Management of Uncertainties in Publish/Subscribe System

by

Haifeng Liu

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Management of Uncertainties in Publish/Subscribe System

Haifeng Liu

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

In the publish/subscribe paradigm, information providers disseminate publications to all consumers who have expressed interest by registering subscriptions. This paradigm has found wide-spread applications, ranging from selective information dissemination to network management. However, all existing publish/subscribe systems cannot capture uncertainty inherent to the information in either subscriptions or publications.

In many situations the large number of data sources exhibit various kinds of uncertainties. Examples of imprecision include: exact knowledge to either specify subscriptions or publications is not available; the match between a subscription and a publication with uncertain data is approximate; the constraints used to define a match is not only content based, but also take the semantic information into consideration. All these kinds of uncertainties have not received much attention in the context of publish/subscribe systems.

In this thesis, we propose new publish/subscribe models to express uncertainties and semantics in publications and subscriptions, along with the matching semantics for each model. We also develop efficient algorithms to perform filtering for our models so that it can be applied to process the rapidly increasing information on the Internet. A thorough experimental evaluation is presented to demonstrate that the proposed systems can offer scalability to large number of subscribers and high publishing rates.

To my parents

Acknowledgements

I am in great debt to many people for the completion of this thesis and the duration of my PhD journey.

The first person I wish to thank is my supervisor, Professor Hans Arno Jacobsen, who has led me into the area of computer science and computer engineering and guided me through the study and research in the past years. He has patiently taught me how to become a good researcher, by kindly correcting me, constantly encouraging me, and always being very cooperative. I believe I would continue to benefit from the spirits he has demonstrated for us in the rest of my life.

I wish to thank my supervisory committee members, Professor John Mylopoulos and Professor Steve Easterbrook, for their helpful feedback on my research work. I am greatly thankful to another two defense committee members, Professor Dave Wortman and Professor Tamer Ozsu from University of Waterloo, for carefully reading my thesis and giving me insightful comments and suggestions.

I had the privilege of interacting with bright and talented Milenko Petrovic and Vinod Muthusamy on a number of projects. They have taught me much, and their advice, feedback, and friendship have made my Ph.D. experience both more educational and more fun. Deepest thanks to all my dear middleware systems research group folks! Especially thank to Guoli Li, Hou Shuang, Zhengdao Xu, Milenko Petrovic, Vinod Muthusamy, Bala Maniymaran, Alex Chuang and Reza Sherafat. I will always remember your strongest support and warmest care when I really needed them! All our brain-storming discussions, hardworking days and nights, together with all the laughters and gatherings, will remain part of my memories.

I owe my gang of friends in Toronto for giving me the most valuable care and help through these years. Dongning, my first officemate and friend in Canada. From you, I have learnt to be more tolerant and considerate. Ying and Fei, you are my closest and most precious friends! I always feel totally free to share all my feelings and troubles with

you, because you are so understanding and wise. We have been together to get through so many things these years. I will cherish every moment we have spent together, no matter happiness and sadness, in my whole life. Yi Zhao, your smartness, openness and broad interests have brought me a lot of fun. Nan, I feel so pleasant to have known you; your intelligence, perseverance, and athleticism have deeply impressed me, I wish you have a very bright career path and a happy life! My dearest Hui, You are the source of my joyness. Your wisdom, generosity, tenderness, optimism and enthusiasm for science, nature and life have always enlightened my life. You are perfect! From you, I know what is beauty. Deepest thank for your patience, encouragement, support and love. And I firmly know that you are the one who are always there for me. I cherish everything you gave to me and I sincerely wish you all the best!

While in Toronto, I would like to give many many thanks to my friends, Ken Pu, Jin Chen, Naiqi Weng, Jingrui Zhang, Xuming He, Jiang Zhu, Dongying Li, Yunfeng Lin, Jin Jin, Jianhong Yu, Chao Li, Grace, Charlton, Xia chen and Samuel for their support and a great amount of amusement.

Finally but the most importantly, I would like to thank my great family – my beloved parents, my wonderful husband Xiaofei He, and my lovely sister Haibo Liu, for everything that they have given to me. They have stood by me in everything I have done, providing unconditional constant and endless support, encouragement and love. They are an inspiration to me in all that they do.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the proliferation of pervasive computing devices, the integration of network access technologies, the amount of information on the Internet 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. As the amount of information on the Web increases rapidly, efficient and timely dissemination of information has been seen as a key to distributing information to assist end-users.

Today most information dissemination services use a pull-based architecture where users actively pull information from a web site to find what they need. This pull-based architecture is not efficient in the sense that it not only consumes unnecessary resources, but also makes it difficult to ensure timely delivery of updates.

Also, the large amount of information providers offering content is driving the need for an information dissemination model that offers its users highly pertinent information in a demand-driven manner.

This can be supported extremely well through the publish/subscribe paradigm [33], which provides the loosely coupled form of interaction required in such large scale set-

tings. The publish/subscribe system anonymously interconnects information providers with information consumers in a distributed environment. The publish/subscribe system performs the matching and filtering task and ensures the timely delivery of publications to all interested subscribers.

The strength of this interaction style lies in the full decoupling in time, space and synchronization between publishers and subscribers. In the following sections, we describe applications and problems that could benefit from publish/subscribe systems, but require support from the publish/subscribe system that go beyond their current capabilities.

1.1 Motivation

Publish/subscribe has been well studied and many systems have been developed supporting this paradigm. Existing research prototypes, include, among others, Gryphon [10], LeSubscribe [34], and ToPSS [7, 56, 71, 15, 87]; industrial strength systems include various implementations of JMS [43, 65], the CORBA Notification Service [67], TIBCO's Tib/Rendezvoud product [89] and Web Services Notification [1].

All publish/subscribe systems developed to date are based on the crisp matching semantics, which means that neither subscribers nor publishers can express uncertain or partial information and a match between a subscription stored in the system and an event submitted to the system is either true or false – that is, a subscription matches or does not match. However, in many situations the large number of data sources exhibit various kinds of imperfection. In these cases, uncertainty about the state of the world has to be cast into the crisp data model that defines absolute limits. Moreover, for a user of the publish/subscribe system, it may be simpler to describe the state of the world with imperfect concepts – we say, in an approximate manner.

Examples of imperfect information include: exact knowledge to either specify subscriptions or publications is not available; the match between a subscription and a pub-

lication with uncertain data is approximate; the constraints used to define a match is not only content based, but also take the semantic information into consideration. The management of these kinds of imperfect information have received little attention in the scope of publish/subscribe research in the past. Next, we illustrate the problem for above imperfection with a number of concrete application scenarios and use cases.

First, in a selective information dissemination context, for instance, users may want to submit subscriptions about a book whose constraints on price is "cheap". On the other hand, information providers may not have exact information for all items published. In a second-hand market, a seller may not know the exact age of a antique vase so that she can just describe it as an "old" vase. Temperature and humidity information collected by sensors are often not fully precise, but only correct within a certain error interval around the value measured. It would be better to publish imprecise information, rather than a wrong exact value, if such publish/subscribe capabilities were possible.

Also, uncertainty exists for location-based information dissemination services. due to the movement of the mobile user and transmission delay, approximate location would be more appropriate rather than assessing the precise location of each user. To date location-based enabling technology, for instance, does not allow to continuously and accurately track a user's location. The user location is in many current location-based models assumed to be provided by the user herself. Otherwise, if location information is gathered through GPS, it is only accurate to a certain degree and can only be collected so and so often (e.g., it takes several seconds to get the first location quote.)

For these reasons, we think, it is of great advantage to provide a publish/subscribe data model and an approximate matching scheme that allows the expression and processing of uncertainty for both subscriptions and publications.

Secondly, most current publish/subscribe systems are based on content matching mechanism and do not consider the structure or the semantics of the data. In systems such as Elvin [86], LeSubscribe [34], PADRES [38] and ToPSS [56, 71, 87, 15, 51] and

CREAM [21], the publication and subscription model is based on attribute value pairs. There is no structure and semantic information associated at all. Systems as described in [17, 28, 29] can process XML publications and XPath subscriptions. However, these approaches are based on a tree index structure to support filtering of XML documents over XPath queries. These approaches do not support the filtering of graph-structured data, which is more general than tree-structured data.

Graph-based data is used in many emerging semantic web applications (e.g., blogs and wiki updates.) For example, RDF (Resource Description Framework) is used to represent information and to exchange knowledge on the web [76]. A RDF document is represented as a directed labeled graph. Use of RDF also makes it possible to use ontologies built on top of RDF using languages such as RDFS [80] and OWL [50] to process syntactically different, but semantically-equivalent information. Existing work as in Racer [42] is a publish/subscribe system based on a description logics inference engine and can be used for RDF/OWL filtering. Racer does not scale well, the matching time for Racer is on the order of 10s of seconds, even for very simple subscriptions (e.g. $retrieve(book)$).

A prime application for filtering RDF documents is RSS (RDF Site Summary) [85], a RDF-based language for expressing content changes on the web, which has grown considerably in popularity. RSS is so versatile that any kind of content changes can be described (e.g., web site modifications, wiki updates, history of a source code from a versioning software (e.g., CVS)). In current RSS delivery systems, multiple RSS aggregators continuously *poll* numerous RSS feed sites [77], which results in that websites hosting popular RSS feeds can be significantly overloaded with useless network traffic. The publish/subscribe system offers a solution to avoid the inherent polling-based design and provides scalability, as more users publish and expect to stay current with changes submitted by others.

Considering both expressiveness and scalability, it would be better if a publish/subscribe system model is developed to support large-volume graph-based content distribution from

diverse sources and allows the use of ontologies to specify class taxonomy as semantic information about the data.

Thirdly, in all the current publish/subscribe systems, the matching result is reported only after evaluating the whole subscription pattern, especially in the domain of the complex event processing (CEP). Li *et al.* [55] develop a distributed middleware platform that allows the user to subscribe to data published in both the future and the past. The matching result, which is either true or false, will not be reported until all the subscribed data has been seen. However, predicting the future matching result with a probability, given the current partial match is a novel and useful feature in complex event processing. Many commercial applications of CEP including algorithmic trading, credit card fraud detection, intrusion detection, business activity monitoring, and security monitoring could benefit from this feature. Take intrusion detection as an example, if a series of suspicious activities is indicative of a potential intrusion, the system could responds by resetting the connection or by re-configuring the firewall to block network traffic from the suspected malicious source. There is no publish/subscribe model that can predict the occurrence of complex events associated with a probability based on a partial match (i.e., a partially observed state.)

1.2 Problem Statement

In summary, we are concerned with three types of imperfect information in a publish/subscribe system. One type is semantic imperfect information and the other two are content imperfection. In general, we call all the above imperfection exhibited in a publish/susbcribe system as *uncertainties*. This thesis aims to effectively manage these three kinds of *uncertainties* in publish/subscribe systems. Next, we define the three types of imperfect information in detail.

The *semantic* imperfection refers to the matching semantics in a publish/subscribe

system. The components in a publish/subscribe system, especially in a distributed environment, are a priori decoupled, anonymous, and do not necessarily "speak" the same language. However, different language may refer to the same entity or have the same meaning in the real world. For example, an "icde paper" won't match a "publication" in a traditional content-based publish/subscribe system, but a match will be detected in a semantic publish/subscribe system. This problem can be solved by a *semantic* publish/subscribe system, which based the matching on semantics, not syntax of the language.

For the content imperfection, we concern two major types of imperfect information as defined in [83]: *imprecision* and *uncertainty*. *Imprecision* is related to the content of the statement. Publications and subscriptions are statements about events and users' interests. The expressions may be incomplete, ambiguous, or not well-defined, but involve the content of the statements. Thus, we refer to this type of imperfection in publications and subscriptions as imprecision.

Another type of imperfection exists in the matching between publications and subscriptions, which we refer to as uncertainty [1]. *Uncertainty* concerns the state of knowledge about the relationship between the world and the statement about the world. All publish/subscribe systems developed to date are based on the assumption that a match between a subscription and a publication is either true or false. However, it is difficult to decide whether a publication matches a subscription involving imprecision in the publication and the subscription. We call the imperfection inherent to the matching problem uncertainty.

To illustrate the difference between imprecision and uncertainty, consider these two examples: (1) Charles is a tall guy, and I am sure of it. (2) Charles is six feet tall, but I am not sure of it. The height of Charles is imprecise in the former case, but it is certain.

[1]The *uncertainties* used in the title is more general. In this paragraph, *uncertainty* has a specific meaning and is a sub-class of the general one.

In the latter statement, the height is precise but uncertain.

The goal of this thesis is to design publish/subscribe systems supporting applications that involve representing, processing and filtering the above three types of imperfect information at internet scale.

