Efficient Filtering of RSS Documents on Computer Cluster

Haifeng Liu
Zhejiang University
42 PUBLICATIONS **1,261** CITATIONS
SEE PROFILE

Hans-arno Jacobsen
University of Toronto
220 PUBLICATIONS **3,111** CITATIONS
SEE PROFILE

Efficient Filtering of RSS Documents on Computer Cluster [*]

Haifeng Liu Milenko Petrovic Hans-Arno Jacobsen
Department of Computer Science / Electrical Computing Engineering, University of Toronto
hfliu@cs.toronto.edu, {petrovi, jacobsen}@eecg.toronto.edu

ABSTRACT

RSS filtering is very important today with the increasing amount of information on the Web. There are many tools to aggregate and manipulate content from around the web based on the RSS format. Today clusters are the infrastructure of choice for many large Internet service provider. In this paper we develop algorithms to enable efficient filtering of RSS documents, which is in a graph structured data format, on a computing cluster. We propose indexing and filtering algorithms and suggest several optimizations. The results indicate that the system throughput increases to 400% on a cluster infrastructure over a non-clustered, centralized implementation. In general, we observe that the filtering performance of our algorithms scales linearly in the number of compute nodes in the cluster.

1. INTRODUCTION

The amount of information on the Web is continuously increasing. This fuels the urge of users to stay informed about changes to the content. In general, users want to be updated about daily news headlines of interest to them, be notified when there is a reply in a discussion they participate in, or their favorite web personality has updated her blog etcetera.

The Resource Description Format (RDF) is a graph-structured language by the W3C. RDF Site Summary (RSS) is another metadata language by the W3C for describing content changes.[1] RSS can track many kinds of content changes (e.g., web site modifications, wiki updates, and source code changes in versioning tools.) An *RSS feed* is a stream of RSS metadata that tracks changes for a particular content over time.

The simplicity and extensibility of RSS have made it a corner stone of Web 2.0 as a method for exchanging and accessing application data. It is used in modern, asynchronous browser user interfaces (AJAX [11]), in desktop applications and operating systems (Windows-Vista[2]), in aggregating content from around the web (Yahoo Pipes[3]), as well as for accessing frequently-updated databases (Google base[4], Yahoo traffic[5]/weather[6]). This results in the richness of RSS documents being used. For example google base RSS interface for application developers[7] goes beyond simple, flat, commonly-found RSS structure. It provides a rich (and extensible) schema for a wide range of semi-structured documents. The richness of semi-structured documents being used in turn requires expressive filtering capabilities, since scalability is important for building RSS-based applications.

An *RSS feed aggregator* is a service that monitors large numbers of feeds. For example, Yahoo Piples is a powerful tool to aggregate, manipulate feeds. It allows users to *subscribe* to the content that they are interested in without explicitly specifying which RSS feeds the content is coming from. This is particularly convenient for the user, since the number of RSS feeds that can carry information of interest to the user can be very large. In addition, a user does not have the resources to monitor large number of feeds and hence the user can easily miss information of interest.

It is reported from `www.pewinternet.org` that 6 million people in the US use RSS aggregators. Anecdotal evidence suggests that the way RSS dissemination is currently done can severely affect the performance of websites hosting popular RSS feeds.[8] RSS feed aggregators use pull-based architectures, where the aggregator pulls RSS feeds from a web site that hosts the feed. As the number of feeds on the web proliferates, this architecture is not going to scale. It not only consumes unnecessary resources, but also becomes difficult to ensure timely delivery of updates.

Consequently, a push-based architecture is more scalable in the sense that it provides decoupling of senders and receivers, both in space (i.e., data independence) and time (i.e., asynchronous operation), hence decreases the unnecessary polling traffic between information providers and consumers. The publish/subscribe (pub/sub) paradigm follows such an architecture and is well suited for structuring of large and dynamic systems such as RSS feed filtering, for instance. The pub/sub paradigm has been used extensively in the context of selective information dissemination on the Internet [1, 2, 3, 8, 9, 10], where publishers act as information providers, subscribers as information consumers, and a broker routes relevant publications to interested subscribers. However, most current pub/sub systems are based on predicates for content-based filtering or are designed for filtering tree-structured data such as XML documents [1,

[*]Technical Report, MSRG, University of Toronto, November 2007.
[1]http://web.resource.org/rss/1.0/spec.
[2]http://www.microsoft.com/windows/products/windowsvista/
[3]http://pipes.yahoo.com
[4]http://base.google.com

[5]http://developer.yahoo.com/traffic/
[6]http://developer.yahoo.com/weather/
[7]http://code.google.com/apis/base/
[8]OpenSocial opens new can of worms, Oct 31, 2007

10, 2, 9]. These algorithms are not suitable for filtering graph-structured metadata as required by a RSS feed aggregator. The underlying RDF together with its query language, RDQL, follow a graph-structured model. Both RDF and RDQL are based on a graph-structured data and query model. Figure 1 shows an example of an RDF graph and RDQL query as follows:

```
SELECT * FROM Publications WHERE
(HomePage#325,project,RSS Filter) AND
(RSS Filter,supervisor,Prof A) AND
((RSS Filter,year,?x) SUCHTHAT (?x > 2003))
```

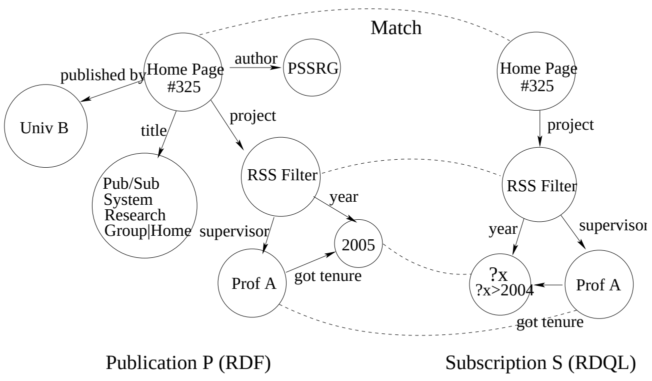


Figure 1: Graph-Structured data, query and match

The plethora of algorithm for filtering tree-structured data and pub/sub-style matching are therefore not enough and cannot be applied. First, different from a tree-structured data and query model (i.e., XML and XPath), a graph generally contains cycles and nodes with multiple parents. Second, there is no concept of root, absolute, or relative level for nodes in a graph. Thus it is difficult to define a starting state for filtering, as required by many of the finite automata-based filtering algorithms for XML/XPath [9]. Third, edges in graph-structured data such as RDF/RDQL have labels that impose semantics used for filtering. However, the edges between two nodes in XML/XPath can only capture structural relationships. For example, the RDQL query "*SELECT ?a, ?b WHERE (?a, http://somewhere/pred1, ?c), (?c, http://somewhere/pred2, ?b)*" can not be expressed by XPath because there is no syntax in XPath to specify the connection edge between two nodes. XPath can only express the query as */?a/?c/?b* which does not realize the required edge semantic.

