Matching Events in a Content-based Subscription System

Marcos K. Aguilera[1] Robert E. Strom[2] Daniel C. Sturman[2] Mark Astley[3] Tushar D. Chandra[2]

Abstract

Content-based subscription systems are an emerging alternative to traditional publish-subscribe systems, because they permit more flexible subscriptions along multiple dimensions. In these systems, each subscription is a predicate which may test arbitrary attributes within an event. However, the matching problem for content-based systems — determining for each event the subset of all subscriptions whose predicates match the event — is still an open problem. We present an efficient, scalable solution to the matching problem. Our solution has an expected time complexity that is *sub-linear* in the number of subscriptions, and it has a space complexity that is *linear*. Specifically, we prove that for predicates reducible to conjunctions of elementary tests, the expected time to match a random event is no greater than $O(N^{1-\lambda})$ where N is the number of subscriptions, and λ is a closed-form expression that depends on the number and type of attributes (in some cases, $\lambda \approx 1/2$). We present some optimizations to our algorithms that improve the search time. We also present the results of simulations that validate the theoretical bounds and that show acceptable performance levels for tens of thousands of subscriptions.

1 Introduction

Publish/subscribe (pub/sub) is a paradigm for interconnecting information providers to information consumers in a distributed environment. Information providers publish information in the form of *events* to the pub/sub system, information consumers subscribe to a particular category of events within the system, and the system ensures the timely delivery of published events to all interested subscribers. A pub/sub system is typically implemented over a network of *brokers* that are responsible for routing events between publishers and subscribers.

The earliest pub/sub systems were *group-based*. In these systems, each event is classified as belonging to one of a fixed set of *groups* (also known as subjects, channels, or topics). Publishers are required to label each event with a group name; consumers subscribe to all events in a particular group. For example a group-based pub/sub system for stock trading may define a group for each issue. Publishers post information labeled with the appropriate issue as the group name, and subscribers subscribe to information regarding some issue. In the past decade, systems supporting this paradigm have matured significantly resulting in several academic and industrial strength solutions [2, 7, 8, 9]. A similar approach has been adopted by the OMG for CORBA event channels [5].

An emerging alternative to group-based systems is content-based subscription systems [1, 3, 10]. These systems support a number of *information spaces*, each associated with an *event schema* defining the type of information contained in each event. Our stock trade example may be defined as an information space whose event schema is a tuple containing three *attributes*: an issue, a price, and a volume, of string, dollar, and integer types respectively. A subscription is then a predicate over these attributes, such as `(issue="IBM") and (price<120) and (volume>1000)`.

Note that with content-based pub/sub, subscribers have the added flexibility of choosing filtering criteria along multiple dimensions, without requiring pre-definition of groups. In our stock trading example, the group-based subscriber is forced to select trades by issue name. In contrast, the content-based subscriber is free to use an orthogonal criterion, such as volume, or indeed a collection of criteria, such as issue, price and volume. Furthermore, content-based pub/sub removes the administrative overhead of maintaining and defining groups, thereby making the system easier to manage. Finally, content-based pub/sub is more general in that it can be used to easily implement group-based pub/sub while the reverse is not true. While content-based pub/sub is the more powerful paradigm, efficient and scalable implementations of such systems have not yet been developed.

In order to efficiently implement a pub/sub system, one must first find an efficient solution to the problem of matching an event

[1] Department of Computer Science, Cornell University, Ithaca, N.Y. 14853-7501, aguilera@cs.cornell.edu

[2] IBM T.J. Watson Research Center, Yorktown Heights, N.Y. 10598, {strom, sturman, tushar}@watson.ibm.com

[3] Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave, Urbana, I.L. 61801, astley@cs.uiuc.edu

against a large number of subscriptions. We refer to this problem as *the matching problem.* One of the strengths of group-based pub/sub systems is that this problem is straightforward to solve using a mere table lookup. However, for content-based pub/sub systems, the matching problem does not have a known, scalable solution.

A simple algorithm for content-based matching is to test all subscriptions against each event. This naive algorithm runs in time linear in the number of subscriptions. In practice, pub/sub systems may be deployed in environments with tens of thousands of publishers and subscribers, and in general pub/sub systems have been aimed at providing support for large-scale, widely distributed applications. Therefore, a linear time solution to the matching problem is not adequate.

In this paper, we propose an algorithm whose time complexity is sub-linear in the number of subscriptions, and whose space complexity is linear. Our algorithm initially pre-processes the set of subscriptions into a data structure that allows fast matching. Pre-processing makes sense in most pub/sub environments, where subscriptions tend to change infrequently enough that they can be considered approximately static, but where events are published at a fast rate. In such cases, the speed-up gained by pre-processing far outweighs its cost. Furthermore, our algorithm allows subscription updates to be incrementally incorporated into existing pre-processed data.

In the pre-processing phase, our algorithm creates a *matching tree*. In the matching tree, each node is a test on some of the attributes, and the edges are results of such tests. Each lower level of the tree is a refinement of the tests performed at higher levels, and at the leaves of the tree we have the subscriptions. With such a tree, we can find the subscriptions that match an event by traversing the tree starting from the root; at each node, we perform the test prescribed by the node and follow all those edges consistent with the result (there may be more than one edge). We then repeat these steps until we get to the leaves. The leaves that are finally visited correspond to the subscriptions that match the event.

In the case where subscriptions consist of equality tests on the attributes, the asymptotic complexity of our algorithm is significantly better than the one of the naive algorithm. More precisely, the expected time to match a random event is $O(N^{1-\lambda})$ where N is the number of subscriptions, and λ depends on the number and type of attributes (in some cases, $\lambda \approx 1/2$). The constants hidden behind the big-O notation are quite reasonable.