The challenges to support imperfect information and semantics at that scale raises several challenging questions. First, we need a publish/subscribe model to express imperfect information and semantic in publications and subscriptions. Second, a corresponding approximate matching and semantic matching definition is required for the new model. Third, we need efficient algorithms to perform filtering and prediction for the model so that it can be applied to process the rapidly increasing information on the Internet. Finally, the approach needs to be scalable to large numbers of subscribers and high publishing rates.

To address the above problem, this thesis extends subscription and publication languages to incorporate the expression of imprecision and semantic at the language level, develops a matching mechanism to support the processing of the extended language in publish/subscribe systems, and develops a prediction mechanism to report future matches with a certain probability.

1.3 Contributions

To address these challenges, we have developed publish/subscribe system models and implementations that enable the representation, filtering, delivery, and processing of uncertain data.

The main contributions of this thesis are:

1. The definition of a highly flexible publish/subscribe system model supporting the expression of uncertainties in the subscription language model and the publication data model. This model supports all combinations of approximate and crisp

subscriptions and publications and is fully implemented. The model allows for fine-grained adjustment to express different users' subjective perception of the concepts represented and allows to tune the matching relations [57, 59, 60].

2. The expression of imprecision in subscriptions and publications raises questions regarding the matching and routing of approximate subscriptions and approximate publications. This thesis articulates the approximate publish/subscribe content-based routing problems and develops algorithms for *matching*, *covering* (containment) and *merging*. The algorithms developed present an implementation of the key routing computations that make up a publish/subscribe content-based router supporting processing imprecise data.

3. We develop density estimation algorithm to learn the possibility distribution for set membership functions used in the approximate publish/subscribe model.

4. An original publish/subscribe system model is developed to support large-volume graph-based content distribution from diverse sources. The model also allows the use of an ontology to specify class taxonomy as semantic information about the data [73, 61].

5. We develop a pipelined filtering algorithm for processing graph-based data and queries on a compute cluster. We also develop 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.

6. A thorough experimental evaluation is presented for each model to demonstrate the filtering performance of our algorithms and validate that the proposed systems offers scalability with the increase in the number of users while maintaining an

efficient filtering rate.

7. A demonstration simulating a real world application is implemented for each system. It integrates a web server through which users can submit their information including subscriptions and publications to the system and a filtering engine running in the background to support the matching and notification [56, 58, 72].

8. A novel model is proposed to support expressive time operators and future matching prediction for complex event processing. This model represents each composite subscription as a finite state machine (also a Markov model) and maintains a Markov probability distribution based on historical data. An efficient matching algorithm and predicting algorithm is developed on top of this model to offer the ability to report the future matches with a probability based on the current match status.

1.4 Organization

The thesis is organized as follows.

In Chapter 2, we survey other research related to this thesis. We discuss publish/subscribe systems, filtering graph-based data, content-based routing and imperfect data management.

The necessary background material this thesis is based on, namely possibility theory, fuzzy set theory and Markov chains are introduced in Chapter 3.

Chapter 4 proposes a new publish/subscribe model and a corresponding distributed content-based publish/subscribe architecture. The model is based on possibility theory and fuzzy set theory to process uncertainties for both subscriptions and publications. We also develop matching, covering (containment) and merging algorithms for this model. The algorithms developed present an implementation of the key routing computations that make up a publish/subscribe content-based router supporting processing of imprecise data.

Chapter 5 proposes a publish/subscribe system model to support large-volume graph-based content distribution from diverse sources. Especially, the model allows the use of an ontology to specify a class taxonomy as semantic information about the data so as to enable semantic matching. We also develop a pipelined filtering algorithm, along with two indexing algorithms for processing graph-based data and queries on a compute cluster.

Chapter 6 proposes a model based on finite state machines to support expressive time operators for complex event processing. For this model, we proposes a data structure and matching algorithm to efficiently evaluating a large number of complex event patterns. Furthermore, along with the matching procedure, a Markov model is maintained in conjunction with the finite state machines to offer the ability to predict the occurrence of complex events associated with a probability based on a partial match.

Finally, conclusions and future work are presented in Chapter 7.

Chapter 2

Related Work

In this chapter, we will review the significant related work from the aspects of publish/subscribe systems, content-based routing and uncertain data management.

2.1 Publish/Subscribe Systems

2.1.1 Publish/Subscribe Research Prototypes

Much work has been devoted to developing publish/subscribe systems and event notification services such as ELVIN [86], Gryphon [3], LeSubscribe [34], READY Salamander [88] and SIENA [16]. These systems are different in the subscription language and publication data model they offer and algorithms performing the matching task.

LeSubscribe [34] aims at publish/subscribe support for web-based applications. It focuses on the algorithmic efficiency in supporting millions of subscriptions and high event processing rates. The language and data model are based on an LDAP-like semi-structured data model for expressing subscriptions and publications. In this system, a subscription is a conjunction of predicates each of which is a triplet (attribute, operator, value). Supported relational operators include $<, \leq, \neq, \geq, >$. This system supports both push and pull based information dissemination. The matching engine of LeSubscribe

falls within the class of two-step matching algorithms - a predicate matching step and a subscription evaluation step. In the first step, all predicates are matched against the publication; then subscriptions are evaluated in the second step based on the set of matched predicates.

Instead of two-step matching algorithms, Gryphon [3] uses a tree-based data structure to index subscriptions which leads to another category of matching algorithms. In Gryphon, all subscriptions are preprocessed into a tree where each non-leaf node is a test for one attribute and the edges derived from that node represent different results. During matching, the incoming publication goes down through the branch it matches until arriving at the leaf nodes containing the matched subscriptions. Another approach using a tree-based algorithm is Binary Decision Diagrams (BDDs) [23]. In this model, each subscription is a boolean function represented by a BDD. This approach is distinguished in two aspects: one is that it can support any boolean formula; the other is that overlapping subscription expressions are operated only once if the variable ordering was chosen properly.

Elvin [86] is a content-based notification/messaging service that targets application integration environments and monitoring of distributed systems. ELVIN supports a more expressive subscription language which is created as strings. Subscriptions contain powerful string processing functions and operators on built-in data types covering integer, string and boolean relations. In addition to the traditional comparison operators like $<, \leq, =, \neq, >, \geq$, Elvin also supports operations such as matching extended regular expressions with strings.

READY [88], a research project project led by the AT&T research lab, is an implementation of the CORBA Notification Service. The specific features of READY, which are not offered by existing commercial products, include: information consumer specifications that can be matched over both single and compound event pattern; and quality of service (QoS) that is managed by providing ordering properties for event delivery.

Rete [53] is an efficient pattern matching algorithm for implementing production rule systems. A Rete-based expert system builds a network of nodes, where each node (except the root) corresponds to a pattern occurring in the left-hand-side (the condition part) of a rule. The path from the root node to a leaf node defines a complete rule left-hand-side. Each node has a memory of facts which satisfy that pattern. This structure is essentially a generalized trie. As new facts are asserted or modified, they propagate along the network, causing nodes to be annotated when that fact matches that pattern. When a fact or combination of facts causes all of the patterns for a given rule to be satisfied, a leaf node is reached and the corresponding rule is triggered.

There are two common properties to all current systems. First is the crisp matching semantic – neither subscriptions nor publications can express uncertain information and a match is either established or not. A gradual match, that is expressed as a confidence, a degree of match, or a probability, does not exist in any previously studied model. Second, in all the current publish/subscribe systems, the matching result is reported only after evaluating the whole subscription pattern. There is no publish/subscribe model that can predict the occurrence of a future matched pattern associated with a probability based on a partial match (i.e., a partially observed state.)

The Middleware System Research Group at the University of Toronto is working on a so-called Toronto Publish/Subscribe System Family, including Approximate Matching-based ToPSS [59], Semantic Matching-based ToPSS [71], Location-aware ToPSS [15], Peer-to-Peer Networks Supporting ToPSS [87], Mobility Supporting ToPSS [14], XML/XPath Matching-based ToPSS [46], Ad Hoc Networks Supporting ToPSS [74] and Federated Publish/Subscribe (PADRES [38, 54, 55]). They are publish/subscribe systems working in different scenarios.

2.1.2 Filtering of Semi-Structured Data

The publish/subscribe matching problem has been investigated extensively. Attribute value pairs are used by the above models to represent publications, while conjunctions of predicates with standard relational operators are used to represent subscriptions. Tree-based XML is used by [5, 28, 47] as the data model and XPATH as the query language.

XTrie [18] 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.* [41] 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.

More specifically, there are several challenges for graph-structured data that cannot be addressed by the above approaches. First, a cycle may exist in a graph or one node has multiple parents, which cannot be solved by XML filter. Second, there is no concept of root in a graph, thus there is no start point when applying XML filter. Third, the edge between two nodes contains semantics meaning except the role of connecting. Fourth, our approach also support ontology evaluation in addition to the structure and predicate matching. In all, the techniques for XML document filtering is not applicable to graph-structured data.

Our matching algorithm is designed to tackle the above challenges of graph-structured data filtering. If applied to filtering tree-structured data,[1] it incurs additional cost compared to other XPath matching algorithms such as [5, 28].

RSS is emerging as the main streaming technology on the Internet, and consequently, we developed a schema operations for efficient filtering of RSS [73]. Prior to this, we

[1] 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.

developed S-ToPSS [71] that extends the traditional predicate-based data model with capabilities to process syntactically different, but semantically-equivalent information. S-ToPSS uses an ontology to be able to deal with syntactically disparate queries and tuples. The ontology, which can include synonyms, a taxonomy and transformation rules, was specified using S-ToPSS specific methods. On the other hand, the data model used in [73] is based on directed graph in general and RDF in particular. Use of RDF makes it possible to use ontologies built on top of RDF using languages such as RDFS and OWL. Currently, our query semantics allow type constraints on nodes of a graph, where the type information is represented as a class hierarchy.

OPS [92] is another set of ontology-based schema operations whose data model is also based on RDF. OPS uses a very general subgraph isomorphism algorithm to implement satisfy. However, this approach, as the results show, unnecessarily increases the filtering complexity because it assumes that any node of the tuple graph can map to any node of the query graph. We compare the performance to OPS and show that our implementation of satisfy always outperforms OPS.

Racer [42] is a data model is based on description logics. Because OWL, an ontology system built using RDF, is based on description logics, Racer can be used for RDF filtering. Racer schema operations do not scale as well as ours (satisfy times are in the order of 10s of seconds even for very simple queries) primarily because of its expressiveness.

2.1.3 Indexing and Routing

SIENA (Scalable Internet Event Notification Architectures) [16] is prototype of a pub/sub event-notification service which is based on content-based networking services and focuses on the routing of subscriptions and publications in a distributed environment. The advantage of this infrastructure is that it maximizes expressiveness in the selection mechanism without sacrificing scalability in the delivery mechanism. But its subscription language does not allow ontology support, which limits its use in filtering information of semantic

web.

SemCast [69] present a semantic multicast approach that split the incoming data streams based on the overlapping of its content. 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 won't reduce the amount of messages routed in the network. Compare to our approach, Semcast does not support imperfect merging which results in a significant decrease of traffic, as we have shown.

Yi-Min Wang *et al* [93] proposed two approaches for subscription partitioning and routing, one based on partitioning the event 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 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.

CREAM [21] 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 query and data model in CREAM, is predicate-based. Unlike our schema operations, 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 CREAMs scalability such as the one we present.

2.1.4 Industrial Standards

There have been a number of standardization efforts on middleware architectures and distributed system interfaces to promote interpretability. The Common Object Request Broker Architecture (CORBA) is a middleware architecture standardized by the Object Management Group (OMG). The CORBA Event Service [66] and Notification Services specifications [68] augment the CORBA middleware platform with event-based messaging capabilities. The Java Message Service (JMS) is the standard Java API for message-oriented middleware proposed by Sun Microsystems to add messaging integration capabilities into the J2EE platform.

The CORBA Event Service [66] specification defines an indirect channel-based event transport for distributed object frameworks. An event channel decouples event suppliers and consumers. Suppliers generate events and place them onto a channel. Consumers obtain events from the channel. Two serious limitations of the Event Service Specification are that it only supports limited event-filtering capabilities, and cannot be configured to support different qualities of service. Most Event Service implementations deliver all events that are sent to a particular channel to all consumers connected to that channel on a best-effort basis.

A primary goal of the Notification Service [68] is to enhance the Event Service by introducing the concepts of event filtering and quality of service specifications. Clients of the Notification Service can subscribe to events by associating filter objects with the proxies through which the clients communicate with event channels. These filter objects encapsulate specific constraints on the events to be delivered to the client. Furthermore, the Notification Service enables each channel, each connection, and each message to be configured to support the desired quality of service with respect to delivery guarantees, event aging characteristics, and event priorities.