While RSS filtering has led to wide interest in the information dissemination community, development of a new kind of RSS-based applications would be easier and faster given a scalable infrastructure for filtering and routing of RSS documents on an Internet-scale. Our earlier work [15] runs RSS filtering in a centralized architecture, which cannot meet the needs for an internet scalability. At that scale the challenges of filtering graph-structured data include large query populations, overlap among queries, high data rates and large number of matches. To address these challenges, in this paper we develop algorithms to effectively distribute RSS filtering over a compute cluster, demonstrating improved scalability by increasing the number of compute nodes in the cluster. Essentially, we develop a cluster-based pub/sub system for filtering RSS documents. Compared to our earlier work [15], we develop two indexing algorithms for a cluster-based architecture to enable workload distribution and cluster-based filtering. Furthermore, we change the data structure and proposed an optimized filtering algorithm which speeds up the constraint evaluation as will be explained in Section 4.5.

Cluster-based pub/sub filtering is also different from query processing in a database system, it is subscriptions, not tuples (publications) that are stored in the pub/sub system. Therefore, rather than processing each query one by one, we have to design an algorithm to evaluate all subscriptions (queries) each time a publication arrives. Moreover, the challenge is to amortize evaluation of subscriptions by representing and evaluating shared parts only once. Figure 2 illustrates the architecture of the pub/sub cluster. The workload (publications and subscriptions) arrives through the front-end server. For subscriptions, the server decides whether to keep them in the server or distribute them to one of back-end nodes based on the subscriptions' relations by building up an index to remember where each subscription is stored. For publications, the server distributes the arriving publication stream to a collection of back-end nodes according to the subscription index.

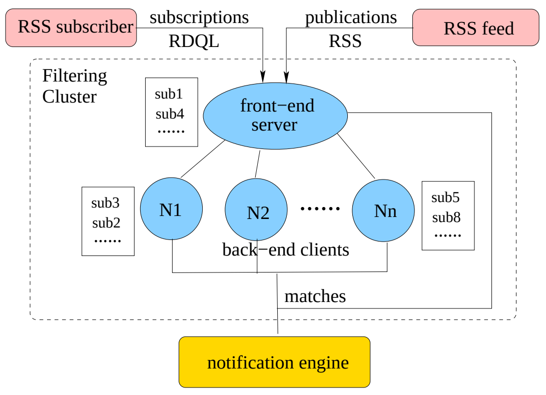


Figure 2: Publish/Subscribe Cluster

The key problem in designing a cluster-based pub/sub system is how to efficiently distribute the workload (subscription indexing and publication filtering) among the cluster nodes. A simple approach of randomly distributing subscriptions among all nodes in the cluster, may result in performance degradation to the extent that it is worse than a centralized filtering system. This can happen since the matching on the cluster involves additional time for index lookup and communication. In the worst case, all matched subscriptions could end up collocated on the same node. In this case the time for communication and index lookup is pure overhead. A good way for workload distribution is that can achieve the maximum parallel filtering. From our earlier work [15], we know that the filtering on a single machine depends on the number of matches. Therefore, the goal of workload distribution is to distribute subscriptions that are possibly matched by the same publication to different machines. Such subscriptions can be identified based on the relations among subscriptions and matching statistics. In this paper, we will propose algorithms based on these two metric to effectively distribute subscriptions, while prevent the worst case mentioned above.

The contributions of this paper are:

1. We developed a *pipelined filtering* algorithm for processing graph-based data and queries on a compute cluster. The algorithm exploits structural differences among subscriptions to quickly identify potentially matching candidate subscriptions. It effectively handles structurally similar workloads by pre-filtering based on graph-node value constraints. Pipelined execution increases the throughput on a multi-processor and compute cluster.

2. We developed two *indexing* algorithms that complement pipelined filtering by effectively partitioning subscriptions into disjoint sets in order to reduce filtering time at each cluster node. *Containment* partitions subscriptions based on semantic similarity, while *merging* partitions subscriptions based on run-time access frequency.

3. We have developed and experimentally evaluated the performance of the filtering system in a cluster environment and compared it to a centralized system. Our results indicate that throughput increases to 400% for a fixed workload of 100,000 complex subscriptions on a cluster infrastructure over a non-clustered, centralized implementation. In general, we observe that the filtering performance of our algorithms scales linearly in the number of compute nodes in the cluster.

2. OVERVIEW OF LANGUAGE MODEL

We first give an overview of the data model, which is the graph-based subscription and publication representation language introduced in [15]. We use the same data model, but in a cluster-based architecture.

A *publication* is represented by a labeled directed graph. We use RDF semantics to interpret the graph as a set of triples *(subject, property, object)*. Each triple is represented by a node-edge-node link where the nodes represent *subject*s and *object*s, and edges between them represent *properties*. Figure 1 illustrates a publication P of a web page about the RSS Filter project under the supervision of Prof A.

A *subscription* is a directed graph pattern specifying the structure of the publication graph with optional constraints on some vertices. A subscription is a set of 5-tuples (*subject, property, object, constraintSet (subject), constraintSet (object)*), which is represented as a link starting from the *subject* node and ending at the *object* node with the *property* as its label. A `constraintSet` represents relational constraints, which is a predicate of the form $(?x, op, v)$, on *subject* and *object* values. For example, Figure 1 illustrates a subscription that specifies interest in a web page which is about the RSS Filter project supervised by Prof A and published after the year 2004. The same subscription can also be represented declaratively by RDQL.

We denote G_P as the publication graph and G_S as the subscription graph pattern. The matching semantics is then defined as verifying whether G_S is embedded in G_P. Graph pattern G_S is *embedded* in G_P if every node in G_S maps to a node in G_P such that all constraints of G_S are satisfied. The dotted line in Figure 1 shows an example of matching where publication P matches subscription S. The single matching problem between one subscription and one publication can be considered as a graph isomorphism problem. However, as we will explain in detail in Section 4, to improve the filtering efficiency, all subscriptions are stored in a union graph G_M. With this representation, the filtering problem between a publication against the union graph G_M is not a graph isomorphism problem anymore, but is defined as determining all *subgraphs* of G_M that are isomorphic to *some subgraph* of G_P.

3. SYSTEM ARCHITECTURE

Going beyond filtering on a single machine requires two steps: (1) determining how to partition the subscriptions among machines of a cluster to maximize parallel filtering and (2) extending the centralized filtering system to a multi-partition case. The first step requires an indexing algorithm to distribute subscriptions and the second step requires a filtering algorithm to distribute publications.