In summary, the main contributions of this paper are as follows:

1. We present a generic matching algorithm whose performance scales better than that of the naive algorithm;

2. In the case where subscriptions consist of equality tests, we show that the matching time grows only sub-linearly in the number of subscriptions, and that the space requirement is linear in the number of subscriptions. This is the first matching algorithm with such characteristics.

We also present some optimizations to the matching algorithm, and show the result of simulations that validate the practicality of the algorithm.

This paper is organized as follows: In Section 2 we formally define the matching problem. We give the general version of our algorithm for this problem in Section 3. This version allows subscriptions that consist of conjunctions of arbitrary tests on attributes. In Section 4 we present a version of our algorithm specialized for the case when subscriptions contain only equality tests on attributes, and show that the asymptotic time complexity of this algorithm is sub-linear in the number of subscriptions. In Section 5 we discuss enhancements that speed up the algorithm. In Section 6 we describe related work, and we conclude the paper in Section 7. In the appendices, we provide some algorithmic details that were omitted from our explanations.

2 The matching problem

An *event schema* defines the space of possible events, by specifying *attribute* names and types. A subscription *sub* is a boolean predicate on events. We say that an event e *matches* a subscription *sub* if and only if $sub(e) = true$. In the matching problem, we are given an event schema and a finite set Sub of subscriptions.[1] Subsequently, we are given an event e, and the goal is to determine all those subscriptions in *Sub* that match e. We allow pre-processing of the set Sub before we are given e.

A solution to the matching problem has two phases: *pre_process*(Sub) and *match*(*pre_processed_data*, *event*). The first phase *pre_process*(Sub) takes the set of subscriptions Sub and outputs an internal representation of the subscriptions. The second phase *match*$(pre\_processed\_data, event)$ takes this internal representation and an event, and outputs those subscriptions that match the event.

We measure the performance of the solution by three parameters:

- *Pre-processing space complexity.* The amount of data generated by $pre\_process$;

- *Pre-processing time complexity.* The time needed to run $pre\_process$;

- *Matching time complexity.* The time needed to run $match$.

3 The tree matching algorithm

The matching problem can be solved easily by testing an event against each subscription (in this case, there is no pre-processing). This naive solution runs in time proportional to the number of subscriptions. In many applications, the number of subscriptions can be extremely high — in the order of magnitude of tens or hundreds of thousands. If events are published at a fast rate, then events need to be matched at a fast rate as well, and the naive solution does not perform adequately. In this section, we provide an algorithm that performs significantly better.

Our algorithm initially pre-processes the set of subscriptions into a *matching tree*. We now describe this tree in detail, and then we explain how it is used to match events. Henceforth, we assume that each subscription is a conjunction of *elementary predicates*, where each elementary predicate represents one possible result of an *elementary test*. An elementary test is a simple operation on one or more attributes of the event e.

That is, a subscription sub is as follows:

$$\begin{aligned} sub &:= pr_1 \wedge pr_2 \wedge \cdots \wedge pr_k \\ pr_i &:= test_i(e) \rightarrow res_i \end{aligned}$$

where the notation $test_i(\ldots) \rightarrow res_i$ means that $test_i$ produces result res_i. For example, in the subscription `(city = New York) and (temperature < 40)`, we have two elementary predicates, pr_1 and pr_2, where

$$\begin{aligned} pr_1 &= test_1(\ldots) \rightarrow \texttt{New York} \\ pr_2 &= test_2(\ldots) \rightarrow \text{“}<\text{"} \\ test_1 &= \text{“}examine\ attribute\ \texttt{city}\text{"} \\ test_2 &= \text{“}compare\ attribute\ \texttt{temperature 40}\text{"} \end{aligned}$$

[1] We assume that subscriptions with identical predicates are coalesced into a single subscription.

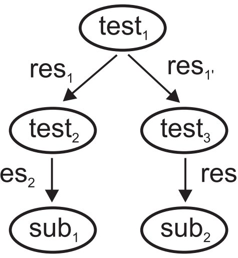

Figure 1: Example of a matching tree

In the matching tree, each non-leaf node contains a test, and edges from that node represent results of that test. A leaf node ℓ contains a subscription sub, instead of a test. Intuitively, sub is the subscription described by walking the tree from the root to ℓ and taking the conjunction of the elementary predicates. More precisely, for any node v on the tree, we define a predicate $pred(v)$ as follows[2]: let the path from the root to v be $(test_1, res_1, test_2, res_2, \ldots, test_j, res_j, v)$; then

$$pred(v) := (test_1 \rightarrow res_1) \wedge \cdots \wedge (test_j \rightarrow res_j) \qquad (1)$$

With this, we require that the subscription sub contained in a leaf ℓ satisfies:

$$pred(\ell) \equiv sub \qquad (2)$$

where $\equiv$ denotes logical equivalence.

Here are some simple examples of the matching tree. Suppose subscriptions sub_1 and sub_2 share $test_1$ as follows:

$$\begin{aligned} sub_1 &= (test_1 \rightarrow res_1) \wedge (test_2 \rightarrow res_2) && (3) \\ sub_2 &= (test_1 \rightarrow res_1') \wedge (test_3 \rightarrow res_3) && (4) \end{aligned}$$

In this case, the matching tree is shown in Figure 1.