The Java Messaging Service (JMS) [65] is an API for enterprise messaging created by Sun Microsystems. JMS is not a messaging system itself. It is an abstraction of the

interfaces and classes needed by messaging clients when communicating with messaging systems. JMS provides both publish/subscribe and point-to-point messaging models. Under the JMS publish/subscribe model, publishers can send a message to many consumers through a virtual channel called a topic. All messages addressed to a topic are delivered to all the topic's subscribers. The message delivery is push-based and no polling is required. The point-to-point messaging model uses queues to store and forward messages from suppliers to consumers. A given queue may have multiple receivers, but only one receiver may consume each message. It is a one-to-one communication model.

2.1.5 Continuous Queries

Continuous queries are issued once and are logically run continuously over a database. Sometimes they are referred to as queries for future data, because data included in the result set may not exist at the time when the query was created, but will be created in the future. Traditional one-time queries, in contrast, run only once to completion and return a result based on the current data sets. The notion of continuous queries is similar to subscriptions in publish/ subscribe systems. A publish/subscribe system will continuously evaluate a subscription against the new incoming publication stream, until the subscription is removed from the system. Two research projects, Open CQ [62] and NiagaraCQ [19], support continuous queries for monitoring persistent datasets spread over a wide-area network. Open CQ uses a query processing algorithm based on incremental view maintenance. NiagaraCQ addresses scalability in number of queries by proposing techniques for grouping continuous queries for efficient evaluation. STREAM (Stanford stream data management [8]) is a research project at Stanford that focuses on query processing of continuous queries over data streams. It provides a general and flexible architecture for query processing in the presence of data streams.

2.2 Uncertain and Imprecision Information Management

A number of techniques, including, probability theory, fuzzy set theory, and a general similarity metric-based approach have been applied to model uncertainty and imprecision in query and data. A full exploration would go beyond the scope of this thesis. We discuss a number of representative examples.

Nowadays, applications of fuzzy logic are found in many fields, including databases [13, 75, 94, 20], and expert systems [52].

Ronald Fagin uses the operations on fuzzy sets to combine fuzzy information from multiple systems [35, 36]. In the database, each object has several attributes and a grade of membership is assigned to each attribute for measurement. To determine the top k objects that have the highest overall grades, Fagin gives an efficient algorithm ("Fagin Algorithm", or FA) [36] to merge several sorted lists based on rating of objects from different multimedia systems. For some monotone aggregation functions, FA is optimal with high probability in the worst case. A more elegant and remarkably simple algorithm ("Threshold Algorithm". or TA) is proposed in [37] which is proved optimal in a much stronger sense than FA. The authors show that TA is essentially optimal, not just for some monotone aggregation functions, but for all of them; and not just in a high-probability worst-case sense, but over every database. FA requires large buffers whose sizes may grow unboundedly as the database size grows. Unlike FA, TA allows early stopping, which yields, in a precise sense, an approximate version of the top k answers.

Another application using the knowledge of fuzzy sets is in [95]. Wolski *et al.* propose a fuzzy trigger to incorporate imprecise reasoning in active database. The rules that control the event-condition action are modelled by fuzzy membership functions. This work proposes two trigger models. C-fuzzy trigger involves fuzzy inference only in the process of evaluation of the condition. If actions are also expressed in fuzzy terms and integrated

with the condition part, it leads to the CA-fuzzy trigger. [94] introduces a retrieval language based on fuzzy logic and addresses the problem of retrieving using relevance feedback, a method that automatically adapts the representation of the underlying fuzzy set.

With the ever increasing of web documents generation, an efficient information filtering system based on the similarity between the documents and users' profiles is in great need. Yan and Molina [98] suggested indexed structure under vector space model and an algorithm to compute the similarity, which is a distance function of match between a document and a profile based on the "importance" value of certain attributes. In this model, a document is identified by a set of terms represented as an multi-dimensional vector. Each term is assigned a weight as the statistical importance indication. Profiles appears just like documents consisting a list of terms, each with a weight. The similarity degree between a document-profile pair is measured by applying a cosine measure which is a dot time operation if both documents and profiles are normalized by their lengths. Based on this similarity measurement, several index refinements are devised to improve I/O and CPU processing time.

Fuhr introduced a probabilistic relational algebra in [39] to represent imprecise attribute values and integrate vague queries in database system. Probabilistic databases are currently an active area of research. A probabilistic database is an uncertain database in which the possible worlds have associated probabilities. While there are currently no commercial probabilistic database systems, several research prototypes exist including Trio [12], Orion [81], MystiQ [27] and MayBMS [6].

Trio [12] is a new kind of database management system which integrates data, uncertainty of the data, and data lineage together. Trio is based on an extended relational model called ULDBs, and it supports a SQL-based query language called TriQL.

The ORION database system (previously known as U-DBMS) [81], developed by the database group of Purdue University, is a state-of-the-art uncertain database manage-

ment system with built-in support for probabilistic data as first class data types. In contrast to other uncertain databases, Orion supports both attribute and tuple uncertainty with arbitrary correlations. This enables the database engine to handle both discrete and continuous pdfs in a natural and accurate manner. The underlying model is closed under the basic relational operators and is consistent with Possible Worlds Semantics.

MystiQ [27] is a system that uses a probabilistic data model to find answers in large numbers of data sources exhibiting various kinds of imprecisions. Moreover, users sometimes want to ask complex, structurally rich queries, using query constructs typically found in SQL queries: joins, subqueries, existential/universal quantifiers, aggregate and group-by queries. The goal of MystiQ is to develop efficient query processing techniques for finding answers in large probabilistic databases.

MayBMS [6] is a state-of-the-art probabilistic database management system that has been built as an extension of Postgres, an open-source relational database management system. MayBMS uses probabilistic versions of conditional tables as the representation system, but in a form engineered for admitting the efficient evaluation and automatic optimization of most operations of the language using robust and mature relational database technology.

Although there is a large amount of related work involving representation and processing of uncertainties in databases and information management systems, none has studied the use of possibility theory and fuzzy set theory to model uncertainty in the language and data model of publish/subscribe systems and apply them into content-based routing for event distribution. This thesis is the first to develop an approximate covering and merging algorithm using possibility theory and fuzzy set theory to increase the system scalability and capture the imprecision inherent to many application domains.

Chapter 3

Background

In this chapter, we describe the publish/subscribe interaction model that is central to dealing with uncertain information. We also gives an overview of key theories used in our work, including fuzzy set theory [49], possibility theory [31] and markov model [63] in probability theory, to model and manage the uncertain data. The detailed discussion can be found in [49, 31, 63].

3.1 Overview of Publish/Subscribe System

A new data processing paradigm – publish/subscribe – is becoming increasingly popular for information dissemination applications. Example applications range from selective information dissemination, online shopping, online auctioning [70] to location-based services [15, 97] and sensor networks [90], to just name a few.

Publish/subscribe systems anonymously interconnect information providers with information consumers in a distributed environment. Information providers publish information in the form of publications (or events) and information consumers subscribe their interests in the form of subscriptions. The publish/subscribe system matches events with subscriptions and ensures the timely notification of subscribers upon event occurrence. Figure 3.1 shows the paradigm of publish/subscribe systems.

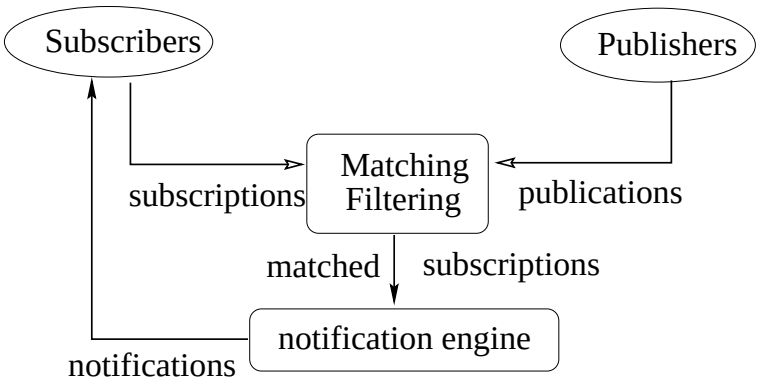


Figure 3.1: Publish/Subscribe Paradigm

Events are published in the form of publications and users' interests are subscribed in the form of subscriptions. A publication describes the attributes of a real word artifact. A subscription defines a user's interest through a list of predicates, where each predicate is a constraint on an attribute domain. The matching problem is to filter all satisfied subscriptions whose constraints are matched by an incoming publication.

Publications in the publish/subscribe system can be seen as data items (e.g., tuples, columns, or tables) in a relational database model and subscriptions closely resemble database queries. Therefore, publish/subscribe systems solve a problem inverse to database query processing.

The different ways of specifying the events of interest have led to several subscription schemes. Based on different subscription schemes, publish/subscribe system can be classified as topic-based, content-based and type-based. The earliest publish/subscribe scheme is based on the notion of topics or subjects, and is implemented by many industrial strength solutions (e.g., [4, 25, 82, 89]). The content-based (or property-based [79]) publish/subscribe variant improves on topics by introducing a subscription scheme based on the actual content of the considered events. In other terms, events are not classified according to some pre-defined external criterion (e.g., topic name), but according to the properties of the events themselves. Such properties can be internal attributes of data structures carrying events, as in Gryphon [10], Siena [16], Elvin [86], and Jedi [26], or

meta-data associated to events, as in the Java Messaging Service [44]. Replacing the namebased topic classification model by a scheme that filters events according to their type [32] leads to type-based publish/subscribe.

Publish/subscribe is a messaging paradigm and an information management methodology, and is applicable in many application domains. Selective information dissemination is the class of distributed applications that distributes information according to some restrictions or conditions. A more general form of data subscription is exemplified by the emerging peer-to-peer file sharing and publishing systems, such as Napster., Gnutella, Mojo Notion, Free Haven [30], and Freenet [22]. These systems are forms of publish/subscribe systems, where the broker component is physically distributed. There are many other applications to which the publish/subscribe paradigm is applicable, such as workflow management [26], intraenterprise process automation, supply chain management, enterprise application integration [11], and network monitoring.

3.2 Theories of Uncertainties

A key question in our work is how to express and process uncertainty in publish/subscribe systems. A simple method to express uncertainty about an imprecisely known value is to define it as an interval. For example, the interval [50, 150] would be reasonable to represent the age of a piece of "post-modern" painting in an online auction. In a crisp system, it needs two predicates to represent this interval: $(age \geq 50)$ and $(age \leq 150)$. Moreover, this method imposes a sharp boundary to differentiate members belonging to the set of post-modern paintings from non-members. A painting which was created 49 years ago may satisfy the subscriber, but it won't be delivered to the subscriber since it is out of the domain of the interval [50,150]. To overcome this limitation, fuzzy set theory [49] and possibility theory [31] have been developed. The publish/subscribe model we are introducing is based on these theories to model uncertainty in publications and

subscriptions.

3.2.1 Fuzzy Set Theory

Sharp boundaries that differentiate between objects belonging to a set versus objects not belonging to a set can be eliminated by introducing degrees of membership. This is the approach taken by fuzzy set theory.

Definition: A fuzzy set $\tilde{M}$ on a universal set U is a set that specifies for each element x of U a degree of membership to the fuzzy set $\tilde{M}$. It is defined by a membership function (a.k.a. characteristic function),

$$\mu_{\tilde{M}} : U \rightarrow [0, 1]$$

that specifies for each $x \in U$ its degree of membership $\mu_M(x)$ to the fuzzy set $\tilde{M}$.□

The membership function is a generalization of the characteristic function in classic set theory. It allows to express gradual set membership. For example, we can define a possible membership function for the fuzzy set of "post-modern paintings"as shown in Figure 3.2, where the domain ranges over the possible ages in the given application context. We use membership functions to represent predicates in subscriptions that constraint uncertain and vague concepts, such as "price is cheap" "age is old", and "location is close to".

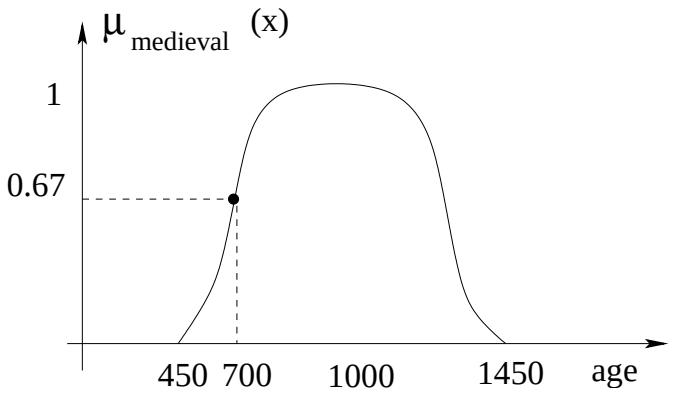


Figure 3.2: The membership function representing "post-modern paintings".

