, L. Chen, A. W-C. Fu, H. Chen and F. Zou, On Efficient Content Matching in Distributed Pub/Sub Systems, *INFOCOM, 2009*.

[14] F. Cao and J. P. Singh, Efficient Event Routing in Content-based Publish-Subscribe Service Networks, *INFOCOM, 2004*.

[15] M. Petrovic, I. Burcea and H-A. Jacobsen, S-ToPSS: Semantic Toronto Publish/Subscribe System, *VLDB, 2003*.

[16] A. Yu, P. K. Agarwal and J. Yang, Generating Wide-Area Content-Based Publish/Subscribe Workloads, *NetDB, 2009*.

[17] A. V. Aho and M. J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, *Communications of the ACM*, Volume 18, No. 6, June 1975, 333-340.

[18] E. M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *Journal of the ACM*, Volume 23, No. 2, 1973, 262-272.

[19] W. Lu and S. Sahni, Efficient two-dimensional multibit tries for packet classification, *IEEE Transactions on Computers* Volume 58, No. 12, 2009, 1695-1709.

[20] J. L. Bentley, Decomposable searching problems, *Information Processing Letters*, Volume 8, No. 5, 1979, 244-201.