3.1 Subscription Partitioning

In a clustered system, we organize a set of machines into a two-level network as shown in Figure 2. One node is selected as the *server*. The remaining machines are the *nodes* that connect to the server directly. The server is responsible for partitioning subscriptions among the back-end nodes in order to maximize throughput. All publications enter the system via the server.[9]

The partitioning of subscriptions is done dynamically based on the workload itself and also the recent filtering statistics. In particular, the system tries to prevent filtering hot-spots by distributing the filtering among a number of nodes. The server acts as an *index* for a subscription partition. The index only stores the most general subscriptions in order to achieve parallel filtering. The index is created based on the semantics of the subscriptions and their access frequencies. Subscriptions that are semantically related via *containment* relationship, are always partitioned into disjoint sets. Similarly, subscriptions that are frequently accessed together (via similar publications) are also partitioned into disjoint sets. The intuition behind this is that subscriptions that are related either using *containment* or using *similarity* of publications are most likely to be part of the same filtering result set. Consequently, we distribute the job of determining those results among many nodes in order to achieve *concurrent filtering of the matching result set as identified by similarity or containment.*

Containment Partitioning: Given two subscriptions S_1 and S_2, S_1 *contains* S_2 if and only if all the publications that match S_2 also match S_1, which is denoted by $S_1 \succeq S_2$. When the server receives a subscription S, S will be inserted into the subscription index if and only if there is not another subscription S' in the index that contains S. If there is another subscription S' in the index that contains S, S will be distributed to a back-end node, and vice versa. The server will not miss any publication that matches S even though S is not stored in the index, since it receives all publications that match S' and the publications that match S are a subset of the publications that match S'. Further filtering can be executed on the back-end nodes in parallel.

[9]The index at the server can be replicated to multiple machines if the index lookup becomes the bottleneck. In this case, one of the replicas acts as the master and the others as slaves. The master is responsible for doing index updates, while the slaves only do index lookups.

Access Frequency Partitioning: Subscriptions that are not in a fully containment relation may still relate to each other if they are matched by same publication. Subscriptions that frequently matched together by the same publication, can be merged into a new subscription. This process is called merging. The new subscription will replace the individual subscriptions in the index, while the individual subscriptions are distributed to disjoint back-end nodes so that they can be evaluated concurrently. A merged subscription S_M, based on merging subscription S_1 and S_2, *contains* both S_1 and S_2, which is represented as $S_M \succeq S_1$ and $S_M \succeq S_2$.

3.2 Filtering on a Cluster

The publications are processed as follows. For each incoming publication at the server, check if there are *matching* subscriptions. If there are, deliver the publication to the subscribers. Note that some of the matching subscriptions are part of the subscription index as created by containment and merging, thus publication will be delivered to back-end nodes for further filtering.

The system throughput on a pub/sub cluster can be increased, compared to a centralized system, by the *parallel* filtering executed at the back-end nodes. The throughput is further increased by the ability to *pipeline* the filtering of the two stages: index at server and filtering at the back-end nodes. This is possible because of the stateless operation of partitioning and filtering operations.

4. DATA STRUCTURE AND ALGORITHMS

The goal in designing a cluster-based pub/sub system is to partition the subscriptions that are possibly matched together into disjoint sets and delivered to different back-end nodes so that filtering can be processed concurrently and maximum throughput can be achieved. In this section, we first describe the data structure to store subscriptions, then presents two indexing algorithms and a filtering algorithm to fulfill subscription partition and pipelined filtering on the cluster. The indexing algorithms identify those subscriptions that are possibly matched together: one exploits semantic relationships among subscriptions via *containment*, and the other exploits the run-time relationships via access frequency during filtering.

4.1 Data Structure

To avoid evaluating subscriptions one by one, we store subscription graphs in a way that exploits commonalities between them and filters publications efficiently. To exploit overlap between subscriptions we integrate all subscriptions into a single graph. We denote the graph containing all subscriptions by G_M (subscription matrix). All the following algorithms are designed based on G_M. Indexing algorithms will explore the relations between a subscription graph G_S and G_M and the filtering algorithm will explore the relations between a publication graph G_P and G_M. As an example, Figures 3 shows how two subscriptions are combined into one subscription matrix graph, G_M.

We use a two-level hash table to store G_M. The first-level hash table contains all edge labels. Each entry E_i is a pointer to a second-level hash table. The second-level hash table contains all the pairs of vertices between whom the edge has label E_i. Each entry of the second table points to a list of subscriptions that contain the edge between this

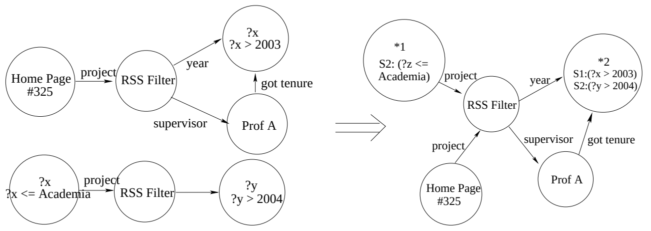


Figure 3: Subscription Matrix

pair of vertices. If any vertex is a variable, the subscriptions are classified according to constraints associated with this variable and are ordered into a table, as shown in Figure 4. For example, for integer constraints there are four disjoint operators: $=$, $>$, $<$ and $\neq$. In the constraint table, we keep four lists for each of the four operators. An entry in each of these lists maps a value to a set of subscriptions that have a satisfying constraint. For example, an entry in the "$>$" list contains a key value 10 and subscriptions S_2 and S_5. This means that S_2 and S_5 contain a common edge $E2(A, \star)$ with the same constraint "$\star > 10$". The four lists are ordered according to the values of keys in an increasing order.

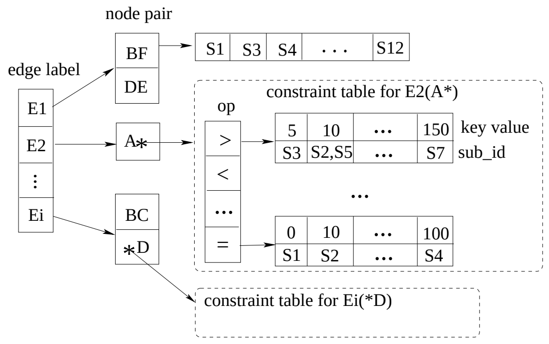


Figure 4: Data Structure

4.2 Precise Indexing Using Containment

Based on the graphical representation of subscriptions, the definition of *containment* can be stated as follows: for two subscriptions S_1 and S_2, S_1 *contains* S_2 iff for each edge $e_1(v_1, w_1)$ in S_1, there is an edge $e_2(v_2, w_2)$ in S_2 that $e_1.label = e_2.label$ and v_1 contains v_2 and w_1 contains w_2. The node containment relation is defined as v_1 contains v_2 iff $v_1.label = v_2.label$ or v_1 is a variable. Subscriptions with containment relations form a containment tree. Figure 5 shows an example of subscription containment relations where S_2 contains S_1, and also an example of a containment tree formed by five subscriptions on the left. Only the root subscription is stored in the index (at the server node). The descendant subscriptions (contained by the root) are partitioned into disjoint sets and distributed to different back-end nodes. In other words, there is no containment relation among the subscriptions in the index.