The tree can have special "don't care edges" — which we call **-edges* — that represent the fact that subscriptions reachable through the edge do not care about the result of a test. These edges are necessary when some of the subscriptions are independent of that test. For example, suppose:

$$\begin{aligned} sub_3 &= (test_1 \rightarrow res_1) \wedge (test_2 \rightarrow res_2) \\ sub_4 &= (test_3 \rightarrow res_3) \wedge (test_4 \rightarrow res_4) \end{aligned}$$

In this case, the matching tree is shown in Figure 2. When the matching tree has *-edges, for each node v we define $pred(v)$ exactly as before (see Equation 1), and we assume by convention that $test_i \rightarrow *$ is equivalent to *true*. For example, in Figure 2, we have that $pred(sub_4) = (test_1 \rightarrow *) \wedge (test_3 \rightarrow res_3) \wedge (test_4 \rightarrow res_4) \equiv (test_3 \rightarrow res_3) \wedge (test_4 \rightarrow res_4) = sub_4$.

If $test_1$ and $test_3$ happen to be related, the matching tree could look different. More precisely, if $(test_3 \rightarrow res_3) \Rightarrow (test_1 \rightarrow res_1)$ then another possible matching tree is shown in Figure 3. Note that it is still the case that $pred(sub_4) \equiv sub_4$. Intuitively, this matching tree is better than the one in Figure 2, because to match an event, in Figure 2 we always need to evaluate $test_1$ and $test_3$, whereas in Figure 3, we only evaluate $test_3$ when $test_1$ evaluates to res_1.

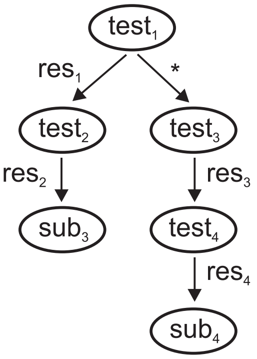

Figure 2: Matching tree with a *-edge

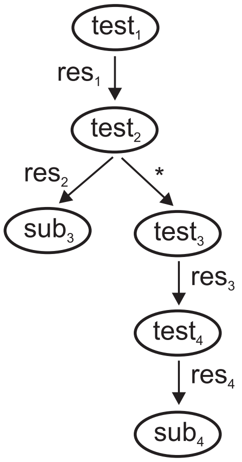

Figure 3: Matching tree when $(test_3 \rightarrow res_3) \Rightarrow (test_1 \rightarrow res_1)$

[2] If v is the root node, we define $pred(v)$ to be $true$.

```
procedure match($Tree$, $event$)
    visit($Tree$, $root$, $event$)

procedure visit($Tree$, $v$, $event$)
    if $v$ is a leaf node of Tree then output($v$)
    else
        perform test prescribed by $v$ on event
        if $v$ has an edge $e$ with the result of test
        then visit(Tree, (child of $v$ at the endpoint
                 of $e$ in Tree), event)
        if $v$ has a *-edge $e$
        then visit(Tree, (child of $v$ at the endpoint
                 of * in Tree), event)
```

Figure 4: General matching algorithm

In the specialization of the generic matching algorithm that we consider in Section 4, different tests in the tree will *never* be related.

The algorithm $pre\_process$ that creates the matching tree works as follows. We assume that the elementary predicates in subscriptions are ordered according to a fixed total order. To create the matching tree, we start with the empty tree, and we process one subscription at a time by examining each of its elementary predicates (in order), and adding nodes to the tree as necessary. For instance, the processing of sub_1 (see Equation 3) would create nodes $test_1$, $test_2$ and sub_1 of Figure 1; and the subsequent processing of sub_2 (see Equation 4) would create the remaining two nodes (note that $test_1$ is not added again to the tree). The details of the algorithm are given in Appendix A.

The algorithm *match* that uses the tree to match events is given in Figure 4. The idea is to walk the matching tree by performing the test prescribed by each node and following the edge that represents the result of the test, and the *-edge if it is present. The set of matching subscriptions will be all those leaves that are visited. This particular algorithm traverses the tree in a depth-first order, but clearly other orderings, such as breadth-first, would also work.

4 Matching equality tests

We now consider a version of the tree matching algorithm specialized to the case where subscriptions consist of conjunctions of equality tests of attributes against constant values. We analyze the performance of the tree matching algorithm in this special case, and show that (1) the time complexity to match events is sub-linear in the number of subscriptions, (2) the space complexity is linear in the number of subscriptions, and (3) the time complexity to pre-process is linear in the number of subscriptions.

More precisely, in this section we assume subscriptions are of the form

$$sub := (attr_1 = v_1) \wedge \cdots \wedge (attr_K = v_K)$$

where K is the number of attributes in the schema, and each v_j is either a constant or it is *, meaning that any value matches the j-th predicate.

With this assumption, we can assign each level of the matching tree to an attribute. For simplicity we assume that the i-th attribute is assigned to level i. At level i, all nodes contain the test "examine the contents of attribute i", and edges from the nodes are the values against which the i-th attribute is being tested. For example, suppose the set of subscriptions is

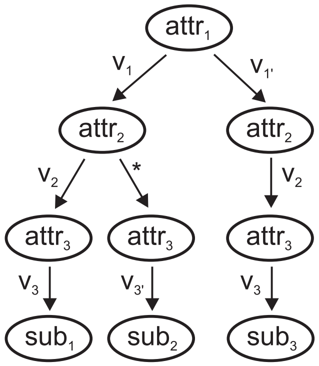

Figure 5: A matching tree for equality tests

$$\begin{aligned}
sub_1 &:= (attr_1 = v_1) \wedge (attr_2 = v_2) \wedge (attr_3 = v_3) \\
sub_2 &:= (attr_1 = v_1) \wedge (attr_2 = *) \wedge (attr_3 = v_3') \\
sub_3 &:= (attr_1 = v_1') \wedge (attr_2 = v_2) \wedge (attr_3 = v_3)
\end{aligned}$$

In this case, the subscription tree is shown in Figure 5.

