G-ToPSS: Fast Filtering of Graph-based Metadata

Milenko Petrovic
Department of Computer Engineering
University of Toronto
petrovi@eecg.toronto.edu

Haifeng Liu
Department of Computer Science
University of Toronto
hfliu@cs.toronto.edu

Hans-Arno Jacobsen
Department of Computer Engineering
Department of Computer Science
University of Toronto
jacobsen@eecg.toronto.edu

ABSTRACT

RDF is increasingly being used to represent metadata. RDF Site Summary (RSS) is an application of RDF on the Web that has considerably grown in popularity. However, the way RSS systems operate today does not scale well. In this paper we introduce G-ToPSS, a scalable publish/subscribe system for selective information dissemination. G-ToPSS is particularly well suited for applications that deal with large-volume content distribution from diverse sources. RSS is an instance of the content distribution problem. G-ToPSS allows use of ontology as a way to provide additional information about the data. Furthermore, in this paper we show how G-ToPSS can support RDFS class taxonomies. We have implemented and experimentally evaluated G-ToPSS and we provide results in the paper demonstrating its scalability compared to alternatives.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval; H.3.4 [**Information Systems**]: Systems and Software

General Terms

Design, performance, algorithms

Keywords

publish/subscribe, content-based routing, RDF, information dissemination, graph matching

1. INTRODUCTION

The amount of information on the Internet is continuously increasing. It is becoming increasingly easier for non-computer oriented users to publish information on the Internet because of myriads of user-friendly tools that now exist. For example, it is very easy for a user to keep an "online" diary (e.g., blogs) using a variety of tools. Collaboration tools such as a wiki, allow users to quickly publish information from within a web browser, without requiring access or knowledge of any additional applications. Finally, applications for web page authoring are becoming ever so easier to use. As a result of the advances in web page authoring tools, the number of information publishers has grown considerably.

RDF Site Summary (RSS) is a metadata language by the W3C for describing content changes.[1] RSS is so versatile that any kind of content changes can be described (e.g., web site modifications, wiki updates, history of source code from a versioning software (e.g., CVS)). A RSS feed is a stream of RSS metadata that tracks changes for a particular content over time.

Typically, users apply a tool, which can read RSS feeds, to periodically check a number of RSS feeds by pulling RSS files from a web site. When RSS feeds indicate that the content has been updated, the user is informed. The user is expected to explicitly specify which RSS feeds to monitor.

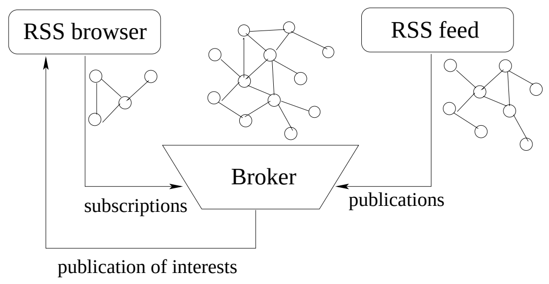


Figure 1: RDF Site Summary Dissemination System based on G-ToPSS

A RSS feed aggregator is a service that monitors large numbers of 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.

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 (e.g., due to ease of publishing information on the web), this architecture is not going to scale. It not only consumes unnecessary resources, but also becomes difficult to ensure timely delivery of updates.

[1]http://web.resource.org/rss/1.0/spec

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

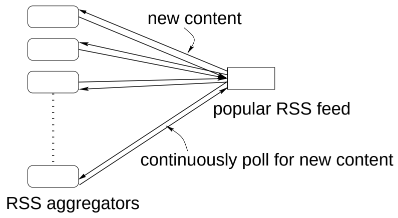


Figure 2: Current RSS dissemination architecture

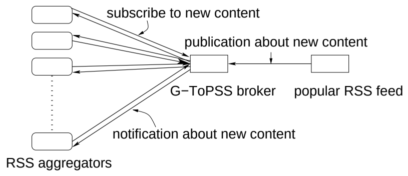


Figure 3: G-ToPSS RSS dissemination architecture

Figure 2 illustrates the scalability problem. Multiple RSS aggregators (i.e., personal (desktop) aggregators, online news aggregators, and server side aggregators) *poll* numerous RSS feed sites, *each*. Anecdotal evidence suggests that the way RSS dissemination is currently done can severely affect the performance of websites hosting popular RSS feeds. [2]

In this paper, we describe G-ToPSS[3], a graph-based publish/subscribe architecture for dissemination of RDF data. The G-ToPSS system provides fast filtering of RDF metadata such as RSS publications, as well as timely delivery of publications to interested subscribers in a scalable manner. Figure 1 shows the architecture of G-ToPSS. The new information system architecture significantly reduces the number of unnecessary polls of RSS feed sites (see Figure 3).

RSS is just one application that can benefit from this architecture. Another application that is increasingly becoming important is content management in the enterprise. PDF is the de facto standard for representing documents in electronic form while preserving their original formatting. RDF metadata can be embedded in PDF documents, which aids in document management. G-ToPSS provides an architecture that could be applied to efficiently content-based routing.

In addition, [8] describes a number of uses cases for RDF data access, many of which can directly benefit from the described architecture. Some examples include “finding unknown media objects,” “avoiding traffic jams” and “exploring the neighborhood.”