When a new subscription S comes into the server, if it is contained by other subscription in the index, S will be sent to a back-end node. If there are other subscriptions contained by S, these subscriptions will be partitioned equally

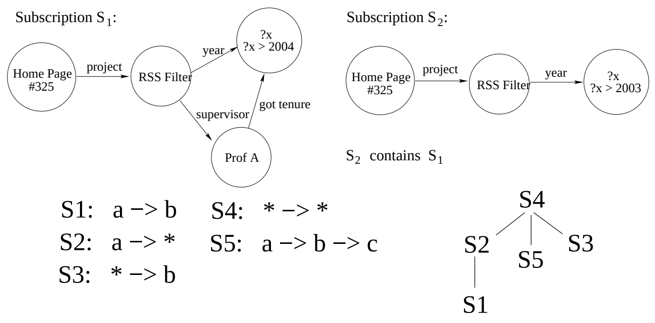


Figure 5: Example of Subscription Containment and Containment Tree

into disjoint set and sent to each node. We propose an algorithm that, for each new subscription, determines all possible containment relations by accessing the subscriptions in the index only once.

The `containmentChecking` algorithm is executed for each new subscription that is to be added to the index. There are two stages in the `containmentChecking` algorithm. First, we use edge labels to filter out unsatisfied subscriptions. Only those subscriptions that contain all edges of S or whose edges are contained in S will be processed in the next stage. Then for each edge of these subscription candidates, we check the node containment relation.

Algorithm *containmentChecking*(G_S)
Output: boolean *contained*; a set R of subscriptions contained by G_S

```
1.  $SubSet = \emptyset$
2.  for each edge $e(v,w) \in G_S$
3.      for each $e2(v2,w2) \in G_M$ where $e2.label = e.label$
4.          for each $S_i \in e2.subs$
5.              $S_i.counter++$
6.              $e.bindingTable += (v2, w2)$
7.              $e2.bindingTable += (v, w)$
8.              $SubSet = SubSet + S_i$
9.  $R = \emptyset, contained = false, containing = false$
10. for each subscription $S_i \in SubSet$
11.     if $S_i.counter++$ satisfied containing condition
12.         $b = e.bindingTable \mid e \in S$
13.         for every edge $e \in S.edges$
14.             $b = b \bowtie e.smEdge.bindingTable$
15.         if $b.isContained()$
16.             $R = R \cup S_i$, $containing = true$;
17.     if $S_i.counter++$ satisfied contained condition
18.         $b = e.bindingTable \mid e \in S_i$
19.         for every edge $e \in S_i.edges$
20.             $b = b \bowtie e.smEdge.bindingTable$
21.         if $b.isContained()$
22.             $contained = true$, $R.add(S_i)$
23.             return
24. return $R$ and $contained$
```

In the second stage, the containment relation is computed for each subscription in the candidate set *SubSet*. From the definition of the containment relation, we know that if S_1 contains S_2 this implies that $S_1.edges \leq S_2.edges$. Therefore, for each subscription S_i in *Subset*, we first check the edge counter condition. If $S.edges \geq S_i.edges$, we join the binding tables of edges belonging to S_i. If $S_i.edges \geq S.edges$, we join the binding tables of edges belonging to S. If the result table is not empty, and the bindings in the table are covered by S_i or S, then a containment relation is detected.

Figure 6 illustrates an example of checking containment

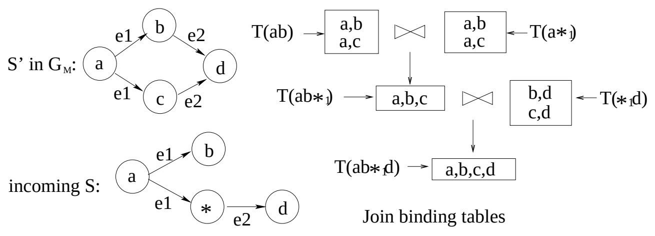


Figure 6: Computing Containment Relation

relations between S and S'. From edge $e1(a,b)$ in S, we get two edges in S' with the same label $e1(a,b)$ and $e1(a,c)$. Then we put (a,b),(a,c) into the binding table of $e1(a,b)$ in S. From edge $e1(a,\star)$ in S, we obtain $e1(a,b)$ and $e1(a,c)$ in S' and put (a,b),(a,c) into the binding table of $e1(a,\star)$ in S. Similarly, (b,d) and (c,d) are put into the binding table of $e2(\star,d)$. Finally, we get three tables as shown in Figure 6. Joining of these three tables produces one entry (a,b,c,d) in the result set. Since (a,b,c,d) is contained by $(a,b,\star,d)$, we conclude that S' is contained by S.

Complexity Analysis: Different from general query containment problem for conjunctive queries, where the containment relations are indentified when all queries are given, our `containmentChecking` algorithm checks containment relation for each newly arriving subscription. As we mentioned earlier there is no containment relation among the subscriptions in the index. Thus, we only check the containment relation between the newly arriving subscription and other subscriptions in the index. The `containmentChecking` algorithm consists of two stages. The time for the first stage depends on the number of edges in the subscription, and the second stage depends on *SubSet* size. Overall, the time to evaluate containment among all subscriptions is $O(m)+O(n)$ where m is the number of edges in the new subscription, n is the number of subscriptions in *SubSet* which is approximately equal to the number of contained subscriptions. Since $m << n$, the overall time to check for the containment relations is linear in the number of contained subscriptions, $O(ratio_{overlap} * number\_of\_subscriptions)$.

4.3 Imprecise Indexing Using Merging

The second indexing algorithm identifies those subscriptions that are frequently part of the same result set at runtime. Such subscriptions can be determined by analyzing run-time filtering statistics based on the matched publication result set in which they appear. The identified subscriptions are removed from the index and partitioned into disjoint sets at the back-end nodes (for concurrent evaluation). A new subscription is added to the index that represents the *merger* of the removed subscriptions. The information about the sets to which the removed subscriptions have been placed is recorded with the new subscription.