The pre-processing function that creates this tree is straightforward and is given in Appendix B. The matching function is the same as in Section 3. We now analyze the performance of the algorithm.

Pre-processing time complexity

For each subscription that we need to add to the matching tree, we spend time proportional to the number K of attributes in procedure *pre_process*. Therefore, if there are N subscriptions, the total time spent in *pre_process* is $O(NK)$.[3] Since K is a constant (which depends on the event schema), the pre-processing time is linear in the number of subscriptions.

Space complexity

For the space complexity, note that each subscription can add at most $K + 1$ nodes to the matching tree, namely, one for each attribute and one for the leaf node containing the subscription. Thus, the space required for the matching tree is $O(NK)$, that is, linear in the number of subscriptions.

Matching time complexity

We now analyze the time required to match an event in procedure *match*. We measure the event matching time by counting the number of tree nodes that are visited during the match. In any reasonable implementation of the matching procedure *match*, this number is proportional to the actual time necessary to match the event, since the algorithm performs a simple elementary test per node, which is assumed to take constant time. For example, in a typical implementation, the attribute is evaluated, and its value searched in a hash table to determine the successor edge (if any); that successor edge, if present, and the *-edge, if present, are then followed.

The event matching time is a function of the set of subscriptions: a large set of subscriptions generates a large matching tree, which requires a larger time to run the algorithm. The matching time is also a function of the particular event being matched; indeed, different events cause different sets of nodes to be visited during matching — even if the set of subscriptions is kept constant. One way to

[3] Note that *any* algorithm that reads all N subscriptions requires time at least NK.

handle this difficulty is to consider the worst case: how long does it take to match the worst possible event, as a function of the set of subscriptions? Unfortunately, there are cases where the worst case performance is linear in the number of subscriptions. For example, let v be any value and consider a tree that contains only edges labeled v and *-edges. To match the event whose attributes are all v, we need to visit all nodes in the tree. Thus the matching time is equal to the size of the tree. It is easy to see that the size of the tree is between $|S|$ and $(K+1)|S|$ where S is the set of subscriptions and K is the number of attributes in the schema. Thus, in this example, the (worst-case) matching time grows linearly with the number of subscriptions.

In the rest of this section we take a different approach. We compute the *expected* time to match a random event, and show that even with the subscriptions chosen to maximize this expected time, the expected time is sub-linear in the number of subscriptions. Although here we assume a uniform distribution on events, the techniques we describe can be used to analyze other distributions as well. We also make the simplifying assumption that all attributes range over the same set of values, but our analysis can be extended to the more general case where attributes range over different set of values (this extension is very cumbersome, however).

Henceforth, let:

- K be the number of attributes in the schema, and $\bar{K} := K+1$;
- V be the number of possible values for each attribute;
- S be an arbitrary set of subscriptions.
- $C(S)$ the expected time to match a random event against the set S of subscriptions.

We can obtain an easy upper bound on $C(S)$ by noting that when we match an event we follow at most two branches for every level in the tree. Thus, the total number of nodes visited is at most $2^0 + 2^1 + \cdots + 2^K$.[4] This bound, however, is unsatisfactory because it is exponential in K. We are interested in bounds that are polynomial in K, V and $|S|$, and we next show one such a bound that is sublinear in $|S|$.

Theorem 1 *Suppose that all events are equally likely. The expected time $C(S)$ to match a random event is bounded above by*

$$C(S) \leq \frac{V\bar{K}(\bar{K}|S|^{1-\lambda} - 1)(\ln V + \ln \bar{K})}{(VK-1)\ln K} \tag{5}$$

where

$$\lambda \quad := \quad \frac{\ln V}{\ln V + \ln K} > 0$$

Since $V \geq 2$ and $\bar{K} \geq 2$, we have $(V\bar{K})/(V\bar{K}-1) \leq 4/3$. Also, since $\bar{K} \geq 2$, we have $1/\ln \bar{K} < 3/2$. By introducing these results in equation (5), we derive the following

Corollary 1 $C(S) \leq 2\bar{K}|S|^{1-\lambda}\ (\ln V + \ln \bar{K})$.

We now proceed to prove Theorem 1. Henceforth, let S_T be the subscription tree obtained when we pre-process S. For each node v of this tree, we define $cost(v)$ to be the number of times that this node is visited when we run the matching algorithm with all the possible V^K events. Note that this number is always a power of V. For example, if v is the root node of the tree, then $cost(v)$ is V^K. In general, $cost(v) = V^{K-\lambda}$ where λ is the number of non-* edges in the path from the root to node v.

When all V^K events are equally likely, then the probability that a node v is visited when matching a random event is clearly equal to $V^{-K}cost(v)$. Thus, the expected number $C(S)$ of nodes of S_T visited is:

$$C(S) = V^{-K} \sum_{v \in nodes(S_T)} cost(v) \tag{6}$$

where $nodes(S_T)$ is the set of nodes of the tree S_T.