G-ToPSS employs the publish/subscribe, data-centric communication model. There are three main entities in this model: publishers, subscribers and brokers. Publishers send all data to a broker (or a network of brokers). Subscribers register with the broker their interest in receiving some data. The role of a broker is to mediate communication between the publishers and the subscribers by matching the published data with the interests of the subscribers. This way the subscribers do not need to know who is publishing the data, as long as the data meets their specific interest, and the publishers do not need to know who are the ultimate receivers of their publications. This provides decoupling of senders and receivers of data both in space and time, which makes the publish/subscribe paradigm particularly well suited for structuring of large and dynamic distributed systems such as RSS feed dissemination for example.

The contributions of this paper are:

1. An original publish/subscribe system model is developed to support large-volume graph-based content distribution from diverse sources.

2. G-ToPSS allows the use of ontology to specify class taxonomy as semantic information about the data.

3. G-ToPSS system offers scalability with the increase of the number of users while maintains efficient filtering rate.

The paper is organized as following. In Section 2 we briefly summarize related work. The G-ToPSS publish/ subscribe model supporting graph matching is developed in Section 3. Section 4 describes the graph matching algorithms and data structures. Section 5 presents the experimental evaluation and Section 6 concludes the paper and discusses the directions for future work.

2. RELATED WORK

Use of the publish/subscribe communication model for selective information dissemination has been studied extensively. Existing publish/subscribe systems [9, 1, 6, 4] use attribute-value pairs to represent publications, while conjunctions of predicates with standard relational operators are used to represent subscriptions. Systems such as those described in [2, 7] use XML to express publications and XPath as the subscription language. XPath provides a way to express path patterns over a tree, but it does not allow patterns to be further constrained using relational operators, as does G-ToPSS and other non-XPath systems.

Previously, we have built a prototype publish/subscribe system S-ToPSS [13] that extends the traditional attribute-value-pair-based systems with capabilities to process syntactically different, but semantically-equivalent information, thus achieving another level of decoupling, which we termed *representational decoupling*. S-ToPSS uses an ontology to be able to deal with syntactically disparate subscriptions and publications. The ontology which can include synonyms, a taxonomy and transformation rules was specified using S-ToPSS specific methods. On the other hand, G-ToPSS publication and subscription data models are based on directed graphs in general and RDF in particular. Use of RDF makes it possible for G-ToPSS to use ontologies built on top of RDF using languages such as RDFS and OWL. To illustrate this, in this paper, we extend the G-ToPSS subscription language with type constraints for subjects and objects, where the type information is represented in a RDFS taxonomy.

OPS [14] is another ontology-based publish/subscribe system whose publication and subscription model is also based on RDF. OPS uses a very general subgraph isomorphism algorithm for matching over overlapping graphs. However, this approach, as we show in this paper, unnecessarily increases the matching complexity because it assumes that any

[2]InfoWorld RSS growing pains, July 16, 2004
RSS Traffic Burdens Publisher’s Servers, July 19, 2004
[3]G-ToPSS is a part of the Toronto Publish/Subscribe System (ToPSS) research effort, which comprises a large number of publish/subscribe research projects, such as M-ToPSS (mobility-aware) [3], S-ToPSS (semantic matching) [13], A-ToPSS (approximate matching) [12], L-ToPSS (location-based matching) [15], padres (federated p/s) [11] and others.

node of the publication graph can map to any node of the subscription graph. In this paper, we compare the performance of G-ToPSS to OPS and show that G-ToPSS always outperforms OPS.

A RDF document can be represented as directed labelled graph. Every node in the graph has a unique name, and no two edges between any two nodes can have the same label either. Given this assumption, in this paper, we show how to store such graphs in a way that exploits commonalities between them and how to use this data structure to efficiently filter publications.

Racer [10] is a publish/subscribe system based on a description logics inference engine. Because OWL is based on description logics, Racer can be used for RDF/OWL filtering. Racer does not scale as well as G-ToPSS (matching times are in the order of 10s of seconds even for very simple subscriptions), but it does have more powerful inference capabilities.

CREAM [5] is an event-based middleware platform for distributed heterogeneous event-based applications. Its event dissemination service is based on the publish/subscribe model. Similar to other publish/subscribe systems, the subscription and publication model in CREAM, is based on attribute-value pairs. Like S-ToPSS, attributes and values can be associated with semantic information from an ontology. Unlike G-ToPSS, which is based on RDF, ontology and data are represented in a CREAM-specific data model. In addition, we are not aware of any quantitative evaluations of CREAM's scalability such as the one for G-ToPSS presented in this paper.

3. G-TOPSS MODEL

We describe the three components of the G-ToPSS data model: publications, subscriptions and ontology.

3.1 Publication Data Model

A G-ToPSS publication is represented as a *directed labelled graph.* In this paper, 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 (as shown in Figure 4). *subject* and *property* are URI references, while *object* is either an URI reference or a literal. A publication is a directed graph where the vertices represent *subject*s and *object*s and edges between them represent *properties*.

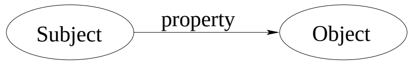


Figure 4: RDF triple graph

Figure 5(c) illustrates a publication about one of Prof. Jacobsen's papers published in the 2001 SIGMOD conference.