To simplify the representation of the merged subscription, we define the merger as the intersecting subgraph of two subscriptions where $G_M = G_{S_1} \cap G_{S_2}$. In other words, G_M contains a set of edges that belong to both G_{S_1} and G_{S_2}. The merging relation also forms a merging tree, the root is the merger and the leaves are the merged subscriptions. Only the merger exists in the server as an index.

The indexing algorithm based on merging is called imprecise indexing because the merging rule introduces false positives. For example, the publication shown in Figure 7 matches the index subscription (merger), but it does not match any of the original two subscriptions.

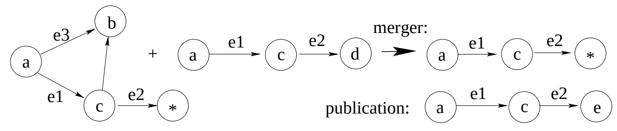


Figure 7: Merger of Subscriptions

When a set of subscriptions are frequently matched together, the publications matching them are similar and contain a common part. Based on this observation, we determine the set of subscriptions for merging based on the *similarity of publications*. Given two publications P_1, P_2, the *similarity* between P_1 and P_2 is defined as $\frac{|P_1.edges \cap P_2.edges|}{|P_1.edges \cup P_2.edges|}$.

The server keeps statistics on subscription access frequency through result set membership, as identified by a publication. We define a *publication summary* as an abstract publication that contains the intersecting part of similar publications. A set of publication summaries are stored in a publication matrix E_M (similar to subscription matrix). The statistic collection is done separately from the matching process. It is performed offline by duplicating the publication stream (or some sample of it) and the index, as that is the only information required for the collection process. While the performance of the collection process is not important, we assume it does not require real-time response, and that it can even be done off-line. The collection process algorithm is described in Algorithm *statCollection*.

```
Algorithm statCollection(p)
1.  p.counter = p.matchedSubs.size
2.  find out p' in E_M such that p' has the largest similarity
    (larger than threshold Th) with p
3.  while (p' != null)
4.      $p_{new}$ = merge(p, p')
5.      $p_{new}.counter = p.counter + p'.counter$
6.      $p = p_{new}$
7.      find out p' in $E_M$ such that p' has the largest
        similarity (larger than threshold Th) with p
8.  insert p into $E_M$
```

In our approach, the policy is to perform index merging updates when the size of the index reaches a certain threshold C. We first choose a publication summary p from E_M with the largest access frequency, then match p against the subscription matrix G_M to get a set of subscriptions, *mergeSubs*. These subscriptions will be removed from the index, and a new, merged, subscription is added to the index. *mergeSubs* are then partitioned into disjoint sets (and distributed to back-end nodes).

Complexity Analysis: The merging process takes out subscription from the *mergSubs* set one by one and merges them until *mergSubs* is empty or the index size decreases to below the capacity threshold C. Therefore, the time complexity depends on $max(mergeSubs.size, index.size - C)$.

4.4 Index Maintenance

In a pub/sub cluster, subscriptions are spread out among all the nodes. When an unsubscription message is received, we need to modify the subscription index so as to ensure the correctness of filtering. Next, we will discuss the operation of subscription deletion for each indexing algorithm.

When receiving an $unSub(S)$ message, it is simple if S is not in the index. We just forward an $unSub(S)$ message to all nodes and the node which contains S will delete it. Deleting a subscription from a node will not affect publication filtering. However, if S is in the index, we need to maintain the containment and merging relations to ensure correct filtering.

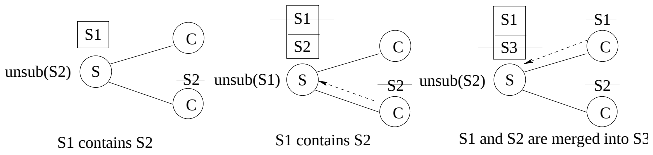


Figure 8: Deleting Subscriptions

When receiving an $unSub(S)$ message, we need to update the subscription index, containment trees and merging trees. For the subscription index, delete S from the index if it exists; otherwise forward $unSub(S)$ to all nodes, which will delete S if they know about it.

If S is in a containment tree, S can be the root or inner (or leaf) node. If S is the root, $S.children$ (contained subscriptions) will be recovered into the subscription index. If S is an inner (or leaf) node, the index remains unchanged, delete S from the node. The containment tree is updated by putting $S.children$ directly under $S.parent$.

If S is in a merging tree, S can only be the leaf node since only leaves are real subscriptions and root (or inner) nodes are constructed mergers. In this case, the parent of S will be decomposed into the original subscriptions. The merger should be deleted from the index and the other subscription will be inserted into the index. Figure 8 shows examples to *unSub* a subscription which has a containment relation.

4.5 Pipelined Filtering Algorithm

For each incoming publication on a cluster, the server does a lookup in the subscription index to check if there are matching subscriptions. For each matched subscription s, the publication is forwarded to the appropriate subscriber. Note that one or more of the subscribers could be other cluster nodes, in which case, the publication is forwarded to the appropriate back-end node for the second step of filtering. While the back-end nodes are filtering the publication, the server starts filtering the next publication. This pipelined processing enables higher throughout than a single machine filtering. The pseudocode is shown in *filtering*.

```
Algorithm filtering($G_P$)
1.  $R_{matched} = \emptyset$
2.  $R_1$ = server.match($G_P$)
3.  for each $s \in R_1$
4.      node_set O = server.index.get(s)
5.      if (O! = $\emptyset$)
6.          for each $o \in O$
7.              $R_{matched} = R_{matched} \cup$ o.match($G_P$)
8.      else  $R_{matched} = R_{matched} \cup s$
9.  return $R_{matched}$
```

In either the index server or back-end nodes, given all stored subscriptions, G_M, and a publication, G_P, the matching problem is to identify all the subgraphs, G_{S_i} (representing a subscription S_i) in G_M which are matched by G_P. To match a publication against G_M, first, for each edge in the publication, we check all the individual matched subscription edges in G_M. Then we use natural *join* operation for connectivity checking between edges to determine the satisfying bindings for variables and evaluate the constraints sequentially. The join operation consumes the majority of the matching time. To avoid this expensive operation as much as possible, we push the constraint evaluation forward before the join operation. There are two advantages. First, the number of subscriptions remaining for join operation decreases significantly. Second, the subscriptions whose constraints are satisfied can be retrieved faster without evaluating the constraints for each subscription sequentially. Algorithm *match* is the procedure for matching publications against stored subscriptions on a single node. There are two stages in the matching process. In the first stage, for the publication edge $e(v, w)$, the potentially matched edges in G_M are $e'(v', w')$ where $e.label = e'.label$ and $(v' = v || v' = \star)$ and $(w' = w || w' = \star))$. For each $e'(v', w')$, the constraint-satisfied subscriptions C include three parts: the list of subscriptions that have no constraints on either v' or w'; the subscriptions whose constraints on variable v' are satisfied by publication value v; and the subscriptions whose constraints on variable w' are satisfied by publication value w.