There are many possible function representations to express gradual set membership. Here, we use a parametric representation as suggested by many authors [31, 49]. The membership function of a fuzzy set $\tilde{M}$ can be described with a pair of functions, defined on $\Re^+ \rightarrow [0,1]$, denoted by L and R, such that L (and also R) is monotonically increasing (and monotonically decreasing) and is upper semi-continuous ($u.s.c$). This function pair and four parameters $(\underline{m}, \overline{m}, \alpha, \beta) \in \Re^2 \cup \{+\infty, -\infty\}$ define the membership function of a fuzzy set $\tilde{M}$ as follows:

$$\mu_M(u) = \begin{cases} L(u) & \forall u \in [\underline{m} - \alpha, \underline{m}] \\ 1 & \forall u \in [\underline{m}, \overline{m}] \\ R(u) & \forall u \in [\overline{m}, \overline{m} + \beta] \end{cases}$$

A fuzzy set is characterized by its membership function, so without ambiguity we can say $\tilde{M}$ is defined by $\mu_M(u) = (\underline{m}, \overline{m}, \alpha, \beta)_{LR}(u)$. This representation can be used to model a wide range of different gradual set membership relations (e.g., bell-shaped, trapezoidal, triangular etc.)

Definition: $[\underline{m}, \overline{m}]$ is the *core* of fuzzy set $\tilde{M}$, denoted by $\dot{\mu}_M$. $\underline{m}$ and $\overline{m}$ are referred to the *lower* and *upper model values* of $\tilde{M}$, respectively. The *support* of a fuzzy set $\tilde{M}$, denoted by $S(\mu_M)$, is the domain of values where $\mu_M(u) > 0$. If $\tilde{M}$ is of bounded support, then $S(\mu_M)$=$[\underline{m} - \alpha, \overline{m} + \beta]$. α and β are called the *left-hand spread* and the *right-hand spread*.□

There are many advantages of this representation. First, it eliminates the sharp boundaries inherent to a crisp or interval-based representation. Second, it is a very general representation and it is straight forward to implement. Third, this formalization is very expressive. Finally, it is easily extended to represent crisp sets defined through crisp constraints. In this case the membership function degenerates to the characteristic function as follows:

$$\mu_{p \geq v}(x) = \begin{cases} 1 & \text{if} \quad x \in [v, \infty) \\ 0 & \text{if} \quad x \notin [v, \infty) \end{cases}$$

Operations involving two or more fuzzy sets are generally defined by a mapping T that aggregates the membership functions as follows:

$$\mu_{\text{op}(A_1,...,A_n)}(x) = T(\mu_{A_1}(x), ..., \mu_{A_n}(x))$$

Intersection, union, and other set operations are defined in this manner. The operator T is referred to as a triangular norm (T-norm). T-norms that model set intersection must satisfy the following axioms (a generalization from classical set theory): $T(0,0) = 0, T(a,1) = T(1,a) = a$ (boundary condition), $T(a,b) \leq T(c,d)$ *if* $a \leq c$ *and* $b \leq d$ (monotonicity), $T(a,b) = T(b,a)$ (commutativity), and $T(a,T(b,c)) = T(T(a,b),c)$ (associativity).[1] Set union is defined and motivated in a similar manner. Operators that define set union are denoted as S-norms. Different S-norms and T-norms are used in the literature to represent set union and set intersection. A popular choice is to use maximum as union and minimum as intersection.

3.2.2 Possibility Theory

Possibility theory formally defines measures, which reflect users' subjective uncertainty of a given state of the world [31]. The measures express the confidence in the possibility that x is A. Possibility measures are based on possibility distributions, $\pi_A(x)$, that quantify these conditions.

A possibility measure that has values for each element in the universe of discourse can be interpreted by the membership function of a fuzzy set. For example, the antique shop has an art piece, where the age is described as post-modern:

$$\pi_{age-of-art-piece}(x) = \mu_{post-modern}(x).$$

[1]The first axiom imposes the correct generalization to crisp sets. The second axiom implies that a decrease in the membership values in A or B cannot produce an increase in the membership value in A intersection B. The third axiom indicates that the operator is indifferent to the order of the fuzzy sets to be combined. Finally, the fourth axiom allows us to take the intersection of any number of sets in any order of pairwise groupings.

We use two measures, referred to as *possibility measure*(Π) and *necessity measure*(N) to express the plausibility and necessity associated with each attribute in a publication. A possibility measure quantifies information about the plausibility of occurrence of the state represented by the attribute. If it is completely possible to be true then possibility is $\Pi(A) = 1$, if it is impossible then the possibility is $\Pi(A) = 0$; intermediate numbers between [0,1] are also admissible. A necessity measure is introduced to complement the information available about the state described by the attribute. It is associated with the degree with which the occurrence of A is certain. If an event A is sure to happen without any doubt, then necessity $N(A) = 1$.

The relationship between possibility and necessity satisfies[2]:

$$N(A) = 1 - \Pi(\overline{A})$$

$$\forall A, \Pi(A) \geq N(A).$$

3.2.3 Markov Model

In mathematics, a *Markov chain*, named after Andrey Markov, is a stochastic process with the Markov property. Having the Markov property means that, given the present state, future states are independent of the past states. In other words, the description of the present state fully captures all the information that could influence the future evolution of the process. Future states will be reached through a probabilistic process instead of a deterministic one.

At each step the system may change its state from the current state to another state, or remain in the same state, according to a certain probability distribution. The changes of state are called transitions, and the probabilities associated with various state-changes

[2]A possibility distribution is similar to a probability distribution. However, the difference between both is that there is no restriction that the sum of all possibilities on the whole universe must be equal to 1. Another difference is that probability distributions must be defined on disjoint subsets, but possibility distribution can be defined on distinct (as long as not equal) subsets. Thus, a possibility is a more general notion than a probability. [31]

are called transition probabilities.

Definition:A Markov chain is a sequence of random variables $X_1, X_2, X_3, \ldots$ with the Markov property, namely that, given the present state, the future and past states are independent. Formally,

$$\Pr(X_{n+1} = x | X_n = x_n, \ldots, X_1 = x_1) = \Pr(X_{n+1} = x | X_n = x_n).$$

The possible values of X_i form a countable set S called the state space of the chain.

Markov chains are often described by a directed graph, where the edges are labeled by the probabilities of going from one state to the other states.

A *finite state machine* can be used as a representation of a Markov chain. Assuming a sequence of independent and identically distributed input signals (for example, symbols from a binary alphabet chosen by coin tosses), if the machine is in state y at time n, then the probability that it moves to state x at time $n+1$ depends only on the current state.

In our work, we will make use the following properties to calculate the match probability between a publication and a subscription.

Definition: Define the probability of going from state i to state j in n time steps as

$$p_{ij}^{(n)} = \Pr(X_n = j \mid X_0 = i)$$

and the single-step transition as

$$p_{ij} = \Pr(X_1 = j \mid X_0 = i).$$

The n-step transition satisfies the Chapman-Kolmogorov equation, that for any k such that $0 < k < n$,

$$p_{ij}^{(n)} = \sum_{r \in S} p_{ir}^{(k)} p_{rj}^{(n-k)}.$$

Chapter 4

Uncertainties in Content-Based Publish/Subscribe System

In the publish/subscribe paradigm, exact knowledge of either specific subscriptions or publications is not always available. In this chapter, we propose a new publish/subscribe model A-ToPSS in a distributed content-based architecture. The A-ToPSS model is based on possibility theory and fuzzy set theory to process uncertainties for both subscriptions and publications. We also develop algorithms for matching, covering (containment) and merging. The algorithms developed present an implementation of the key routing computations that make up a publish/subscribe content-based router supporting processing imprecise data.

In this chapter, Section 4.1 describes the architecture of an approximate content-based router and reviews content-based routing. The approximate publish/subscribe data model is presented in Section 4.2. Section 4.3 develops the data structure and the algorithms underlying our matching, covering, merging and intersecting operations. A learning technique is proposed in Section 4.4 to tune the membership function and the possibility distribution for approximate subscriptions and publications. Section 4.5 and 4.6 presents the experimental evaluation and a demonstration of a real world application.

4.1 Architecture of Distributed Content-based Routing

4.1.1 Publish/Subscribe Router

A generic publish/subscribe router performs the following three operations: (1) Forwarding of advertisements, (2) Forwarding of subscriptions, and (3) Forwarding of publications. The details of the three operations are highly dependent on the expressiveness of the subscription and publication representation languages. However, the language-dependant features can be factored into four high-level operations: *matching*, *covering*, *merging* and *inserting*. In the following sections, we discuss these operations in more detail, while in this section we just describe their use in a content-based router.

Forwarding of Advertisements

Advertisements are used by publishers to announce the set of publications they are going to publish. Consequently, advertisements create routing paths for subscriptions from subscribers to publishers, whereas subscriptions build routing paths for publications from publishers to subscribers. Usually, both subscriptions and advertisements have the same formal representation. The following are the steps performed by the publish/subscribe router upon receiving an advertisement: (1) For the advertisement received, check if there are *covering* advertisements in the advertisement table. If there are, then we do not need to forward the advertisement. (2) If there is no covering advertisement, *insert* incoming advertisement in the advertisement table, and forward the advertisement to all neighbors. (3) Check if there are *intersecting* subscriptions in the subscription table. If there are, forward the intersecting subscriptions to the neighbor from which the advertisement was received.

When using advertisements, upon receiving a subscription, each broker forwards it

only to the neighbors that previously sent advertisements that intersect with the subscription. Thus, the subscriptions are forwarded only to the brokers that have potentially interesting publishers.

Forwarding Subscriptions

Subscription processing is similar to advertisement processing. Given two subscriptions s_1 and s_2, s_1 *covers* s_2 if and only if all the publications that match s_2 also match s_1. In other words, if we denote with E_1 and E_2 the set of publications that match subscription s_1 and s_2, respectively, then $E_2 \subseteq E_1$.

Informally, when a broker B receives a subscription s, it will send it to its neighbors if and only if it has not previously sent them another subscription s', that covers s. Broker B will receive all publications that match s, since it receives all publications that match s' and the publications that match s are a subset of the publications that match s'. The goal of subscription covering is to quench subscription propagation, thereby reducing network traffic and trimming the size of subscription (i.e. routing) tables.

The processing of the subscriptions at the pub/sub router proceeds as follows: (1) For each incoming subscription, check if there are *covering* subscriptions in the subscription table. If there are, then we do not need to forward the subscription. (2) If there is no covering subscription, *insert* incoming subscription in the subscription table. (3) If the routing table size reaches a limit, do merging. (4) Check if there are *intersecting* advertisements in the advertisement table. If there are, forward the subscription (or merged subscriptions) to those neighbors from which we received the matching advertisements.

Forwarding Publications

Finally, publications are processed as follows. For each incoming publication, check if there are *matching* subscriptions in the subscription table. If there are, forward the publication to all the neighbors from which each of the matching subscriptions was received.

4.1.2 Publish/Subscribe Broker Network

For content-based routing, publish/subscribe brokers are organized into a *content-based routing network* (see Figure 4.1). In the network, one of the most important problems is the routing of a publication to interested subscribers based on the content of the publication and the interests of subscribers expressed in subscriptions. There are a number of routing protocols proposed in the literature [16, 26, 9].

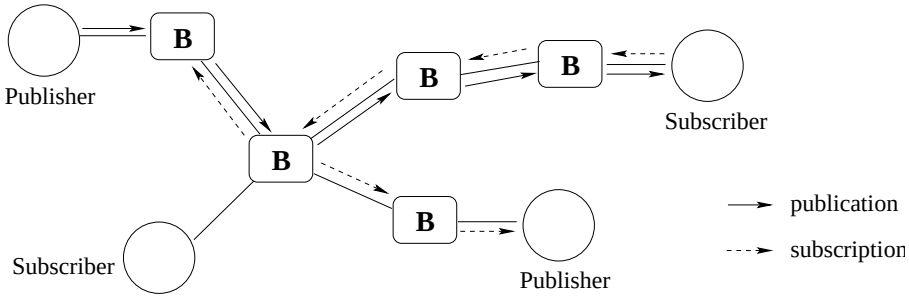


Figure 4.1: Hierarchical broker architecture

The protocols are based on building routing path between publishers and subscribers. Routing path are built through either flooding advertisements or subscriptions. *Advertisements* are messages used by publishers to announce the kind of information published. Advertisements follow the same representation as subscriptions and define the set of publications that are going to be published by a publisher. Advertisements are flooded throughout the routing network and are stored with each broker. Advertisements form a distributed advertisement tree. When a subscriber receives an advertisement, it sends subscriptions along the reverse path of the advertisement tree, if the subscription and advertisement match. These subscriptions are stored in the routing table of each broker along the subscription path, and result in a distributed subscription-multicast tree. Finally, publications are propagated along the reverse subscription routing path (i.e. the subscription-multicast tree rooted at the publisher) and delivered to interested subscribers. This scheme also works, if no advertisements are available, in which case

subscriptions are flooded throughout the network, against which publications are routed.