3.2 Subscription Language Model

A G-ToPSS subscription is a *directed graph pattern* specifying the structure of the publication graph with optional constraints on some vertices. A subscription is represented by a set of 5-tuples (*subject, property, object, constraintSet (subject), constraintSet (object)*). Constraint sets can be empty.

Similar to the publication data model, each 5-tuple can be represented as a link starting from the *subject* node and ending at the *object* node with the *property* as its label. From the publication data model, we know that each node is labelled with a specific value. However, in a subscription, we also allow *subject* and *object* to be either a constrained or unconstrained variable. An unconstrained variable matches any specific value of the publication; while the constraint variable matches only values satisfying the constraint. A constraint is represented as a predicate of the form $(?x, op, v)$ where $?x$ is the variable, *op* is an operator and v is a value.

There are two types of operators: Boolean, for literal value filtering and *is-a*, for RDFS taxonomy filtering. Boolean constrains are one of $=$, $\leq$ and $\geq$ with traditional relational operator semantics. *is-a* operators are also one of $=$, $\leq$ and $\geq$ but with alternative semantics. $\leq$ means the instance $?x$ is the descendant of the class v. $\geq$ means that the instance $?x$ is the ancestor of class v. $=$ means that $?x$ is the direct instance of class v (i.e., a child of v).

For example, Figure 5(a) illustrates a subscription that specifies interest in a paper published at the SIGMOD conference after the year 2000. This type of constraint is for literal value filtering.

The subscription in Figure 5(b) is looking for Arno's publication in a conference after 1999. There are two variables; the one constraining the year is a literal value filter; the other is a semantic constraint which uses the class taxonomy. Only an instance in the publication that is a descendant of the "Publication" class is going to match.

3.3 Matching Semantics

We denote G_P as the publication graph and G_S as the subscription graph pattern. The matching problem is then defined as verifying whether G_S is embedded in G_P (or isomorphic to one or more subgraphs of 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.

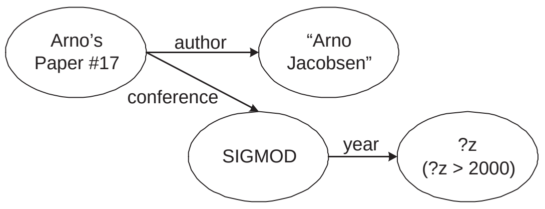
Formally speaking, for each 5-tuple (*subject, property, object, constraintSet (subject), constraintSet (object)*) in subscription graph G_S, there is at least one triple (*subject, property, object*) in publication G_P such that the *subject* and *object* nodes are matched and linked by the same *property* edge. The nodes that match are either the same (i.e., their labels are lexicographically equal) or the node in G_S is a variable for which the value of the node in G_P satisfies all constraints associated with the variable.

For example, the subscription in Figure 5(a) is matched by the publication in Figure 5(c) since the publication contains the same links (*Arno's paper* $\sharp$*17, author, Arno Jacobsen*), ((*Arno's paper* $\sharp$*17, conference, SIGMOD*), and (*2001* $>$ *2000*), thus (*SIGMOD, year,* $?x(?x > 2000)$) is satisfied.

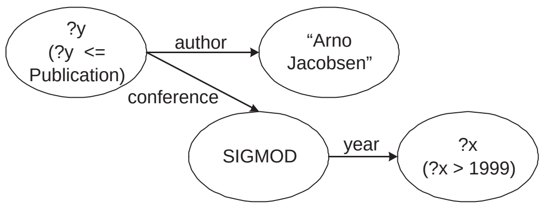
3.4 Ontology Support

A RDFS class taxonomy with *is-a* relationship is the semantic information about a *subject* or an *object* that is available in the G-ToPSS ontology. Multiple inheritance is allowed and the only restriction on the taxonomy is that it must be acyclic. We also list all instances of a class in the taxonomy. Alternatively, this information can be specified in the RDF graph using a *type* property, but for simplicity we have opted to include this information in the taxonomy. Note that an instance can also have multiple parents.

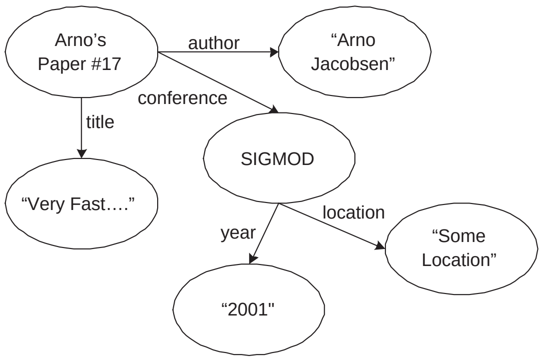
In Figure 6, we show an example of a class taxonomy about an academic bibliography system. Class "Publications" includes three subclasses: "Journal", "Conference Proceeding" and "Technical Report". "Technical Report"