Lemma 1 *For any $j : 0 \leq j \leq K$, S_T contains at most $V^j \binom{\bar{K}}{j+1}$ nodes with cost equal to V^{K-j}.*
Proof. Let j be such that $0 \leq j \leq K$. A node n has cost V^{K-j} if and only if the path from the root to the node has exactly j non-* edges. Such paths are uniquely determined by (1) the number of edges in the path, (2) the position of the non-* edges and (3) the values of the non-* edges. We can bound the number of paths with j non-* edges by counting the possible ways to specify (1), (2) and (3). The number n of edges is between j and K; the position of the non-* edges are j distinct numbers between 1 and n, and so there are $\sum_{i=j\ldots K} \binom{i}{j} = \binom{K+1}{j+1}$ ways of choosing (1) and (2). Moreover, we can assign V distinct values for each non-* edges. Therefore, the number of paths in S_T with exactly j non-* edges is at most $V^j\binom{K+1}{j+1}$. □

Corollary 2 *For any $j : 0 \leq j \leq K$, S_T contains at most $\bar{K}[\bar{K}V]^j$ nodes with cost equal to V^{K-j}.*
Proof.

$$V^j \binom{\bar{K}}{j+1} \leq V^j \frac{\bar{K}^{j+1}}{(j+1)!} \leq \bar{K}[\bar{K}V]^j \tag{7}$$

□

Lemma 2 *S_T has at most $\bar{K}|S|$ nodes.*
Proof. A subscription is associated with a path with K edges (one edge for each attribute). This path contains $K+1 = \bar{K}$ nodes. Thus, if the tree has $|S|$ subscriptions, it has at most $\bar{K}|S|$ nodes. □

Henceforth, we order the nodes of S_T by decreasing order of their cost, and we let $f(i)$ be the cost of the i-th node in the order (if i is greater than the number of nodes, we let $f(i)$ be zero). By Equation (6) and Lemma 2, we have that

$$C(S) = V^{-K} \sum_{i=1}^{\bar{K}|S|} f(i) \tag{8}$$

Definition 1 *Henceforth, let*

$$g(x) := (Ax+B)^{-\lambda}$$

where

$$A \quad := \quad V^{-\bar{K}/\lambda}[V - 1/\bar{K}] \tag{9}$$

$$B \quad := \quad V^{-\bar{K}/\lambda} \tag{10}$$

$$\lambda \quad := \quad \frac{\ln V}{\ln V + \ln K} < 1 \tag{11}$$

Lemma 3 $f(x) \leq g(x)$
Proof. By Corollary 2 and the definition of f, we have that for each i such that $0 \leq i \leq K$ and for each j such that

$$\sum_{p=0\ldots i-1} \bar{K}[\bar{K}V]^p < j \leq \sum_{p=0\ldots i} \bar{K}[\bar{K}V]^p$$

the following holds:

[4] This actually gives a bound on the time to match *any* event, not just on the average matching time.

$$f(j) \leq V^{K-i}$$

Now,

$$g\big(\sum_{p=0\ldots i} \bar{K}[\bar{K}V]^p\big) = g\big(\bar{K}\frac{[\bar{K}V]^{i+1}-1}{KV-1}\big)$$

By using the definition of g, we conclude that

$$g\big(\sum_{p=0\ldots i} \bar{K}[\bar{K}V]^p\big) = V^{K-i}.$$

The lemma now follows because g is a non-increasing function. □

Proof of Theorem 1. We have that

$$\begin{aligned} C(S) &= V^{-K}\sum_{x=1}^{\bar{K}|S|} f(x) \\ &\leq V^{-K}\sum_{x=1}^{\bar{K}|S|} g(x) \\ &\leq V^{-K}\int_0^{\bar{K}|S|} g(x)dx \\ &= V^{-K}\frac{(A\bar{K}|S|+B)^{1-\lambda}-B^{1-\lambda}}{A(1-\lambda)} \end{aligned}$$

After replacing the values of A and B given in (9) and (10), and simplifying, we obtain:

$$C(S) \leq \frac{V\bar{K}[(V\bar{K}|S|-|S|+1)^{1-\lambda}-1]}{(VK-1)(1-\lambda)}$$

After using the fact that $V\bar{K}|S| - |S| + 1 \leq V\bar{K}|S|$ and that $(V\bar{K})^{1-\lambda} = \bar{K}$, and after replacing the value of λ given in (11) we obtain Equation (5). □

5 Optimizations to the general tree matching algorithm

A certain amount of static analysis of the subscription tree can be used to streamline the search in the above algorithm. An extremely straightforward and obvious optimization is to collapse a chain of edges into a single edge whenever the intermediate nodes have only a *-edge. For example, the edge from node J to node A in Figure 6 can be rewritten to lead directly to node B. In the simulation runs discussed later, where some attributes are rarely tested by a subscription, this simple transformation of the tree led to a 60% reduction in matching time.

A second optimization allows some successor nodes to be precomputed at analysis time, thereby reducing the number of attribute re-evaluations needed at matching time. This optimization is based upon the assumption that the parallel subsearches (steps 9 and 12 of Figure 4) will be performed in some known serial order, e.g. a non-* edge will be followed before a *-edge. We can then annotate the search data structure to use the information obtained by traversing the non-* edges to skip over tests in the *-path which are implied by tests already performed in the non-* path.

For example, let us suppose that all subscriptions are equality tests, that each elementary test is a simple evaluation of an attribute, that the matching tree is the one shown in Figure 6, and that we always follow non-* paths before *-paths. Suppose that we are matching the event $\langle 1, 2, 3, 8, 2\rangle$. We follow the path $\langle a_1 = 1, a_2 = 2, a_3 = 3\rangle$ to node C in Figure 6, and then find ourselves blocked when $a_4 = 8$ and there is no non-* path to follow. Static analysis can predict that any search reaching node C must later traverse the paths labeled $\langle a_1 = *, a_2 = 2, a_3 = 3\rangle$, $\langle a_1 = 1, a_2 = *, a_3 = 3\rangle$ and $\langle a_1 = 1, a_2 = 2, a_3 = *\rangle$, if they exist, since these predicates are implied by $\langle a_1 = 1, a_2 = 2, a_3 = 3\rangle$. The second and third of these paths exist and lead to nodes G and H. At analysis time, we designate G and H as *successors* of C. But the remaining path (to the dotted node labeled D) does not exist; so instead of D, D's successors (the nodes E and F whose paths are $\langle a_1 = *, a_2 = *, a_3 = 3\rangle$ and $\langle a_1 = *, a_2 = 2, a_3 = *\rangle$) are designated as successors of C. (Of course, the node I, reached from C via a *-edge, is also designated as a successor.)