With a publication value v, the following satisfied subscriptions can be retrieved from the constraint table by traversing each of the four lists: In the "=" list, the subscriptions with a key value v; in the ">" list, the subscriptions with key value $v'(v' < v)$; in the "<" list, the subscriptions with key value $v'(v' > v)$ and in the "≠" list, all other subscriptions except those with key value v. Since the lists are ordered, the traversal can be stopped at the first non-satisfied lookup, hence speeding up the matching.

```
Algorithm match(G_P)
1.  $SubSet = \emptyset$
2.  for each $e(v, w) \in G_P$
3.      get a set of matched edges $E$ from $G_M$
4.      for each $e'(v', w') \in E$
5.          get a set of subscriptions $C$ whose constraints
            are satisfied
6.          for each $s \in C$
7.              $e'.bindingTable += (v, w)$;
8.              $SubSet = SubSet + s$
9.  $result = \emptyset$
10. for each subscriptions $s \in SubSet$
11.     join binding table $e.bindingTable$ for each edge $e \in$
        $s.edges$
12.     if the final binding table is not empty
13.         result = result + $s$
```

Complexity Analysis: The matching algorithm consists of two steps. The time for the first step depends on the number of edges in the publication. The time for the second step depends on the size of *SubSet* and the cost of join operation for each subscription in *SubSet*. The computation of a join operation depends on the size of the binding table of the variable. Suppose the average number of variables in a subscription is α and the average number of possible bindings for a variable is β. Then the cost of the join operations for one subscription to determine satisfying bindings is β^α. Overall, the matching time to evaluate a publication against all subscriptions in G_M is $|P.edges| + \beta^\alpha * |SubSet|$. In practice, α, β and $|P.edges|$ are small compared to the large number of subscriptions that need to be filtered. And *SubSet* is approximately equal to the number of finally matched subscriptions. The overall matching time is thus linear in the number of matched subscriptions: $O(ratio_{match} * number\_of\_subscriptions)$.

5. EVALUATION

In this section we evaluate our approach to demonstrate that the cluster-based system achieves large scalability and high system throughput. We run experiments to evaluate the performance of our two indexing algorithms and effect of filtering on a cluster compared to a single matching engine. We emulate the cluster-based architecture by running one machine as the server and $2 \sim 6$ other machines as the back-end nodes, which communicate with the server by sockets. As we show later, even with a modest size cluster, the performance improvements over a centralized solution are obvious. We are using a synthetic workload so that we can independently examine various aspects of the approach. The workload parameters used in the experiments are publication size $Size_P(30, 90)$, subscription size $Size_S(5, 10)$, number of subscriptions $N_{sub}(100, 000)$, number of matched subscriptions $N_{matches}(100)$, number of variables (i.e., stars) in one subscription $N_{stars}(2)$, number of subscriptions that containing variables $N_{sub^*}(90, 000)$, and the ratio of overlap among subscriptions $overlap_s(50\%)$. The numbers in brackets are the default values for these parameters. $Size_P$ and $Size_S$ are determined by the number of nodes and the number of edges in the publication graph and the subscription graph, respectively. We generate the test workload using the above parameters. For each experiment, we vary one parameter and fix the others to their default values.

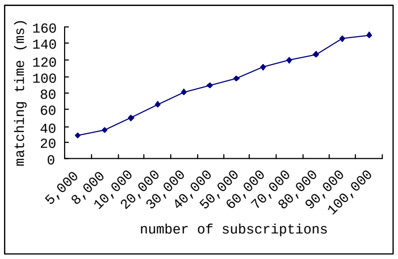
We generate the test workload using the above parameters. For each experiment, we vary one parameter and fix the others to their default values.

5.1 Indexing Performance

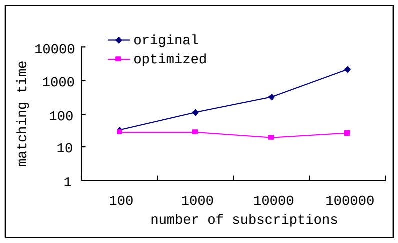
To evaluate the performance of the indexing algorithms, we measure the subscription index size on the server, subscription insert time, index lookup time and merging time. The subscription index size is the number of subscriptions stored on the server. The insertion time is calculated as the average over 1000 subscriptions inserted. The index lookup(match) time is computed as the average over 100 publication matching operations on the server. For merging, we also evaluate the number of false positives introduced by an imprecise merger. The following factors influence the performance of the algorithms: number of total subscriptions received by the cluster), ratio of subscription overlap and merging percentage. We examine the effects of these factors. For each experiment, we vary one parameter and fix the others to their default values as specified in Table 1.

We first evaluate the time for index lookup on the server. Figure 9(a) shows the time to find all matched indexing subscriptions for a publication given a set of subscriptions. Since we fixed the match ratio, the number of matches increases linearly with the number of subscriptions, so does the matching time.

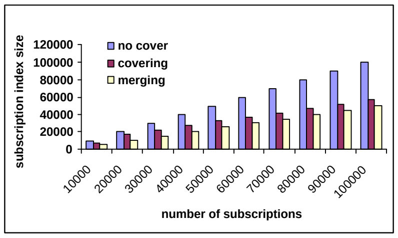
Next we compare our index lookup algorithm *match* with GToPSS described in [15]. Our algorithm is especially useful for the workload where most subscriptions have same graph structure but different constraints. In the experiment, we fix the number of matched subscriptions (both structurally and constraints) and vary the total number of subscriptions that have same structure but different constraints. Figure 9(b) shows that the matching time of GToPSS increases to $2500ms$ when the number of subscriptions increase to $100K$. This is because almost all subscriptions are involved in binding checking and constraint evaluation. However, the