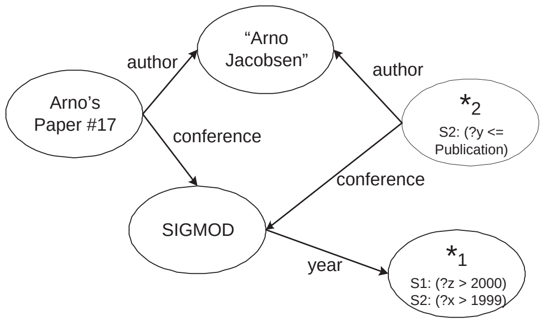
(a) Subscription S_1



(b) Subscription S_2



(c) event



(d) G_M contains S_1 and S_2

Figure 5: Example subscriptions, events and G_M

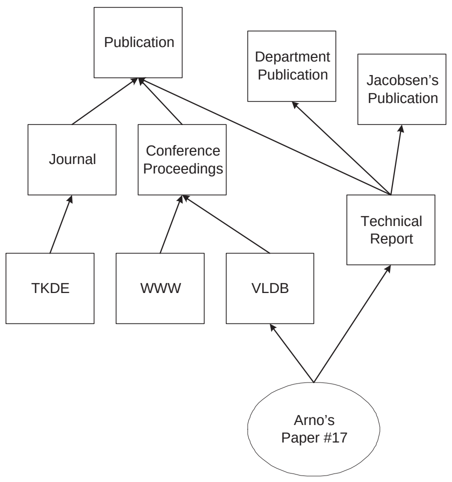


Figure 6: Example taxonomy

belongs to "Publications", "Department Publications" and "Jacobsen's Publications" simultaneously. The document instance "Arno's paper ♯17" belongs to both "Jacobsen's Publications" and "SIGMOD" proceedings.

As a side note, existing publish/subscribe systems are classified as either content-based or hierarchical (topic) based. Thus, class taxonomy is a way to seamless integrate both models. When filtering, a subscription is matched if and only if both the content and the hierarchical constraints are satisfied.

4. ALGORITHM AND DATA STRUCTURE

To exploit overlap between subscriptions we integrate all subscriptions into a single graph. We denote the graph containing all subscriptions as G_M. Given all subscriptions, G_M, a publication, G_P, the publish/subscribe graph 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. In other words, the goal is to determine all graph patterns, G_{S_i}, in G_M that match some subgraph of G_P.

This matching problem is different from subgraph isomorphism. The subgraph isomorphism problem can be stated as follows: given graph G_1 and G_2, identify all subgraphs of G_2 which are isomorphic to G_1. This differs from the problem we are trying to solve which is to identify all subgraphs of G_2 that are isomorphic to *some* subgraph of G_1.

4.1 Data Structure

Since there can be multiple edges between the same pair of nodes, we use two-level hash tables to represent G_M. At the first level, we use a hash table to store all the pairs of vertices taking the names of the two nodes as the hash key. Each entry of the first hash table is a pointer to another (second-level) hash table that contains a list of all the edges between these two nodes. The edge label (i.e., "property" in the 5-tuple) is used as the hash key. Each edge points to a list of subscriptions that contain this edge.

Figure 7 shows the data structure of G_M. There are two edges between node A and B and both $s1$ and $s2$ contain the edge a between A and B.

Any subscription can contain multiple variables that can be matched by any vertex in the publication graph. For ex-

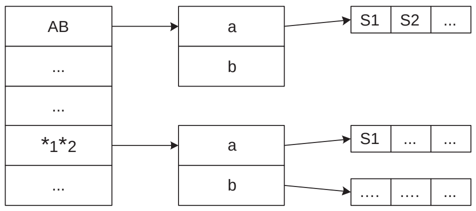


Figure 7: Data Structure

ample, Figures 5(a) and 5(b) show two subscription graphs containing variables and the merged subscription graph, G_M, in Figure 5(d).

The data structure from Figure 7 allows us to store uniquely labelled nodes only once. In other words, nodes belonging to different subscriptions, but with the same label map to the same node in G_M. This is possible because each node in a graph is uniquely identified by its label. However, this is not the case with nodes with variable labels. Variable labels do not uniquely identify nodes, but instead they represent a (possibly constrained) pattern on node labels from a publication.

We introduce a special sequence of labels, $\star_i | i \geq 1$, to represent variables. The value of index i is bounded by the number of variables in the subscription with the most variables among all subscriptions in G_M.

For example, in Figure 5(d), we use one node labelled as $\star_1$ to represent both $?x$ and $?z$; $?x$ and $?y$ are represented by two nodes $\star_1$ and $\star_2$ since they appear in the same subscription. Mapping between original variable labels from the subscription (e.g., ?x) to the corresponding *star* name is preserved.

Mapping of variables from subscriptions to star labels is arbitrary for the sake of simplicity, even though some mappings are better than others since they can results in a sparser G_M. In the future, we are going to investigate how much can be gained, in terms of matching performance, by having a more sophisticated mapping.

4.2 Matching Algorithm

First, we discuss how the subscription matrix is created when inserting subscriptions. Suppose G_M is a graph containing all subscriptions, and G_S is a subscription graph. $|G_S.\star|$ is the number of variables in the subscription graph, variable vertices in G_S are labelled as $\star_i$ where $0 < i < |G_S.\star| + 1$. $G_M.\star$ is the number of stars in the G_M. Note that all vertices in G_S and G_M are unique. $G_M.T1$ is the first-level hash table, and $T2$ is the second-level hash table. $E.subs$ is a set of subscriptions containing edge E, $G_M.subs$ is the set of all subscriptions in G_M. E (and $E2$) is a directed edge from $E.v$ to $E.w$, $E.smEdge$ is an edge in G_M that overlaps with E. New Table(A,B) creates a table with 2 columns A and B that will be used to decided on the bindings for variables.

```
Algorithm Insert($G_S$)
1.  if $G_S.\star > G_M.\star$
2.      $G_M.\star = G_S.\star$
3.  for each edge $E \in G_S.edges$
4.      $T2 = G_M.T1.getTable(E.v, E.w)$
5.      if ($T2$ is null)
6.          $T2 = G_M.T1.insert(E.v, E.w)$
7.      $E2 = T2.getEdge(E)$
8.      if ($E2$ is null)
9.          $E2 = T2.insertEdge(E)$
10.     $E2.bindingTable = newTable(E.v, E.w)$
11.     $E2.subs = E2.subs + G_S$
12.     $G_M.subs = G_M.subs + G_S$
13.     $E.smEdge = E2$
```