Each publish/subscribe broker processes and routes subscription messages independently. Often advertisements and subscriptions from different sources are in logical relationships, overlap, or are even identical. These relationships can be exploited to derive optimization for routing computations to reduce the routing table size, improve the matching performance at each node, and save network bandwidth.

4.1.3 Approximate Content-based Routing

In an approximate publish/subscribe system, uncertainties will be expressed and processed using fuzzy set and possibility distribution. As we mentioned above, the content-based routing in a distributed network involves the exploration on relationships among subscriptions and publications. The expression of uncertainty in subscriptions and publications raises questions regarding the matching and routing of crisp/approximate subscriptions with crisp/approximate publications. This chapter articulates the approximate publish/subscribe content-based routing problems and develops algorithms for *matching*, *covering* (containment) and *merging*. The algorithms developed present an implementation of the key routing computations that make up a publish/subscribe content-based router supporting processing imprecise data. In this chapter, we not only propose a model to express uncertainties in publish/subscribe system, but also describe a method for combining fuzzy concepts of all kinds to enable routing computations in an approximate publish/subscribe system.

4.2 Approximate Publish/Subscribe System Model

Our objective is to model uncertainties in subscriptions and publications, and to define semantics for the computation in approximate content-based routing with approximate subscriptions and publications.

4.2.1 Language and Data Model

Publication data model

Publications describe real world artifacts or describe states of interest through a set of attribute value pairs. In our model we account for the fact that for certain attributes precisely defined values may not be available or cannot be defined. In these cases we use a possibility distribution, as defined in Chapter 2, to represent the attributes' approximate values. These latter attributes are also referred to as approximate attributes, whereas attributes with exactly defined values are referred to as crisp attributes. However, our model integrates both kinds of attributes and does not distinguish between them. In the attribute value pair, "$(A,\ \pi(x))$", A is the attribute and π is the "value" – crisp or approximate. The possibility distribution, π, expresses that it is possible that the attribute, A, has the value, x, and quantifies this with a possibility degree, $\pi(x)$. The possibility distribution is defined by a fuzzy set that yields the possibility degree for the value x, as defined by the underlying fuzzy set's membership function. Crisp attributes, "$(A,\ x_0)$," are formalized analogously; π degenerates to a function that yields 1 for input x_0 and 0, otherwise. For short, we describe the attribute value pair, "$(A,\ \pi(x))$", simply as $\pi_A(x)$. A publication is thus defined as a vector of attribute value pairs:

$$p = (\pi_{A_1}(x), \pi_{A_2}(x), \cdots, \pi_{A_n}(x))$$

For example, an apartment that is advertised for rent as $((\text{size}, 60m^2), (\text{rent}, \mathit{cheap}))$ can be represented by a vector of attribute values as $P = ((\mathit{size}, \pi_{60}), (\mathit{rent}, \pi_{\mathit{cheap}}))$ where

$$\pi_{60}(x) = \begin{cases} 1 & \text{if} \quad x = 60; \\ 0 & \text{if} \quad x > 60 \text{ or } x < 60 \end{cases}$$

$$\pi_{\mathit{cheap}}(x) = \begin{cases} 1 & \text{if} \quad x \leq 1200; \\ 1 - \frac{x-1200}{300} & \text{if} \quad 1200 \leq x \leq 1500; \\ 0 & \text{if} \quad x > 1500 \end{cases}$$

Subscription language model

A subscription defines user's interests through a Boolean function over a number of crisp and approximate predicates. In the following we just refer to predicates, unless their approximate character is especially underlined. Each predicate expresses a constraint over a domain of values and is defined through a fuzzy set. In a predicate "(A, $\mu_A(x)$)", A is the attribute name and $\mu_A(x)$ is a membership function which defines a fuzzy set. The fuzzy set represents a fuzzy constraint over all possible values the attribute can take on. The predicate is evaluated by applying the membership function of the fuzzy set to the attribute's value in the publication. The resulting value constitutes the degree of match of the predicate. Note, this may be any value in the interval [0, 1]. Thus, the truth value (i.e., the degree of match) of each predicate is uniquely defined by $\mu_A(x)$. Crisp predicates can be defined in the same manner. In the crisp case, however, the membership function degenerates to the characteristic function over the set of values defined by the predicate (i.e., it yields 1 for all set members and 0 otherwise.)

Predicate matching degrees are aggregated in a subscription relation to yield a final degree of match for each subscription. We use R to represent the relation of the Boolean function over predicates defining a subscription. R represents conjunction, disjunction or any other Boolean operation connecting individual predicates. If we describe the predicate, "(A, $\mu_A(x)$)", simply as $\mu_A(x)$, a subscription, s, is formalized as follows:

$$s(x_1, \cdots, x_m) = R(\mu_{A_1}(x_1), \cdots, \mu_{A_m}(x_m)).$$

Here, the subscription, s, consists of m predicates and R defines the Boolean function relating all predicates in s. For example, s may be in conjunctive form: or disjunctive form, or any other form. R employs standard fuzzy set operators (cf. Chapter 3.2) to define the subscription relation. No limitation is imposed by the form of s. That is s may be any Boolean function, not necessarily in a normal form. Mathematically, R constitutes a function in the hyperspace defined over the Cartesian product of the domains of x_is.

For a given input vector $(x_1, \cdots, x_m)$ in this hyperspace, R yields the truth value of s for this input.

As a concrete example, let us define a subscription for a student who is looking for an apartment with 3 constraints specifies as (size is medium), (price is no more than \$450) and (age is not very old). These constraints can be represented by three membership functions as follows:

$$\mu_{\leq 450}(x) = \begin{cases} 1 & \text{if} \quad x \leq 450; \\ 0 & \text{if} \quad x > 450; \end{cases}$$

$$\mu_{medium}(x) = \begin{cases} 0 & \text{if} \quad x \leq 40; \\ \frac{x-40}{10} & \text{if} \quad 40 < x < 50; \\ 1 & \text{if} \quad 50 \leq x \leq 70; \\ 1 - \frac{x-70}{10} & \text{if} \quad 70 < x < 80; \\ 0 & \text{if} \quad x \geq 80; \end{cases}$$

$$\mu_{old}(x) = \begin{cases} 0 & \text{if} \quad x \leq 40; \\ \frac{x-40}{40} & \text{if} \quad 40 < x < 80; \\ 1 & \text{if} \quad x \geq 80; \end{cases}$$

Formally the subscription is represented by:

$$s(x_1, x_2, x_3) = \mu_{medium}(x_1) \wedge \mu_{\leq \$450}(x_2) \wedge (1 - \mu^2_{old}(x_3))$$

where $\wedge$ is used to model a conjunct. To demonstrate some features of fuzzy set theory, we use the negation of the membership function to define the qualifier "not" and the qualifier "very" through the squaring (i.e., damping) the fuzzy set's membership function.

4.2.2 Approximate Routing Computations

Approximate Matching

In the crisp publish/subscribe model, a subscription, either matches a publication, or does not match it. However, in the approximate model each subscription is assigned a degree

of match, depending on which, individual subscription can match a given publication more or less.

We define a match between a subscription and a publication as a *measure* of the *possibility* and *necessity* with which the publication satisfies the constraints expressed by a subscription. We use the pair (Π_{A_i}, N_{A_i}) to denote the evaluation of this measure. Technically speaking, the problem comes down to measuring the match between the predicate, $\mu_{A_i}(x_i)$, and the value, $\pi_{A_i}(x_i)$ for all i and for all x and aggregating the resulting values in the subscription relation R. This measure is taken by computing the intersection between μ_{A_i} and π_{A_i}.

Definition: The *possibility* and *necessity* of a match between μ and π is computed as

$$\Pi = \sup_x \min(\mu(x), \pi(x))$$

$$N = \inf_x \max(\mu(x), 1 - \pi(x)).$$

inf is the "infimum" and sup is the supremum. For finite domains both can be replaced by the "minimum" and the "maximum" operator, respectively. However, for infinite domains the more general inf/sup operators are required, which is the reason for using them in the above equations.

With this matching semantic a much larger number of subscriptions will match than before, as all matches with degrees greater than 0 are perspective matching candidates. A large number of slightly matching subscriptions may not be a useful idea, users may be overwhelmed with notifications about publications that only marginally meet their actual interests. For these reasons, the approximate matching model introduces a number of parameters to control the tolerance of a match on a very fine-granular basis. These parameters are the predicate thresholds θ_Π and θ_N and the subscription thresholds ω_Π and ω_N. With these parameters a publication matches a subscription, if its degrees of match evaluates to values larger than these thresholds.

The general form of subscriptions and publications is as follows:

$$s^{\omega_\Pi,\omega_N}(x_1, \cdots, x_m) = R(\mu_{A_1}^{\theta_{\Pi_{A_1}},\theta_{N_{A_1}}}(x_1), \cdots, \mu_{A_1}^{\theta_{\Pi_{A_m}},\theta_{N_{A_m}}}(x_m)),$$

$$p = (\pi_{A_1}(x_1), \pi_{A_2}(x_2), \cdots, \pi_{A_n}(x_n)).$$

Definition:: Formally, a publication, p, *matches* a subscription, s, if and only if:

$$\forall i \;\; \Pi_i \geq \theta_{\pi_{A_i}} \wedge N_i \geq \theta_{N_{A_i}} \wedge$$

$$R(\mu_{A_1}^{\theta_{\Pi_{A_1}}, \theta_{N_{A_1}}}(x_1), \cdots, \mu_{A_1}^{\theta_{\Pi_{A_m}}, \theta_{N_{A_m}}}(x_m)) \geq \omega_\Pi \wedge$$

$$R(\mu_{A_1}^{\theta_{\Pi_{A_1}}, \theta_{N_{A_1}}}(x_1), \cdots, \mu_{A_1}^{\theta_{\Pi_{A_m}}, \theta_{N_{A_m}}}(x_m)) \geq \omega_N. \quad \square$$

Definition: The *approximate matching problem* can now be stated as follows. *Given a set of subscriptions S and a publication p identify all $s \in S$ such that s and p match with a degree of match greater than the thresholds defined on s.*

From the possibility and necessity computation equations, the following properties can be easily deduced.

Properties:

$$\forall x, \qquad \Pi(x) \geq N(x) \tag{4.1}$$

$$\Pi = 0 \;\; \Leftrightarrow \;\; S(\mu) \cap S(\pi) = \emptyset \tag{4.2}$$

$$\Pi = 1 \;\; \Leftrightarrow \;\; \exists x \in \dot{\mu} \cap \dot{\pi} \tag{4.3}$$

$$N = 0 \;\; \Leftrightarrow \;\; \exists x \in \overline{S(\mu)} \cap \dot{\pi} \tag{4.4}$$

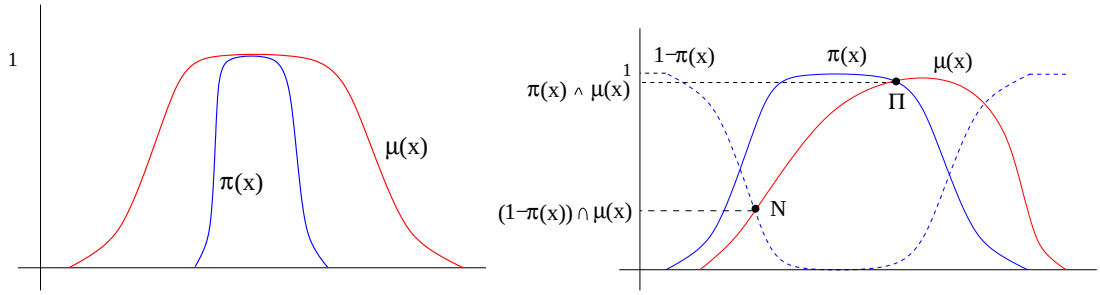
$$N = 1 \;\; \Leftrightarrow \;\; S(\pi) \subseteq \dot{\mu} \tag{4.5}$$

These properties are exploited in the algorithm to optimize its performance (cf. Section 4.3). The properties relate characteristics about the support and the core of the possibility distribution and fuzzy set to infer the degree of match with less computation. These properties are graphically illustrated in Figure 4.2. Figure 4.2(a) and Figure 4.2(d) illustrate property (4) and (5). Figure 4.2(c) illustrates property (3). Figure 4.2(b) illustrates the most general case, where $0 \leq \Pi \leq 1$ and $0 \leq N \leq 1$.