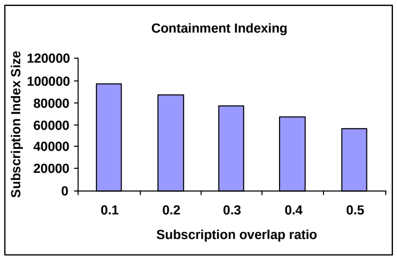
(a) index lookup time vs. #subs



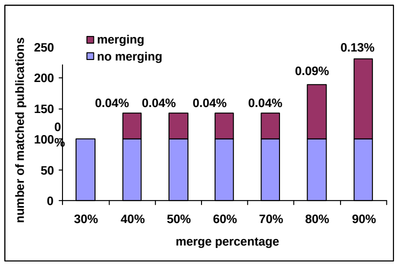
(b) our matching vs. GToPSS



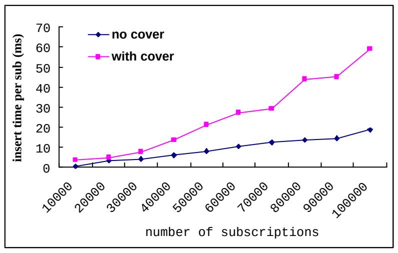
(c) server index size vs. #subs



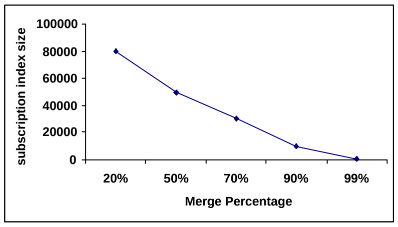
(d) index size vs. overlapping ratio



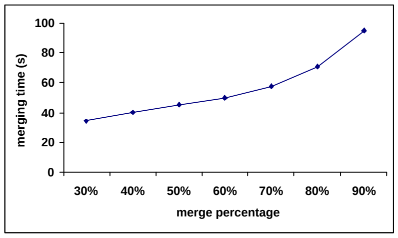
(e) false positive vs. merge percentage



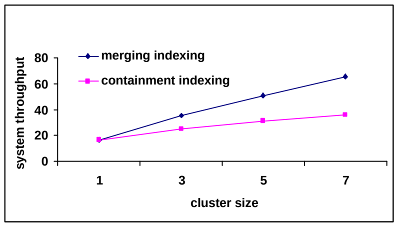
(f) insertion time vs. #subs



(g) index size vs. merge percentage



(h) merging time vs. merge percentage



(i) throughput vs.cluster size

Figure 9: Experimental results

matching time of our algorithm is constant to around $20ms$ and only depends on the number of finally matched subscriptions.

In a cluster-based publish/subscribe system, the size of the index on the server plays an important role in filtering. We examine this metric for centralized architecture and cluster-based architecture with different indexing algorithms. Figure 9(c) shows that the subscription index size on the server is reduced to 50% by indexing.

Figure 9(d) shows the subscription index size for the containment indexing algorithm when the ratio of overlap among subscriptions varies. The workload contains 100,000 subscriptions. The larger the ratio of overlap in subscriptions, the more containment relations existed among the subscriptions; and the more subscriptions are distributed to backend nodes; thus the smaller the subscription index size. The subscription index size is approximately equal to $number\_of\_subscriptions * (1 - ratio_{overlap})$. The subscription index size resulting from merge indexing mainly depends on the merging threshold set in the merging algorithm, not on the ratio of overlap. For a fixed workload, the subscription index size for the merging algorithm varies according to the merge percentage we set in the algorithm. Figure 9(g) shows this relation. Subscription index size decreases to 629 when the merge percentage increases to 99%. For some subscriptions without similarity, we just leave them there, since the merging result will be an empty subscription. Therefore, The final index size is larger than $number\_of\_subscriptions * (1 - merge\_percentage)$. [10]

The merging of subscriptions may result in the introduction of false positive (i.e., unmatched publications that are

[10]We usually set two thresholds for merging, a higher threshold is used to trigger merging and a lower threshold to control the index size and the difference between them gives the space for the index to keep increasing without the need to trigger merging frequently.

forwarded into back-end nodes, but do not actually match the individual subscriptions. They are not forwarded to real subscribers, since they are filtered out by the back-end node.) False positives result in extra time of filtering and thus decrease the system throughput. We experiment the impact of false positives based on a workload of 10,000 subscriptions and 20,000 publications. Figure 9(e) shows that there are only 0.13% false positives for a 90% merge percentage.

In a cluster-based architecture, the publication filtering can be executed for disjoint subscription sets concurrently, thus improve the filtering efficiency. The price for this improvement is the indexing time consumed by the indexing algorithms to check the relations among all subscriptions and to perform subscription partitioning. To evaluate this trade off, we measure the average insertion time for one subscription. In Figure 9(f), the subscription insertion time with containment indexing is larger than that without any indexing, due to the required containment checking computations. Also the average insertion time for one subscription grows with the increase in the number of subscription, which validates the time complexity analysis of the containment indexing algorithm for constant ratio of overlap.

Finally, we evaluate the merging algorithm. As we mentioned in Section 4.2, the merging algorithm is dependent on the length of the merge candidate list and the difference between the size of subscription index and the capacity threshold. Thus the merging time in Figure 9(h) increases with the increase of merging percentage.

5.2 Filtering Performance on a Cluster

The goal of using a cluster-based architecture for filtering is to increase the system throughput. The following parameters are factors that influence the system throughput: number of nodes in the cluster, indexing algorithm, merge percentage. We examine the effect of these factors. In the experiments, we use a workload of $100K$ subscriptions.

System throughput depends on the degree of filtering parallelism. The parallelism degree depends on the number of nodes in the cluster. Figure 9(i) shows this effect. In the experiments, we fixed the workload at $100K$ subscriptions and there are 0.5% matches per publication. We run the same workload on the clusters with different number of nodes using containment indexing and merging, respectively. The number of back-end nodes equals to $cluster\_size - 1$. $cluster\_size$ equals 1 represents a centralized architecture, there is only one node which contains the entire workload. The indexing algorithms only work when $cluster\_size > 1$. In the experiment, we fixed the merge percentage for the merging algorithm at 70%. The figure shows that the system throughput increases with the increase in the number of nodes and the merging algorithm achieves a higher throughput than the containment indexing algorithm. For a 7-nodes cluster, the throughput increases to 400% using the merging algorithm and increases to more than 200% using the containment indexing algorithm.

In a cluster-based architecture, if the number of nodes are fixed, the more subscriptions are distributed from server to back-end nodes, the larger the parallelism degree and the larger the system throughput. In the merging algorithm, the merge percentage controls how many subscriptions will be merged, which is how many subscriptions will be distributed to back-end nodes. Figure 10 shows the effect of the merge percentage on system throughput for a workload with $100K$ subscriptions and among which there are 1% matches. We compared the throughput for 3 types of architectures: single filtering engine, cluster filtering engine with 2 back-end nodes and cluster filtering engine with 4 back-end nodes. The figure validates the above argument. With 90% merge percentage, the throughput of a cluster-based architecture is 4 times of that in a centralized architecture. Furthermore, we can see that the benefit of the cluster filtering system increases with the increase of merge percentage.