Algorithm *Insert* is the procedure for subscription insertion. For each edge in G_S, we check if there is a corresponding edge in the first-level hash table. If there is no such edge, we update the hash tables by inserting $E.vE.w$ into the first-level hash table and edge E into the corresponding second-level hash table. Finally, the subscription id is inserted into the list associated with edge E and added to $G_M.subs$.

Next, we explain how to perform matching using the subscription graph G_M when a publication arrives. G_E is the publication graph (the number of edges in G_E is m). G'_E is a completed graph containing vertices $E.v$, $E.w$, $\star_i$ such that $0 < i < |G_M.\star| + 1$. All nodes in G_E are unique. *SubSet* contains all subscriptions that have at least one edge in G_M that are referenced by G_E. *Result* is a set of (S,R) where S is a subscription and R is a satisfying binding for variables. Natural join ($\bowtie$) is an equality join on all common columns.

```
Algorithm match($G_E$)
1.  for each $E \in G_E.edges$
2.      create a fully connected graph $G'_E$
3.      for each edge $E2 \in G'_E$
4.          $T2 = G_M.T1.getTable(E2.v, E2.w)$
5.          if ($T2$ not null)
6.              $E3 = T2.getEdge(E)$
7.              if ($E3$ not null)
8.                  for all $S \in E3.subs$
9.                      $S.edgeCount++$
10.                     $E3.bindingTable+ = (E.v, E.w)$
11.                     $SubSet = SubSet + E3.subs$
12. $result = 0$
13. for all subscriptions $S \in SubSet$
14.     if ($S.edgeCount \geq |S.edges|$)
15.         $S.edgeCount = 0$
16.         $b = E.smEdge.bindingTable | E \in S$
17.         for every edge $E2 \in S.edges - E$
18.             $b = b \bowtie E2.smEdge.bindingTable$
19.         for every row $R \in b$
20.             if CheckConstraint($R$,$C_S$, $T$)
21.                 result = result + $(S, R)$
```

Algorithm *match* is the procedure for matching publications against subscriptions. There are two stages in the matching process. First, for each edge in the publication, we check all the matched subscription edges in G_M. Then we find the satisfying bindings for variables and evaluate the constraints.

In the first stage, for the publication edge v_1v_2, the potentially matched edges in G_M include v_1v_2, $v_1\star_i$, $\star_iv_2$ and $\star_i\star_j$. There are three actions to perform on these potentially matched edges. (1) Add v_1v_2 into the binding tables of all these matched edges so that they can be used in the second stage. (2) Increase the counters of subscriptions associated with these edges. (3) Put these subscriptions into the *Subset* as the candidates of matches. This completes the first stage of matching.

In the second stage, we find the matched subscriptions by checking the candidates in *Subset* one-by-one. For each subscription s_i in *Subset*, we join all the binding tables of edges belonging to s_i. If the result table is not empty, then the entries in the result table contain all valid binding values for all variables in the subscription.

Figure 8 illustrates an example for a binding table join. For example, the subscription contains two edges $A\star_1$ and $\star_1 B$. There are three entries in the binding table of $A\star_1$ which means $A\star_1$ is matched by three edges AB, AC and AE in the publication. $\star_1 B$ is matched by 5 edges in the publication. Joining of these two tables produces ACB and AEB and hence $\star_1$ can be bounded with value C and E.

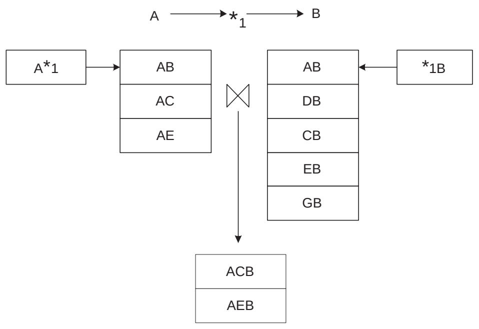


Figure 8: Binding table join

After identifying all valid bindings of variables, we can use the binding value w to evaluate the constraint. For the constraint $(?x, op, v)$, we need to check whether $(w\ op\ v)$ is true. For the value filtering constraint, $(w\ op\ v)$ is evaluated using standard relational operator comparison.