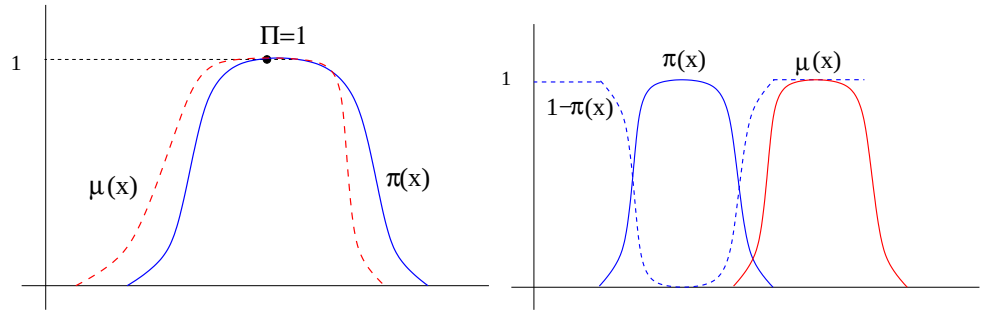
The possibility measure, Π, represents the degree of match and, its dual measure, N, represents the degree of no-match (cf. discussion Section 2). From Property 1, above,

it follows that the possibility, Π, is always greater or equal to the necessity, N. The subjective interpretation of this is that an optimistic subscriber would count on the leaner possibility measure, while, a pessimistic subscriber would count on the stricter necessity measure.

Finally note, that for crisp attributes, "(A, x_0)", the possibility distribution function π yields 1 for x_0 and 0, otherwise. So the intersection of π and μ can only occur at the point x_0, which is the value $\mu(x_0)$.



(a) $N = 1$ (b) $0 < N < 1$, $1 < \Pi < 1$



(c) $\Pi = 1$ (d) $N = 0$

Figure 4.2: Cases of possibility and necessity measure

Discussion of Alternative Matching Semantic:

Intuitively speaking, the ratio of the area of overlap between π and μ over the whole area of π may seem like an alternative measure to evaluate the degree of match between predicates and values. An interpretation of this ratio could be the assessment of how the

domain of π can satisfy μ. However, this method is not sufficient, as there exist situation in which subscriptions match only to a small degree, but the degree of match computed by this method is 1. Consider the example in Figure 4.3. The domain of the fuzzy set defining the approximate attribute in publication, π, is totally contained inside μ and it is completely covered by μ. It seems that all the values of the domain of discourse would satisfy the predicate defined by μ over this domain, thus yielding a degree of match of 1. However, consider the price \$60, its membership in π is 0.1, its membership in μ is 0.5.

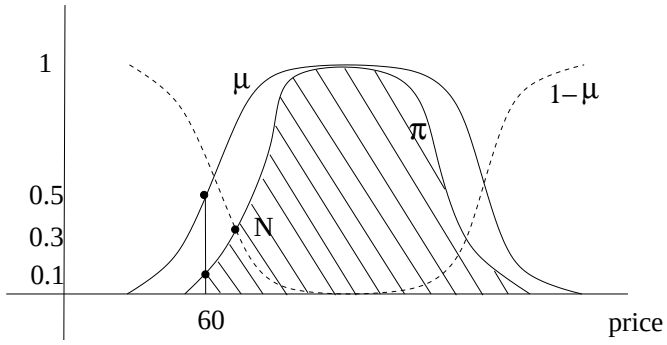


Figure 4.3: Degree of match defined as ratio of overlap

It is still possible that the price, the publisher observes is \$60, though this possibility is rated as only 0.1. The subscription matches with this price with a degree of match of 0.5 (as resulting from the application of the membership function at the point 60), but not with degree 1. Therefore, it is not appropriate to define the matching degree as 1 in this situation. On the other hand, possibility and necessity measures solve this problem. It is possible that the value provided by the publication satisfies the subscription, the *possibility degree* is 1. But it is not necessarily the case; so according to the formula above, the *necessity degree* is only 0.3.

Approximate Covering

The goal of subscription covering is to quench subscription propagation, thereby reducing network traffic and trimming the size of subscription (i.e. routing) tables.

Given two subscriptions s_1 and s_2, s_1 *covers* s_2 if and only if all the publications that match s_2 also match s_1. Based on the A-ToPSS subscription language model, a cover relation can be defined as follows: for two subscriptions s_1 and s_2, s_1 covers s_2, represented as $s_1 \succeq s_2$, iff for each predicate p_{1i} in s_1, there is a predicate p_{2j} in s_2 covered by p_{1i}. The predicate cover relation is defined based on the membership functions and thresholds: $p_1(A_1, \mu_1, \theta_{\Pi_1}, \theta_{N_1}) = \mu_{A_1}^{\theta_{\Pi_1}, \theta_{N_1}}(x)$ covers $p_2(A_2, \mu_2, \theta_{\Pi_2}, \theta_{N_2}) = \mu_{A_2}^{\theta_{\Pi_2}, \theta_{N_2}}(x)$ iff $A_1 = A_2$, $\mu_1(\mu_{A_1})$ covers $\mu_2(\mu_{A_2})$ and $\theta_{\Pi_1} \leq \theta_{\Pi_2}$, $\theta_{N_1} \leq \theta_{N_2}$. μ_1 covers μ_2 means that for any value x, the inequality $\mu_1(x) \geq \mu_2(x)$ is established. The condition on thresholds is to guarantee that the thresholds of the outside function won't filter out publications which match the inner function that has lower thresholds. Figure 4.4 shows an example of cover relation.

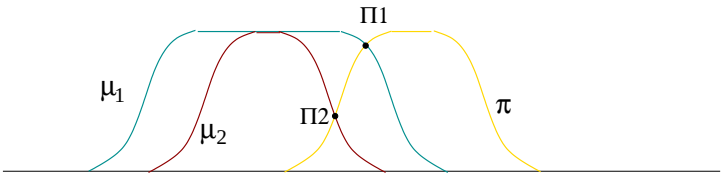


Figure 4.4: μ_1 covers μ_2, p_1 covers p_2 iff $\theta_{\Pi_1} \leq \theta_{\Pi_2}$ and $\theta_{N_1} \leq \theta_{N_2}$

Informally, when a broker B receives a subscription s, the broker will send the subscription s to its neighbors if and only if the broker has not previously sent them another subscription s', that covers s. If such s' exists, the broker won't send s to its neighbors. However, it will still receive all publications that match s, since it receives all publications that match s' and the publications that match s are a subset of the publications that match s'.

Covering-based routing results in the reduction of the routing table size without information loss, so that the performance of the matching algorithm can be improved and no redundant information is forwarded into the network.

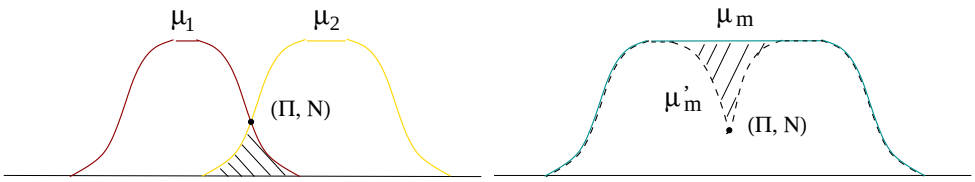
Approximate Merging

The merging technique is used for further minimizing the routing table size and network traffic overhead. It is an extension of the covering relation. Two subscriptions that are largely overlapping each other can be merged into a more general subscription and be forwarded into the network.

Subscriptions that have no cover relation may overlap with each other. These overlapping subscriptions can be merged into a new subscription, thus further reduce the size of the routing table and network traffic. A merged subscription s_M merging subscription s_1 and s_2 covers both s_1 and s_2, which is represented as $s_M \succeq s_1$ and $s_M \succeq s_2$. In other words, if $P(s_1)$ is the publication set matching s_1, $P(s_2)$ is the publication set matching s_2 and $P(s_M)$ is the publication set matching s_M, s_M is a merger of s_1 and s_2 iff $P(s_M) \supseteq P(s_1) \cup P(s_2)$. If any publication that matches s_M matches either s_1 or s_2, s_M is called a *perfect merger*; otherwise s_M is an *imperfect merger*. Imperfect merger may introduce false positives, that is, some publications that do not match any of the original subscriptions will match the imperfect merger and be forwarded into the network, thus increasing the network traffic overhead.

A subscription is a set of predicates. The merger of subscriptions should be defined based on merger of predicates. We first look at how to merge predicates. In A-ToPSS, a predicate is basically a membership function. Thus, the merged predicate should also be a membership function which defines a larger domain for the values of the constraint. Figure 4.5 shows an example of two overlapping membership functions and two possible merger choices. The merger of μ_1 and μ_2 could be either μ_m (the function with solid line) or μ'_m (the one with dashed line) in Figure 4.5(b). μ'_m is a perfect merger since it does not include any area which is not in original membership functions. But its representation is complicated and does not satisfy the membership function definition. To simplify the representation, we take μ_m, which is a concave function, as the merged membership function. It may introduce some false positives (when publications located

in the shaded area), but it is easy to represent and process.



(a) Overlapped subscriptions (b) Possible merger

Figure 4.5: How to merge overlapped subscriptions

Each predicate is also assigned a pair of matching thresholds to filter out slightly matching subscriptions. Thus the problem of merging predicates involve the policy on how to choose new matching thresholds for the merged predicates. Possible choices are taking *min*, *max* and *average* of the original thresholds. Use of *min* won't miss any correct matched publications, but may introduce more false positive. Use of *max* can reduce the false positives but some correct matched publications will be missed. *average* is a compromise way to balance the effects of these two choices. Also *average* is a better aggregation function for the case that two predicates overlapping each other largely, but have big difference on the thresholds. We leave the freedom of how to assign thresholds for merged predicates and only use R to represent the aggregation function on thresholds of two predicates. The effectiveness of different choices for this aggregation will be evaluated in the experiments.

Formally, the merger of predicates can be defined as: given two predicates $p_1 = (A_1, \mu_1, \theta_{\Pi_1}, \theta_{N_1})$ and $p_2 = (A_2, \mu_2, \theta_{\Pi_2}, \theta_{N_2})$ where $A_1 = A_2$ and μ_1, μ_2 are defined as follows:

$$\mu_1(x) = \begin{cases} L_1(x) & \forall x \in [\underline{m_1} - \alpha_1, \underline{m_1}] \\ 1 & \forall x \in [\underline{m_1}, \overline{m_1}] \\ R_1(x) & \forall x \in [\overline{m_1}, \overline{m_1} + \beta_1] \end{cases}$$

$$\mu_2(x) = \begin{cases} L_2(x) & \forall x \in [\underline{m_2} - \alpha_2, \underline{m_2}] \\ 1 & \forall x \in [\underline{m_2}, \overline{m_2}] \\ R_2(x) & \forall x \in [\overline{m_2}, \overline{m_2} + \beta_2] \end{cases}$$

The merged predicate of $p_m(A_m, \mu_m, \theta_{\Pi_m}, \theta_{N_m})$ is calculate as $A_m = A_1 = A_2$, $\theta_{\Pi_m} = R(\theta_{\Pi_1}, \theta_{\Pi_2})$, $\theta_{N_m} = R(\theta_{N_1}, \theta_{N_2})$ and

$$\mu_m(x) = \begin{cases} L_m(x) & \forall x \in [\min(\underline{m_1} - \alpha_1, \underline{m_2} - \alpha_2), \min(\underline{m_1}, \underline{m_2})] \\ 1 & \forall x \in [\min(\underline{m_1}, \underline{m_2}), \max(\overline{m_1}, \overline{m_2})] \\ R_m(x) & \forall x \in [\max(\overline{m_1}, \overline{m_2}), \max(\overline{m_1} + \beta_1, \overline{m_2} + \beta_2)] \end{cases}$$

With the definition of a predicate merger, we define the merger of subscriptions as follows. Given two subscriptions $s_i = \{p_{i1}, p_{i2}, \cdot, p_{ik}, \cdots, p_{im}\}$, $s_j = \{p_{j1}, p_{j2}, \cdots, p_{jk}, \cdots, p_{jn}\}$ where the first k predicates of s_i and s_j share the same attributes. The merger of s_i and s_j is s_m, $s_m = \{p_{m1}, p_{m2}, \cdots, p_{mk}\}$ where p_{ml} is the merger of p_{il} and $p_{jl}, l = 1, 2, \cdots, k$.