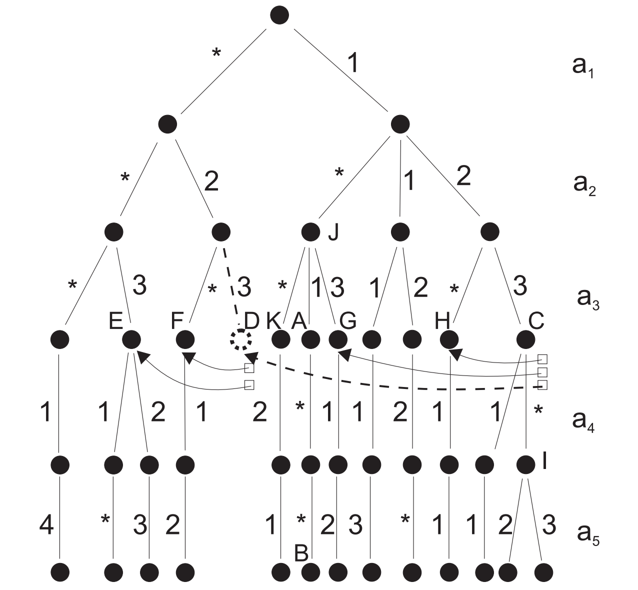

Figure 6: A matching tree with successor node annotations

More formally, if the path p to a node N ends in n consecutive non-* segments, the successor set $SS(p)$ corresponding to that path consists of the n paths p_i obtained by replacing one of the non-* segments with a *. The successor node set stored in the node at p contains: for each p_i in $SS(p)$, a pointer to the node reached by path p_i if it exists, else the nodes in the successor node set of the node at p_i. If there is a child node reachable from N by a segment labeled *, this child node is also included in the successor node set.

In the general case, node N_2 is a successor of N_1 iff $pred(N_1) \Rightarrow pred(N_2)$ and there does not exist an intermediate node N_3 such that $pred(N_1) \Rightarrow pred(N_2) \Rightarrow pred(N_3)$.

Even more aggressive static analysis can be performed. For example, suppose we know at analysis time that we will always follow a successful test before following *-edges. Then if we have reached node C and if we are blocked, we know not only that $\langle a_1 = 1, a_2 = 2, a_3 = 3\rangle$, but also that $a_4 \neq 1$. This information allows us to refine the successor set, since we know that at nodes F, G, and H, the test of a_4 will also fail. We replace F, G, and H with their successor nodes, which in this case is the single node K, the successor of G.

When this form of static analysis is used, the order of following nodes at matching time is constrained so that only non-* branches are followed until a node is reached for which there is no child node labeled with the value of the tested attribute, or until a leaf is reached. Then the successor node set is used to determine where to continue the search. The performance of this approach has been measured, and leads to increased (but still linear) space, and about a 20% additional improvement in search time relative to the first optimization.

The search can be further improved, at the cost of increased space, by factoring out certain attributes. That is, certain attributes — preferably those for which the subscriptions rarely contain "don't care" tests — are selected as indices. A separate subtree is built for each possible value of the index attributes. The subtrees do not include tests for the index attributes. A subscription (minus the tests for index attributes) is placed into each subtree consistent with those of its elementary predicates which test the index attributes. This means that if the subscription has "don't care" on m of the index attributes, and there are V values per attribute, it must be inserted into V^m subtrees. Therefore, in order for this optimization to be scalable, the number of index attributes must be kept small enough so that V^m is small relative to the number of subscriptions.

6 Related Work

As far as we know, there are no other algorithms for the matching problem with sub-linear time-complexity, and linear space complexity. The content-based subscription systems that have been developed so far have not yet adapted scalable matching algorithms. SIENA allows content-based subscriptions to a distributed network of event servers (brokers) [3]. SIENA filters events before forwarding them on to servers or clients. However, a scalable matching algorithm for use at each server has not been developed. The Elvin system [10] uses an approach similar to that used in SIENA. Publishers are informed of subscriptions so that they may "quench" events (not generate events) for which there are no subscribers. In [10], plans are discussed for optimizing Elvin event matching by integrating an algorithm similar to the one in this paper. This algorithm, presented in [4], converts subscriptions into a deterministic finite automata for matching. However, the main difference between [4] and our work is that we seek matching algorithms with (worst-case) space complexity *linear* in the number of subscriptions, while in [4], the space complexity is exponential.

Another algorithm for optimizing matching is discussed in [6]. At analysis time, one of the tests a_{ij} of each subscription is chosen as the gating test; the remaining tests of the subscription (if any) are residual tests. At matching time, each of the attributes a_j in the event being matched is examined. The event value v_j is used to select those subscriptions i whose gating tests include $a_{ij} = v_j$. The residual tests of each selected subscription are then evaluated: if any residual test fails, the subscription is not matched; if all residual tests succeed, the subscription is matched. Our tree matching algorithm performs this type of test for each attribute, not just a single gating test attribute.

7 Discussion

In this paper, we have presented a matching algorithm suitable for a content-based subscription system. For the case where subscriptions contain only equality tests, the algorithm matches events in expected time sub-linear in the number of subscriptions, given a uniform distribution of events but a worst-case set of subscriptions. The space requirement for the matching tree is linear in the number of subscriptions.

In addition to the theoretical analysis of this algorithm, performance was also tested with a variety of simulated loads. In these tests, we assumed an event schema of K attributes, each attribute having V possible values.