For the class taxonomy filtering constraint $(w\ op\ v)$, we need to check the descendant-ancestor relationship between the specific instance w and the class v by traversing the taxonomy tree. The constraint checking algorithm is shown in Algorithm *CheckConstraint*.

```
Algorithm $CheckConstraint(R, C_S, T)$
1.  for each variable $\star$ in $S$
2.      find the value $v$ in $R$ and the constraint $(op, c)$
3.      return isTrue($v, op, c, T$)
Algorithm $isTrue(v, op, c, T)$if
1.  $op = LT$ return isNodeDescendant(v, c, T)
2.  if $op = GT$ return isNodeDescendant(c, v, T)
3.  if $op = EQ$ return (c.equals(v))
```

For example, in Figure 5(d), for subscription s_2, $\star_1$ is matched by node "2001" since $2001 > 1999$ and $\star_2$ is matched by node "Arno's paper ♯17" since it is descendant of class "Publication."

4.3 Analysis

Space Complexity: The space cost mainly includes two parts: hash tables and linked lists associated with each edge to store the subscription ids that contain this edge. The size for the hash tables is determined by the number of unique edges among all the subscriptions. The length of the linked list depends on the average number of subscriptions each edge is associated with. Therefore, we have

$$O(|G_M.edgs| + |G_M.edgs| \times N_{s_e})$$

where $|G_M.edges|$ is the number of unique edges in matrix G_M and N_{S_e} is the average number of subscriptions each edge is associated with.

Time Complexity: The time of the Insert(G_S) algorithm depends on the number of edges for each subscription and the complexity is

$$O(|G_S.edges|).$$

For the matching algorithm, it consists of two steps. First, edge matching. By checking each edges in the publication, we determine all the subscriptions that have at least one edge matched by the publication. The time of the first step depends on the size of the completed graph G'_E and the number of edges in the publication. Since each graph G'_E contains all the stars in G_M plus $E.v$ and $E.w$, the number of edges in G'_E is $\binom{k+2}{2}$. Suppose k is the number of stars in G_M, m is the number of edges in the publication, we have

$$O(m * 2\binom{k+2}{2}) \sim O(mk^2).$$

In the second step, for each subscription in *SubSet*, if all the edges of it are matched, we perform a join operation on the binding tables to determine whether there is a satisfying binding of the variables, then we check the constraints. To join two tables, the time is linear with the size of the smaller table. The time complexity to find satisfying bindings of variables for each subscription is

$$O(k * l)$$

where k is the number of stars in G_M and l is the size of the smallest binding table for variables.

The time to check whether the constraint for the variable is satisfied according to the class taxonomy is dependent on the complexity of the taxonomy tree. Since multiple parents are allowed in the class taxonomy tree, the time is $O(d^t)$ where d is the depth of the tree and t is the average number of parents each node may have.

Overall, the matching time to evaluate all subscriptions is

$$O(mk^2) + O(n * k * l + n * k * d^t)$$

where n is the number of subscriptions in *SubSet*. In real applications, the class taxonomy tree is fixed, the number of variables in one subscription is small (usually 1 to 3, at most 5), $m << n$, and n is around the number of matched subscriptions. Therefore, the overall matching time is linear with the number of matched subscriptions:

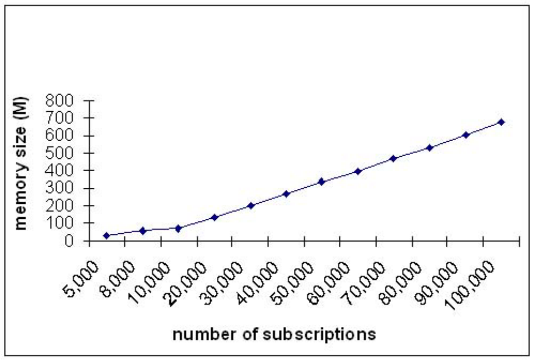
$$O(ratio_{match} * number\_of\_subscriptions).$$

5. EVALUATION

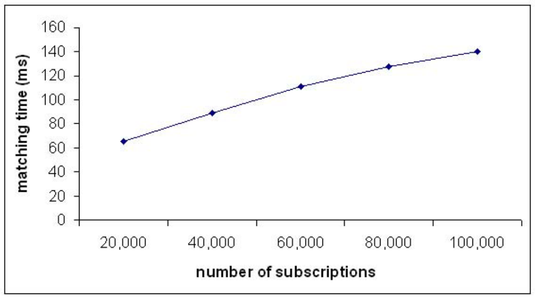
We have implemented the algorithm in Java. We experimentally evaluate the rate of matching and the memory use. We run the experiments on a Linux system with 1GB RAM and a 1GHz microprocessor. We are using a synthetic workload so that we can independently examine various aspects of G-ToPSS. We report the results for the two most important metrics from a user's perspective, namely the rate of matching and the memory requirements. The workload parameters are shown in Table 1.

$Size_P$ and $Size_S$ are decided by (number of nodes, number of edges) the publication graph and the subscription graph. The number of edges must be larger than the number of nodes in order to obtain a connected graph. We use $ratio_{match}$ to control the number of matched subscriptions that are generated as subgraphs from the publication graph.

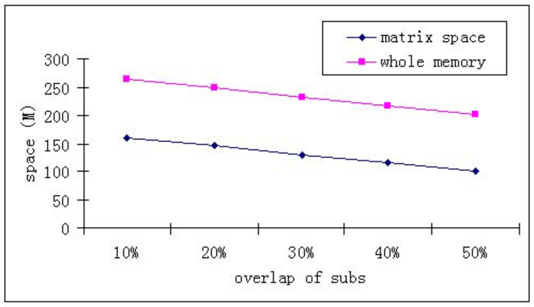
We generate the test workload using the parameter values from Table 1. A publication is generated first. For example, for publication of size (k,m) we first generate a simple path of length $k-1$ then we generate $m-k+1$ edges between random pairs of the k nodes.