4.3 Data Structures and Algorithms

4.3.1 Data Structures

To exploit the overlap between subscriptions, we use a hash table to index predicates according to their attribute names, a predicate vector to store the degree of match for each predicate, a linked list associated with each predicate to record the subscriptions that contain it (or using an association bit matrix) and a subscription vector to keep track of the degree of match of each subscription. The overall data structure is depicted in Figure 4.6.

In Figure 4.6, a_i is the attribute name. Each predicate is represented by a pair (pid, μ). *pid* is the predicate ID and μ is the membership function to describe user's constraint on the attribute a_i. μ is represented by a list of parameters $(\underline{m}, \overline{m}, \alpha, \beta, L_m, R_m)$. L_m and R_m are the indexes into a function family indicating which functions are used for

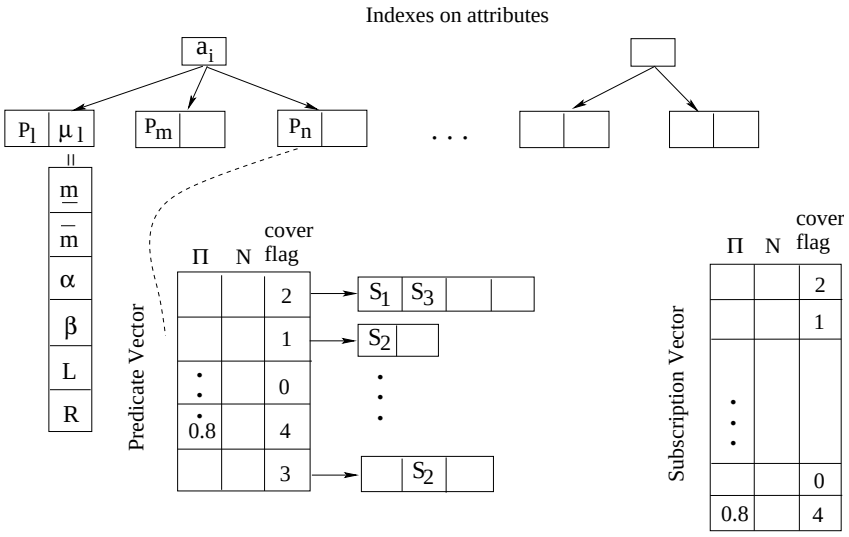


Figure 4.6: Data structures

left-hand spread and right-hand spread functions. The exact choice of these parameters depends on the real application. We use one predicate vector to store both thresholds (θ_Π, θ_N) and the matching degrees (Π, N) . A flag is used to indicate whether the numbers are thresholds or matching degrees. At first it stores the thresholds θ_Π and θ_N.

Each publication is a set of pairs $(attr, \pi)$ for different attributes. π is a function showing the possibility distribution of uncertain value. Similar to μ, π is represented as $(\underline{n}, \overline{n}, \gamma, \delta, L_n, R_n)$.

4.3.2 Approximate Matching Algorithms

The matching algorithm proceeds in two stages. First predicates are matched and, second, matching subscriptions are identified. This is a similar break-down as applied in many crisp matching algorithms.

Predicate evaluation: A publication is a set of pairs of $(attr, \pi)$ where $\pi = (\underline{n}, \overline{n}, \gamma, \delta, L_n, R_n)$. The attribute-name, $attr$, is used as the hash key to locate the corresponding predicate-table. Each predicate is stored only once in the system. Each predicate is in the form $(pid, attr, \mu, \theta_\Pi, \theta_N)$, where $\mu = (\underline{m}, \overline{m}, \alpha, \beta, L_m, R_m)$. The predicate evaluation computes the possibility and necessity of match for the given input attribute,

respectively and store them in a vector V_p. After all attributes of the given publication have been processed the matched degrees (i.e., each possibility and necessity) are used to derive matched subscriptions. Algorithm *predMatch* depicts the predicate matching algorithm.

Algorithm *predMatch*(e)

1. $V_p = 0$, $SatPreds = \emptyset$
2. **for** each attribute a_i in e
3. locate the corresponding index i in I
4. **for** each predicate $\mathrm{p}(a_i, \mu_i, \theta_{\Pi_i}, \theta_{N_i})$ reached by i
5. $V_p[p].\Pi = \sup \min(\mu_i, \pi_i)$
6. $V_p[p].N = \inf \max(\mu_i, \pi_i)$
7. **if** $V_p[p].\Pi > 0$ and $V_p[p].\Pi \geq \theta_{\Pi_i}$ and $V_p[p].N \geq \theta_{N_i}$
8. $SatPreds = SatPreds \cup \{p\}$
9. return $SatPreds$

Subscription evaluation: Subscriptions may be conjuncts of predicates, disjuncts of predicates, or normal forms. The algorithm we present for subscription evaluation works for either conjunctive or disjunctive subscriptions. To also process normal forms a further stage based on the truth values of subscription terms is required, which we don't present here (it is analogous to the subscription evaluation stage.) The algorithm calculates the degree of match, as expressed by a possibility measure and a necessity measure for each subscription based on the predicates matching degrees. Here we use minimum operation for conjunctive predicates. Other choice such as product could also be used. At the end of evaluation, we will compare the possibility and necessity of each subscription with user's thresholds ω_Π and ω_N, only return user the subscriptions whose degrees are larger. Algorithm *subMatch* depicts the detailed procedure. In the algorithm, V_p is predicate vector where stores all predicate matching degrees, V_s is the subscription vector and $List$ is an array of lists that store predicate subscription association;

Algorithm *subMatch*(Vp)

1. $V_s = 0$, $SatS = \emptyset$, $Count = 0$
2. **for** each $p \in Vp$ where $V_p[p].\Pi \geq p.\theta_\Pi$ and $V_p[p].N \geq p.\theta_N$
3. **for** each s in List[p]
4. **if** $Count[s] = 0$

5. **then** $V_s[s].\Pi = V_p[p].\Pi$
6. $V_s[s].N = V_p[p].N$
7. **else** $V_s[s].\Pi = \min(V_s[s].\Pi, Vp[p].\Pi)$
8. $V_s[s].N = \min(V_s[s].N, V_p[p].N)$
9. $Count[s] + +$
10. **for** each s
11. **if** $Count[s] = preds\_per\_sub[s]$
12. **then** $SatS = SatS \cup \{s\}$
13. return $SatS$

Improved Predicate Matching Optimization: The previous algorithm evaluates all predicates related to one attribute that is referenced by a given publication (i.e., iterated over each of its attributes). More specifically, at least one comparison between the two functions μ and π was required for each predicate to determine whether a match occurred. To minimize the number of comparisons, we improve our algorithm by sorting the predicates of the same attribute so that the predicate matching algorithm can stop earlier rather than evaluate all predicates.

In the representation of a predicate membership function $\mu_i = (\underline{m_i}, \overline{m_i}, \alpha_i, \beta_i)$, let $m_{i_1} = \underline{m_i} - \alpha_i$, $m_{i_2} = \underline{m_i}$, $m_{i_3} = \overline{m_i}$, $m_{i_4} = \overline{m_i} + \beta_i$. These are four critical points because they differentiate the boundaries where the membership degree has value 0 and where has value 1. Obviously, we have $m_{i_1} \leq m_{i_2} \leq m_{i_3} \leq m_{i_4}$. Similarly, for an event possibility distribution function $\pi = (\underline{n}, \overline{n}, \gamma, \delta)$, let $n_1 = \underline{n} - \gamma$, $n_2 = \underline{n}$, $n_3 = \overline{n}$, $n_4 = \overline{n} + \delta$, and we have $n_1 \leq n_2 \leq n_3 \leq n_4$.

The location of the mu_i functions and the π function determines whether a match occurs or not. Figure 4.7 shows examples of the matched predicates and no-matched predicates for a fixed publication. Suppose each predicate is represented as a membership function $\mu_i = (\underline{m_i}, \overline{m_i}, \alpha_i, \beta_i)$ and the publication is represented by a possibility distribution function $\pi = (\underline{n}, \overline{n}, \gamma, \delta)$. A match is established once the predicate "touches" the publication, i.e., μ_i and π intersect (e.g., μ_2 and μ_3 in Figure 4.7). Otherwise there is no

match. Concretely speaking, the predicate wont match the publication if its right-hand spread is to the left of the attribute function π (e.g., in Figure 4.7, $m_{1_4} \leq n_1$); or the predicate's left-hand spread is to the right of π (e.g., μ_4 in Figure 4.7, $m_{4_1} \geq n_4$).

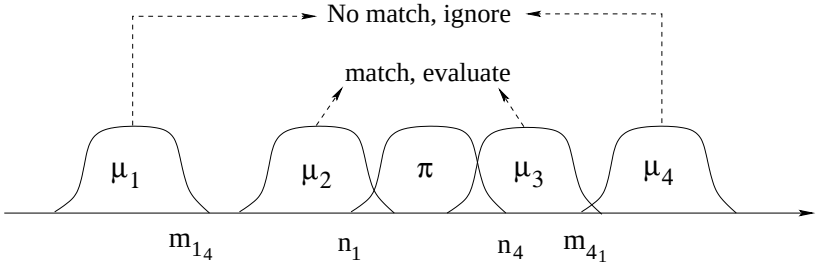


Figure 4.7: Examples of match and no-match between μ and π

Based on the above observation, predicates with the same attribute name, are organized in the order of their μ functions from smallest to largest starting from m_1 to m_4. For example, there are two predicates, p_i and p_j, that are under the same attribute index. We first compare m_{i_1} and m_{j_1}. The predicate with the smaller m_1 value is placed ahead of the other. If $m_{i_1} = m_{j_1}$ then we compare m_{i_2} with m_{j_2} and take the one with a lower value and place it ahead of the other. If the second points are equal then the same comparison is done for the third and forth points. If all the parameters are the same, then the predicate who enters the system earlier is placed ahead of the other. Figure 4.8 shows an example of four predicates that are ordered according to their locations order.

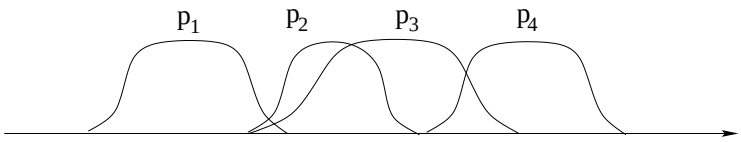


Figure 4.8: Examples of ordered predicates $p_1 < p_2 < p_3 < p_4$

For each attribute a_i of a publication, we pass the predicates whose membership functions are to the left of the π_i, and only evaluate predicates that intersect with the attribute. Predicate matching stops as soon as the above rules establish further none-matches.

In the possibility computation, the improved algorithm first compares n_1 (the first point of π) with the m_4 (the last point of function μ) of the predicates through the ordered predicate list until it reaches the predicate whose μ_4 is larger then n_1. Before that, all predicates are to the left of the π (as the left case in Figure 4.7), hence impossible to match. After $m_4 > n_1$ then we check m_1. If $m_1 > n_4$, then we can stop because from now on all predicates afterwards are to the right of π, thus impossible to match either (as the right case in Figure 4.7). We just need to evaluate the predicates whose $m_4 < n_1$ and $m_1 > n_4$. Algorithm *Improved Possibility Computation* shows the detailed possibility computation.

Algorithm *Improved Possibility Computation*($\sup\min(\mu, \pi)$

1. $j = 1$
2. **while** $m_{1j} < n_4$
3. **do while** $m_{4j} \leq n_1$
4. **do** j++
5. **if** $m_{3j} \leq n_2$
6. **then** find c such that
7. $R_m(\frac{c-\overline{m}}{\beta}) = L_n(\frac{\underline{n}-c}{\gamma})$
8. $\Pi = R_m(\frac{c-\overline{m}}{\beta})$
9. **else if** $m_{2j} \leq n_3$
10. **then** $\Pi = 1$
11. **else** find c such that
12. $R_m(\frac{c-\overline{n}}{\delta}) = L_n(\frac{\underline{m}-c}{\alpha})$
13. $\Pi = R_m(\frac{c-\overline{n}}{\delta})$
14. j++

In the necessity evaluation, the algorithm first compares n_3 (the third point of function π) with m_4 (the last point of function μ) through the ordered predicate list until it reaches the predicate whose μ_4 is larger than n_3. Before that, the complements of predicate functions μ are always intersected with the core of the π, so the necessity must be 0. After $m_4 > n_3$ we compute the necessity of each predicate until $m_1 \leq n_2$ because from

now on all necessities afterwards must be 0. Algorithm *Improved Necessity Computation* shows the detailed possibility computation.