We generated a random mix of N subscriptions as follows: We assumed that the attributes varied in "popularity", where popularity measured the likelihood p_{care} that a particular subscription would contain a test for this attribute as opposed to a "don't care". By convention, the first attribute was the most popular, with a $p_{care}(1) = p_1$. Each successive attribute was progressively less popular by a degradation factor of D; that is $p_{care}(i+1) = Dp_{care}(i)$. The values tested in the subscriptions varied according to a Zipf distribution.

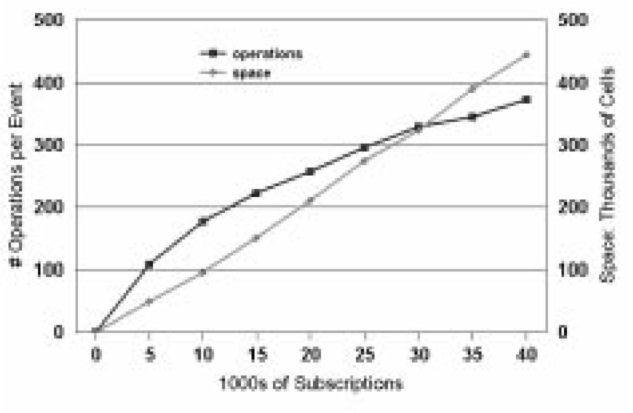

Figure 7: Performance of matching algorithm under simulated workload

We generated random events assuming that the V possible values of each attribute were uniformly distributed.

Figure 7 shows a set of simulations for $V = 3$, $K = 30$, and the factoring optimization for 3 index attributes (that is, 27 subtrees). Values of p_1 and D were chosen so that the number of matches per event was held at 100 independent of N. The space was measured by counting the number of edges plus the size of the successor sets used by the optimization discussed in Section 5.

Other measurements in an actual Java-based prototype have shown that even with as many as 25,000 subscriptions, we can match an event in under 4 milliseconds, even with a fairly unoptimized algorithm.[5]

The analysis and results above are for the special case where all attribute tests are equality tests. We also have a version of the algorithm for inequality and range tests. However, we do not yet have a good enough definition for "typical" ranges to generate simulated loads for a performance analysis. Work on a theoretical analysis of the algorithm with range tests is underway. We are also working on analyzing the performance improvements of the optimizations of Section 5.

The authors' Gryphon pub/sub system [1] uses this matching algorithm (for both equality and inequality tests) to implement a distributed, high-performance content-based pub/sub system. The goal of the Gryphon project is to advance the state-of-the-art in distributed messaging from simple group-based pub/sub, to a full featured *message brokering* system incorporating content-based queries and customized message transformations.

Acknowledgements

We would like to thank the anonymous referees for helpful comments.

[5] This simulation was run on a Pentium 100Mhz, with $K = 30$ and $V = 30$. Subscriptions were generated randomly with a probability of 65% of having a * for each attribute.

A The pre-processing algorithm

We now present in detail the *pre_process* algorithm that is briefly outlined in Section 3. This algorithm is used to generate the matching tree, and is shown in Figure 8. The matching tree is represented by a set T of triples (v, r, v'), where such a triple represents the fact that is an edge labeled r from node v to node v'. The root of the tree is represented by a specially designed node called *tree_root*. For each node v, $v.data$ represents the data associated with v (this data is a test if v is not a leaf, and it is a subscription if v is a leaf).

Procedure *pre_process* processes each subscription in the set Sub, one at a time, by invoking procedure *process_sub*. The latter procedure is responsible for adding the subscription to the currently existing tree. Initially, we check if the tree already exists, and if not we create it (lines 7 and 8). Next, we loop over the primitive predicates in the subscription to check which are already present in the tree (loop in lines 11–25). Finally, we add the remaining primitive predicates that are not yet in the tree (lines 26–35).

The loop in lines 11–25 starts at the root of the tree ($v = tree\_root$) and proceeds down the tree by successively checking that the tree contains the primitive predicates $t_1 \rightarrow r_1, \ldots, t_q \rightarrow r_q$ of the subscription. In line 12 we check if the current tree node v is a leaf, and in that case, we replace that node with a primitive predicate and we exit the loop by setting *found* to *false*. In line 17, we deal with the case that the current tree node v is the current test t_i. In this case, there are two sub-cases: the tree does not contain an edge for the current result r_i (line 18), and the tree already contains such an edge (line 19). In the first sub case, we simply exit the loop by setting *found* to *false*. In the second sub case, we follow that edge on the tree, and continue the loop with the next primitive test.

If v is not the current test t_i, we continue searching the tree for t_i as follows: (1) if the test in the current node v is related to result r' of test t_i as we described in Section 3 (that is, $(t_i \rightarrow r_i) \Rightarrow (v.data \rightarrow r')$ for some edge r' in the tree), then we follow edge r' (line 21), (2) if there is some *-edge at v, then we follow that edge (line 22); or (3) if there are no *-edges at v, then we create a *-edge at v pointing to a node with t_i and we exit the loop (lines 24 and 25).

Once we exit the loop of lines 11–25, we check if there are still primitive predicates that need to be added to the tree (line 26). In that case, we add those predicates, followed by the subscription itself (lines 27–29). Else, we add the subscription to the tree as follows: if the current node is a leaf node, there is nothing to be done — the subscription is already in the tree (line 31); else, we follow *-edges until it is no longer possible, and then add the subscription to the tree (lines 33–35).