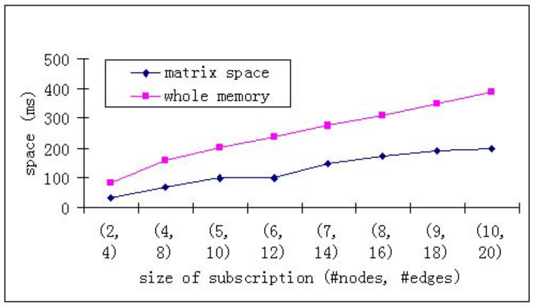
(a) Memory vs. #subscriptions



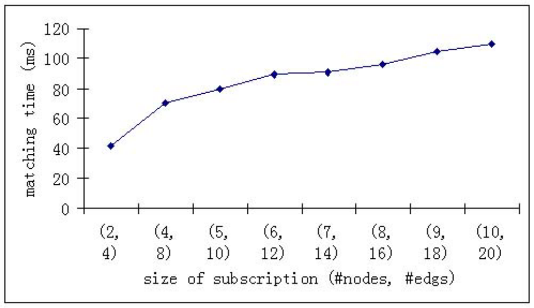
(b) Matching time vs. #subscriptions



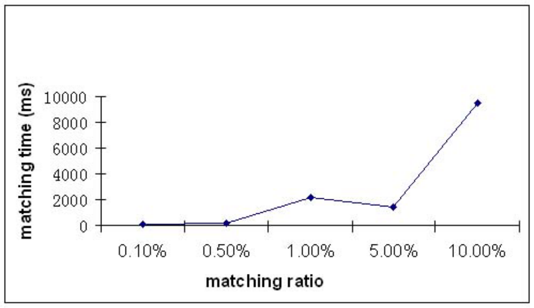
(c) Memory vs. subscription overlap



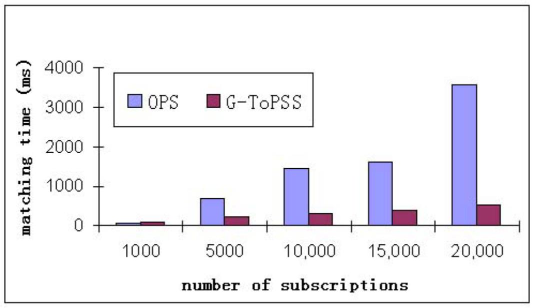
(d) Memory vs. subscription size



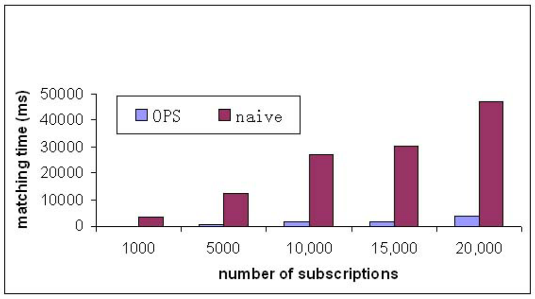
(e) Matching time vs. subscription size



(f) Matching time vs. matching ratio



(g) G-ToPSS vs. OPS



(h) OPS vs. naive

Figure 9: Experimental performance results

Table 1: The workload parameters in experiments

parameters	default values	description
$Size_P$	(35,90)	size of publication
$Size_S$	(5,35)	size of subscription
N_{sub}	30,000	number of subscriptions
$ratio_{match}$	0.1%	ratio of matched subscriptions among all
N_{stars}	2	number of stars (variables) in one subscription
N_{sub^*}	27,000	number of subscriptions containing stars
$overlap_s$	50%	ratio of overlap among subscriptions

Subscriptions are generated in four steps. First, $ratio_{match}$ subscriptions that match the publication are generated by randomly selecting a subgraph of the publication. Then, $overlap_s$ subscriptions are generated that completely overlap using the same technique as for matching subscriptions. Then, $N_{sub} - overlap_s$ non-overlapping subscriptions are generated randomly in the same way that the publication was generated. In the fourth step, N_{stars} vertices are selected from all N_{sub^*} subscriptions and replaced with a variable ($\star$). Alternatively, we limit values that can be bound to a variable by adding constraints.

All measurements are performed after G-ToPSS has loaded all the subscriptions. We look at the effect of the number of subscriptions, subscription size and matching ratio (number of subscriptions matched by a publication). Finally, we compare G-ToPSS with two alternative implementations. For each experiment, we vary one parameter and fix others to their default values as specified in Table 1.

Number of subscriptions: Figure 9(a) shows the memory use with increasing number of subscriptions. We see that the memory size grows linearly as the number of subscriptions increase. Since all subscriptions in our experiments are of the same size and the overlap factor is constant, the memory increase per subscription is also a constant.

Figure 9(b) shows the time to find all matches for a publication given a fixed set of subscriptions. As the set of subscriptions increases, so does the time. The number of subscriptions that match the publication is relative to the total number of subscriptions in the set. Consequently, the number of matches increases as the number of subscriptions is increased.