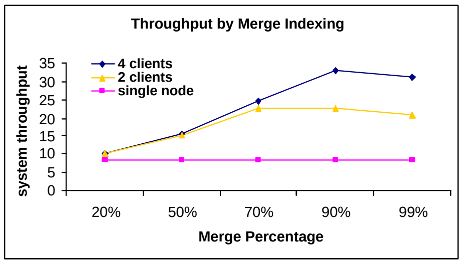


Figure 10: Throughput on Cluster Filtering

6. RELATED WORK

The use of the pub/sub communication model for selective information dissemination has been studied extensively. Existing pub/sub systems [10, 1, 8, 5] use attribute-value pairs to represent publications, while conjunctions of predicates with standard relational operators are used to represent subscriptions. Systems such as described in [2, 9] process XML publications and XPath subscriptions. XTrie [6] proposes an index structure that supports filtering of XML documents based on many XPath expressions. The approach is extensible supporting patterns including constraint predicates. Gupta *et al.* [12] show how to process XML stream over XPath queries including predicates. All these approaches aim at filtering XML documents which is tree-based data and do not support the filtering of general graph-structured data such as RSS documents, which is the main motivation of our work.

Our matching algorithm is designed to tackle the challenges of graph-structured data filtering. If applied to filtering tree-structured data,[11] it incurs additional cost compared to other XPath matching algorithms such as [2, 9]. In our algorithm, the edges are matched individually in the first step without considering the connectivity between them. More edges, and thus more candidate subscriptions are stored for further evaluation. We cannot avoid this since there is no concept of root node in a graph, so there is no unified starting state for all graph queries. The other cost is that we need the join operations in the second step to determine the final structurally matched subscriptions.

Racer [13] is a pub/sub system based on a description logics inference engine. Since OWL is based on description logics, Racer can be used for RDF/OWL filtering. Racer does not scale as well as our system. Its matching times

[11] A tree is essentially a graph. Therefore, our algorithms could be applied to the filtering of XML against XPath. However, the filter language supported by our algorithm does not include the descendant operator, which is supported in XPath.

are in the order of 10s of seconds even for very simple subscriptions [13], however, it offers more powerful inference capabilities. Chirita *at el.* and Cai *et al.* [7, 4] design a pub/sub system supporting metadata and propose a query language based on RDF. Both approaches are based on peer-to-peer network abstractions and express queries in a triple pattern, rather than a graph-based language as central to our model. Furthermore, our model demonstrates greater scalability with a demonstrated throughput of millions of queries per second as compared to the throughput of 250 queries per second reported by RDFPeers [4].

SemCast [14] present a semantic multicast approach that split the incoming data streams based on the overlapping of their contents. In SemCast, three channels are generated from two overlapping profiles. One contains the content common to both and the other two are the noise channels that carry content assigned to only one of them. This approach can eliminate the need for filtering at interior brokers, but it wont reduce the amount of messages routed in the network. And also compare to our approach, SemCast does not support imperfect merging which achieves a large amount of decrease of network traffic as shown in our experiments.

Yi-Min Wang *et al* [16] proposed two approaches to subscription partitioning and routing, one based on partitioning the publication space for equality predicates and the other based on partitioning the subscription set for range queries. For partitioning the subscription set, similar subscriptions are grouped together using an R-tree and assigned to one machine. Our algorithms are for graph-structured data where the semantics of containment and similarity is different from range queries. The measurement for similarity is defined based on not only graph structure but also matching statistics. Moreover, our partitioning approach is to separate similar subscriptions into different machines rather than group them together so that the parallel matching can be achieved and the system throughput is increased.

7. CONCLUSIONS

The use of RDF as a language for representing metadata is growing. Applications such as RSS and content management are exhibiting use patterns that current web-based systems are not designed for. Cluster-based, data-centric, publish/subscribe-based systems, as described in this paper, are a very good fit for such applications. The system we presented in this paper is able to achieve high throughput for very complex subscriptions by distributing the matching on a computing cluster. The containment and merging algorithms we developed are essential to efficiently partitioning of subscriptions among nodes of a cluster. Our experiments show that the throughput scales linearly with the number of cluster nodes. For the 2-clients architecture, the throughput is twice of that of a centralized architecture; while the throughput increases to 400% in a 4-clients architecture. Based on our experiment results, we can analyze that the index size would be $150K$ by applying imprecise indexing algorithm when the total subscriptions are numbered in millions. Since the filtering algorithm can process $150K$ subscriptions in a single node, we can support 1 million subscriptions on a computer cluster. Our experiments are designed to test the scalability limits of the system. In practice, we expect subscriptions to be simpler (i.e., have smaller number of edges and variables) than the ones used in our experiments and be more similar to each other, hence it is likely that there will be even more containment. This observation is based on the expectation that RSS will follow Web content popularity clustering as evident from anecdotal evidence today. Applications exhibiting such interest clustering would benefit the most from our approach.

8. REFERENCES

[1] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, 1999.

[2] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *the 26th VLDB Conference*, 2000.

[3] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *the 19th ICDCS Conference*, 1999.

[4] Min Cai, Martin R. Frank, Baoshi Yan, and Robert M. MacGregor. A subscribable peer-to-peer rdf repository for distributed metadata management. *J. Web Sem.*, 2(2):109–130, 2004.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[6] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11:354–379, 2002.

[7] Paul-Alexandru Chirita, Stratos Idreos, Manolis Koubarakis, and Wolfgang Nejdl. Publish/subscribe for rdf-based p2p networks. *Proceedings of 1st European Semantic Web Symposium*, 2004.

[8] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE TSE*, 27:827–850, sep 2001.

[9] Yanlei Diao, Peter Fischer, Michael Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of the 18th ICDE Conference*, 2002.

[10] Francoise Fabret, Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD*, 2001.

[11] Jesse James Garrett. Ajax: A new approach to web applications. *Adaptive Path*, 2005.

[12] Ashish Kumar Gupta and Dan Suciu. Stream processing of xpath queries with predicates. In *ACM SIGMOD*, 2003.

[13] Volker Haarslev and Ralf Moller. Incremental Query Answering for Implementing Document Retrieval Services. In *Proc. of the International Workshop on Description Logics*, 2003.

[14] Olga Papaemmanouil and Ugur Cetintemel. Semcast: Semantic multicast for content-based data dissemination. In *Proceedings of the 21st ICDE Conference*, 2005.

[15] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. G-ToPSS - fast filtering of graph-based metadata. In *the 14th WWW Conference*, 2005.

[16] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J. Wang. Subscription Partitioning and Routing in Content-based Publish/Subscribe Systems. In *the International Symposium on Distributed Computing Toulouse*, 2002.