B The pre-processing algorithm for equality tests

In this section, we present the pre_process algorithm specialized for the case when subscriptions contain only equality tests. The algorithm is given in Figure 9, and is much simpler than the general one. As in Section A, we assume that the matching tree is represented by T. Just as before, leaf nodes of T contain subscriptions, but unlike before, non-leaf nodes of T do not contain any data; this is because the test associated with a non-leaf node is implicit by the position of the node in the tree. More precisely, if a node v is at level i of the tree, then the test associated with a node in level i is always "examine the contents of attribute i", and the edges leaving v are possible values of attribute i (or it could be the *-edge).

Procedure pre_process(Sub) works as before: it loops over each subscription to be added, and invokes *process_sub*. Function *follow* takes a vertice v and a value r, and returns the node v' obtained by following edge r of node v (if such an edge does not exist at node v, the function adds it to the tree). In procedure process_sub(sub), for each i we set r_i to be the value against which attribute i is being tested in the subscription (if attribute i is not being tested, we set r_i to *). This is done in line 10. Then, we simply succesively call function *follow* on values $r_1, r_2, \ldots, r_K$ (lines 13 and 14). With this, we obtain a leaf node, and then add the subscription to that node (line 15).

References

[1] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. Technical report, IBM, 1998. To appear in the Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Austin, Texas, 1999.

[2] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[4] John Gough and Glenn Smith. Efficient recognition of events in a distributed system. In *Proceedings of ACSC-18*, Adelaide, Australia, 1995.

[5] Object Management Group. Corbaservices: Commo object service specification. Technical report, Object Management Group, July 1998.

[6] Eric N. Hanson, Moez Chaabouni, Chang-Ho Kim, and Yu-Wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of SIGMOD*, pages 271–280, Atlantic City, New Jersey, May 1990.

[7] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communicaiton substrate for fault-tolerant distributed programs. Technical Report TR 91-32, Depart. of Computer Science, The University of Arizona, November 1991.

[8] Brian Oki, Manfred Pfluegl, Alex Siegal, and Dale Skeen. The information bus: An architecture for extensible distributed systems. *Operating Systems Review*, 27(5):58–68, December 1993.

[9] David Powell. Group communications. *Communications of the ACM*, 39(4):50–97, April 1996.

[10] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Australia, September 1997.

```
procedure pre_process($Sub$)
    for each $sub \in Sub$ do process_sub(sub)

procedure process_sub(sub)
    let $sub$ be given by $(t_1 \rightarrow r_1) \wedge \ldots \wedge (t_q \rightarrow r_q)$
    { $q$ is the number of conjunctions in sub }
    if $tree\_root = \perp$
    then $tree\_root \leftarrow \mathbf{new}\ node$; $tree\_root.data \leftarrow t_1$; $found \leftarrow \mathbf{false}$
    else $found \leftarrow \mathbf{true}$
    $v \leftarrow tree\_root$; $i \leftarrow 1$
    while found and $i <= q$ do
        if $v$ is a leaf node then
            let $w, r'$ be such that $(w, r', v) \in T$                              { edge going into $v$ }
            $v' \leftarrow \mathbf{new}\ node$; $v'.data \leftarrow t_i$
            $T \leftarrow T \setminus \{(w, r', v)\} \cup \{(w, r', v'), (v', *, v)\}$
            $v \leftarrow v'$; $found \leftarrow \mathbf{false}$
        else if $v.data = t_i$ then
            if $\nexists w : (t_i, r_i, w) \in T$ then $found \leftarrow \mathbf{false}$
            else let $w : (t_i, r_i, w) \in T$; $v \leftarrow w$; $i \leftarrow i + 1$
        else                                                                         $v.data \neq t_i$
            if $\exists r', w : (v, r', w) \in T \wedge [(t_i \rightarrow r_i) \Rightarrow (v.data \rightarrow r')]$ then $v \leftarrow w$
            else if $\exists w : (v, *, w) \in T$ then $v \leftarrow w$
            else                                                                     { $\nexists w : (v, *, w) \in T$ }
                $v' \leftarrow \mathbf{new}\ node$; $v'.data \leftarrow t_i$
                $T \leftarrow T \cup \{(v, *, v')\}$; $v \leftarrow v'$; $found \leftarrow \mathbf{false}$
    if not found then
        while $i <= q$ do
            $v' \leftarrow \mathbf{new}\ node$; if $i < q$ then $v'.data = t_{i+1}$ else $v'.data = sub$
            $T \leftarrow T \cup \{(v, r_i, v')\}$; $i \leftarrow i + 1$
    else                                                                             { found }
        if $v$ is a leaf node then nop                                               { subscription already in tree }
        else
            while $\exists w : (v, *, w) \in T$ do $v \leftarrow w$
            if $v$ is a leaf node then nop                                           { subscription already in tree }
            else $v' \leftarrow \mathbf{new}\ node$; $v'.data \leftarrow sub$; $T \leftarrow T \cup (v, *, v')$
```

Figure 8: The pre-processing algorithm

```
procedure pre_process($Sub$)
    for each $sub \in Sub$ do process_sub(sub)

function follow($v, r$): node
    if $\exists v' : (v, r, v') \in T$ then return $v'$
    else $v' \leftarrow \mathbf{new}\ node$; $T \leftarrow T \cup (v, r, v')$; return $v'$

procedure process_sub(sub)
    { $K$ is the number of attributes in the schema }
    let $sub$ be given by $(attr_1 = r_1) \wedge \ldots \wedge (attr_K = r_K)$      { we set $r_i$ to "*" if attribute $i$ is not tested in sub }
    if $tree\_root = \perp$ then $tree\_root \leftarrow \mathbf{new}\ node$
    $v \leftarrow tree\_root$
    for $i \leftarrow 1$ to $K$ do
        $v \leftarrow follow(v, r_i)$
    $v.data \leftarrow sub$
```

Figure 9: The pre-processing algorithm for equality tests