The time to match a publication is split between structure matching phase and constraint evaluation phase. As the number of subscriptions increase, both of these times increase by a fixed amount because the number of matches increases constantly.

Subscription size: Figure 9(c) shows how the space used by the subscriptions decreases as the overlap between them increases. We present this to validate our workload. The matrix space is the size of G_M, while *whole memory* is equal to the size of G_M plus the space used to store all the subscriptions.

Figure 9(e) shows the effect of increasing subscription size on the matching time. We see that the time increases more rapidly as the number of edges increases (e.g., from 4 to 8), the time almost doubles. On the other hand, as the increase in number of edges decrease, so does the increase in matching time, hence the matching time is not affected by the number of nodes, but by the number of edges in the subscription.

Matching ratio: Figure 9(f) shows the effect of increasing the number of subscriptions that match the publication. As this number grows, the time to match grows very quickly. This is mainly due to increased time to calculate all the bindings for each subscription.

G-ToPSS vs. Alternatives: In Figure 9(g) we compare the performance of our algorithm to the OPS algorithm. As the graph shows, OPS matching time increases very quickly with the number of subscriptions. The main reason for the significant difference in matching times comes from the differences in basic assumptions. The OPS algorithm makes the same basic assumption as do other, traditional, subgraph isomorphism algorithms, namely that every node in a subscription is a variable. In other words, any node of a publication can match with any other node in the subscription graph. However, this assumption unnecessarily increases the matching complexity, as we see in the evaluation. We make a more realistic assumption that the number of variables in any subscription is low as compared to the total number of nodes in a subscription graph and the nodes in a RDF publication are unique.

Figure 9(h) illustrates that, even though OPS is less scalable than G-ToPSS, it is still far better then a naive approach which sequentially checks all subscriptions to find the matching ones.

6. CONCLUSIONS AND FUTURE WORK

Use of RDF as a language for representing metadata is growing. Applications such as RSS and content management are exhibiting use patterns that current systems were not designed for.

The G-ToPSS prototype shows that a data-centric, push-based architecture such as publish/subscribe is a very good fit for just such applications. G-ToPSS is able to support high matching rates for very complex subscriptions. In practice, we expect these subscriptions to be simpler (i.e., have smaller number of edges and stars) on average than the ones used in our experiments.

Being based on RDF, G-ToPSS can be easily extended to use additional semantic information expressed in languages built on top of RDF, such as RDFS and OWL. We show how a RDFS taxonomy can be used to increase the expressiveness of the G-ToPSS query language. Our implementation uses an efficient traversal of the class hierarchy with support for multiple inheritance, which adds more expressiveness to the language without unduly affecting the matching rate. On the other hand, more powerful inference techniques such as those of Descriptions Logics (on which OWL is based) could augment the constraint filtering without significant changes to the matching engine.

In the future, we will work on extending G-ToPSS with full RDF language features (such as bags and sequences), which we have left out since their implementation does not affect the matching rate but merely adds syntactic sugar.

Extending G-ToPSS to support variables on predicates is straight forward since the same techniques for supporting variables on subjects and objects can be used. Consequently, matching time complexity is not affected by this extension.

In addition, we are going to implement several optimizations for constraint processing. If the number of overlapping constraints is large, then the systems can benefit from parallel constraint evaluation, such as only evaluating unique constraint once by exploiting the overlap among constraints. Techniques for parallel constraint evaluation have already been examined in previous research on attribute-based publish/subscribe systems and we believe that the same techniques can be applied here with small modifications in insertion and matching algorithm. These optimizations are useful when the structure of the subscriptions is practically the same so that the degree of overlap is large. In this case, the performance affecting filtering happens during the constraint matching phase. Furthermore, we are currently examining ways of doing natural join for overlapping subscriptions in a way that takes advantage of the overlap.

7. REFERENCES

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

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

[3] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic. Disconnected operation in publish/subscribe middleware. In *Mobile Data Management*, 2004.

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

[5] M. Cilia, C. Bornhoevd, and A. P. Buchmann. CREAM: An Infrastructure for Distributed Heterogeneous Event-based Applications. In *Proceedings of the International Conference on Cooperative Information Systems*, pages 482–502, 2003.

[6] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27:827–850, sep 2001.

[7] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of ICDE2002*, 2002.

[8] K. G. C. (ed). RDF Data Access Use Cases and Requirements. *W3C Working Draft*, 2004.

[9] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD Conference*, 2001.

[10] V. Haarslev and R. Moller. Incremental Query Answering for Implementing Document Retrieval Services. In *Proceedings of the International Workshop on Description Logics*, 2003.

[11] G. Li, S. Hou, and H. A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. *International Conference on Distributed Computing Systems (ICDCS'05).*

[12] H. Liu and H.-A. Jacobsen. A-ToPSS - a publish/subscribe system supporting approximate matching. In *Very Large Databases (VLDB'02)*, University of Toronto, August 2002.

[13] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS - a semantic publish/subscribe system. In *Very Large Databases (VLDB'03)*, Berlin, Germany, September 2003.

[14] J. Wang, B. Jin, and J. Li. An Ontology-Based Publish/Subscribe System. In *Middleware*, 2004.

[15] Z. Xu and H. A. Jacobsen. Efficient constraint processing for location-aware computing. In *6th International Conference on Mobile Data Management (MDM'05), Ayia Napa, Cyprus*, 2005.