Algorithm *Improved Necessity Computation*($\inf \max(\mu, \pi)$

1. $j = 1$
2. **while** $m_{1j} < n_2$
3. **do while** $m_{4j} \leq n_3$
4. **do** j++
5. **if** $m_{2j} < n_1$ **and** $m_{3j} > n_4$
6. **then** N=1
7. **else** find c_1 and c_2 such that
8. $R_m(\frac{c_1 - \overline{m}}{\beta}) = 1 - R_n(\frac{c_1 - n}{\delta})[= N_1]$
9. $L_m(\frac{m - c_2}{\alpha}) = 1 - L_n(\frac{n - c_2}{\gamma})[= N_2]$
10. $N = min(N_1, N_2)$
11. j++

Precision-space trade off: The approximate matching scheme trades off the processing of uncertain and vague information against precision. This suggest that a degree of match that is computed for a subscription must not be highly accurate, i.e., accurate to the n-th digit after the comma, as it is based on uncertainty anyway. We use this as motivation to experiment with different encodings for the degrees of match in our algorithm. The objective is to save space, while not sacrificing computational accuracy in our approximate matching model. We use three encodings: Float, one-byte, representing ten values, and one-byte representing 256 possible values for the degree of match. This is a straight forward encoding, with more refined schemes deferred to future work. The effects of different encodings will be evaluated in the experiments chapter.

Complexity Analysis: The space cost includes mainly the following parts: predicate hash table, predicate vector, subscription vector, and the association list for each predicate.

$$Space = \sum (Space_p * N_p) + 2N_p + 2N_s + N_p * N_{s_p}$$

Where $Space_p$ is the space for one approximate predicate function $\mu = [\underline{m}, \bar{m}, \alpha, \beta]_{LR}$, N_p is the number of predicates, $Space_p * N_p$ is the space to store all distinct predicates in the system, and N_s is the number of subscriptions. Each predicate and subscription is associated with two measures: possibility and necessity. Their types depend on the encoding chosen (float or char). The space cost for approximate matching is greater than crisp matching in which just one bit is used to record whether a predicate is matched or not matched. Using N_{s_p} as the average number of subscriptions associated with each predicate, the space of association lists takes $N_p * N_{s_p}$. Overall, the space cost is linear with the number of predicates and subscriptions: $Space = O(N_p + N_s)$.

The algorithm consists of two steps. First, predicate matching, consists of the time to retrieve the attribute from the index, which is just one lookup (hash table). Then all predicates under the same attribute are evaluated. In the original algorithm, all predicates membership functions under the same attribute need to be computed to get the possibility and necessity matching against the publication possibility distribution function. Assume that the time spending to evaluate each predicate membership function associate with the attribute is t_1, and all predicates are distributed uniformly on each attribute. Then the matching time for predicate matching is

$$Time(\text{regular predicate matching}) = t_1 * \frac{N_p}{N_a} * N_{a_e}$$

where N_p is the total number of all predicates, N_a is the total number of all attributes, hence $\frac{N_p}{N_a}$ is the number of predicates associated with one attribute. N_{a_e} is the average attributes number in the event.

In the improved algorithm, we don't need to evaluate all predicates associated with one attribute because of the good organization of the predicates. Evaluation stops at the point where all other predicates won't match for sure. We define $\alpha(\in [0, 1])$ as the coefficient between the number of predicates evaluated in the improved algorithm and in

the original one. This gives us a matching time of:

$$Time(\text{improved predicate matching}) = \alpha * t_1 * \frac{N_p}{N_a} * N_{a_e}$$

In the subscription evaluation, we suppose the time for one lookup is t_2. In our algorithm, for each matched predicate, we need to look up at the predicate subscription association matrix to find out which subscription contains this predicate, hence the time is $t_2 * N_{s_p} * N_{p_{sat}}$ where $N_{p_{sat}}$ is the average number of matched predicates. Since the thresholds θ_Π and θ_N are used to trim off the predicates whose matching degrees are not big enough to satisfy users. We denote β as the coefficient between the number of evaluated predicates and the number of the matched predicates whose matching degrees are beyond the thresholds, then we get $N_{p_{sat}} = \beta * \frac{N_p}{N_a} * N_{a_e}$. The subscription evaluation time is

$$Time(\text{subscription evaluation}) = \beta * t_2 * N_s * \frac{N_p}{N_a} * N_{a_e}.$$

In all, the matching time cost of the sequencing algorithm is

$$Time = \alpha * t_1 * \frac{N_p}{N_a} * N_{a_e} + \beta * t_2 * N_s * \frac{N_p}{N_a} * N_{a_e}.$$

4.3.3 Approximate Covering Algorithm

The covering algorithm is performed for each newly arriving subscription, which guarantees that there is no cover relation among subscriptions in the routing table. Therefore, if the new subscription is covered by any existing subscription, no other subscription in the routing table can be covered by the new subscription. On the other hand, if the new subscription covers an existing subscription, it won't be covered by any other subscription in the routing table. Based on these observations, we propose an algorithm that scans the routing table once, determines all possible cover relations and updates the routing table.

Algorithm *coverChecking* is the procedure for checking cover relations when a new subscription S arrives. There are two stages in this algorithm. First covered and covering

predicates are found and, second, covered and covering subscriptions are identified. This is a similar break-down as applied in the matching algorithm.

Algorithm *coverChecking*(S)

Input: an incoming subscription S

Output: boolean *covered*; a set R of subscriptions covered by S

1. $SubSet = \emptyset$
2. **for** each predicate $p(\mu)$ in S
3. **for** each predicate $p_i(\mu_i)$ where $p.attr == p_i.attr$
4. check predicate cover relation and set $p_i.coverflag$
5. **for** each p where $p.coverflag \neq 0$
6. **for** each S_i in $p.subs$
7. $S_i.count++$
8. set $S_i.coverflag$ according to $p_i.coverflag$
9. $SubSet = SubSet + S_i$
10. $R = \emptyset, covered = false, covering = false$
11. **for** each subscription $S_i \in SubSet$
12. **if** $((|S_i.preds| \geq |S.preds|)$ **and** $(S_i.Count \geq |S.preds|)$ **and** $(S_i.coverflag ==$ "$covering''))$
13. $R = R \cup S_i$
14. $covering = true$;
15. **if** $(|S.preds| \geq |S_i.preds|)$ **and** $(S_i.Count \geq |S_i.preds|)$ **and** $((S_i.coverflag ==$ "$covered''))$
16. $covered = true$
17. return
18. return R and *covered*

Analysis: In the first stage, for each predicate p in arriving S, we use its attribute as the hash key to get a set of predicates and check their cover relations by comparing their membership functions and p's membership function. Only those subscriptions who overlapped with S will go to the second step to check the subscription covering relation. A covering or covered subscription is detected when all their predicates are set by the same coverflag.

To check predicate cover relations, all predicates with the same attribute as the com-

ing predicate are evaluated. Assume all predicates are distributed uniformly on each attribute. The time for predicate checking depends on the number of predicates in the incoming subscription and the average number of predicates for each attribute, which is

$$Time(\text{predicate coverChecking}) = O(\frac{N_p}{N_a} * |S.preds|)$$

where N_p is the total number of all predicates, N_a is the total number of all attributes, hence $\frac{N_p}{N_a}$ is the average number of predicates associated with one attribute. $|S.preds|$ is the number of predicates in the incoming subscription.

In the subscription cover evaluation, for each subscription which contains at least one covered/covering predicate, we check the coverflag for all of its predicates. The time is $O(|SubSet| * |S.preds|)$ which depends on the number of subscriptions in $SubSet$. Thus, the total time for the *coverChecking* algorithm is

$$Time = O(\frac{N_p}{N_a} * N_{s_p}) + O(|SubSet| * |S.preds|).$$

Since the total number of predicates is approximately linear with the number of total subscriptions and $|SubSet| \simeq ratio_{overlap} * number\_of\_subscriptions$. Therefore, the overall time to check covering is linear with the total number of subscriptions and the overlap degree:

$$Time = O(ratio_{overlap} * number\_of\_subscriptions).$$

4.3.4 Approximate Merging Algorithm

The merging-based routing is an extension of covering-based routing to reduce the routing table size and network traffic overhead further. Two subscriptions that are largely overlapping each other can be merged into a more general subscription and be forwarded into the network.

When we merge two subscriptions, we remove the predicates with different attributes, and only keep the predicates with the same attributes and merge them to get the merged subscription. Algorithm *mergeSub* describes the details of the merging procedure.

Algorithm *mergeSub*(S_1, S_2)

Input: two overlapped subscriptions S_1, S_2

Output: a merged subscription S

1. $i = 1$
2. **for** each predicate $p1(\mu)$ in S_1
3. pick a predicate $p_2(\mu_i)$ where $p_1.attr == p_2.attr$
4. **while** $(S_2 \notin p_2.subList)$
5. pick another predicate $p_2(\mu_i)$ where $p_1.attr == p_2.attr$
6. **if** $(S_2 \neq NULL)$
7. $S.pred[i++] = mergePredicate(p_1, p_2)$
8. return S

To minimize the false positives introduced by the merger, the subscriptions to be merged should overlap maximally, in the other words, have the largest possible similarity. Since subscription is a list of predicates, we define the similarity between two subscriptions as the minimum of the similarities between each pair of predicates with the same attribute. The similarity between two predicates is defined based on the overlapping area and their matching degree. Take the predicates in Figure 4.5(a) as an example. The overlap ratio between two predicates, $R_{overlap}$, is the ratio of the shaded area and the area under their merger μ_m. Similar to the matching between subscription and publication, the intersection, (Π, N), is a fuzzy measure for their overlapping ratio. We use a triple $(R_{overlap}, \Pi, N)$ to describe the similarity between two predicates which means they are overlapping each other by $R_{overlap}$ ratio with (Π, N) possibility. When the merging operation is performed, we only select the subscriptions whose similarity is larger than a threshold $(T_{R_{overlap}}, T_\Pi, T_N)$. The threshold of (T_Π, T_N) is for merging, and it is different from (θ_Π, θ_N) which is for matching.

Algorithm *predChecking*(P, L)

Input: an incoming predicate p, a list L which contain all the predicates with the same attribute as p

($*$ compute the overlap between μ and μ_i and set the value, $p_i.coverflag$ stores the type of covering $*$)

($*$ 1: μ_i covers μ, 2: μ covers by μ_i, 3: $\mu = \mu_i$, (0,1): the ratio of overlap between μ and μ_i $*$)

1. **for** each predicate $p_i(\mu_i)$ in L
2. **if** $((\mu == \mu_i)$ **and** $(\theta_\Pi == \theta_{\Pi_i})$ **and** $(\theta_N == \theta_{N_i}))$
3. $p_i.coverflag = 3$
4. **else if** $((\mu$ covers $\mu_i)$ **and** $(\theta_\Pi \leq \theta_{\Pi_i})$ **and** $(\theta_N \leq \theta_{N_i}))$
5. $p_i.coverflag = 2$
6. **else if** $((\mu_i$ covers $\mu)$ **and** $(\theta_\Pi \geq \theta_{\Pi_i})$ **and** $(\theta_N \geq \theta_{N_i}))$
7. $p_i.coverflag = 1$
8. **else** $p_i.coverflag = compute_{overlap}(\mu, \mu_i)$
9. $p_i.Pos = \sup_x \min(\mu(x), \mu_i(x))$
10. $p_i.Nec = \inf_x \max(\mu(x), 1 - \pi(x))$
11. return L

We can calculate the similarity between the new subscription and each subscription along with Algorithm *coverChecking* when the new subscription enters the system. This calculation doesn't need extra time and extra space. The similarity can be computed at the same time when we check the cover relations and are stored in the space for matching. The subscription who has the largest similarity with the incoming subscription will be remembered. If there is no cover relation between incoming subscription and existing subscriptions and the largest similarity is larger than the threshold we set, we will perform the merge operation. This approach is better than offline merging since we don't need extra time to compute the similarity between each pair of subscriptions. Furthermore, the routing table can be maintained as small as possible. The threshold for merging can be used to tune the tradeoff between routing table size and false positives. The combined algorithm to check predicate cover relation and calculate the similarity between predicates is depicted in Algorithm *predChecking*.

Analysis: The procedure to calculate the similarity between subscriptions and determine what to merge is performed simultaneously to checking cover relations. As for the *mergeSub* procedure, it depends on the number of predicates in one subscription and the length of the predicate list which shares the same attribute since we need to go through the list to check whether another subscription contains a predicate who has

the same attribute. In the worst case we need to look through the whole list for each predicate. Thus the complexity to merge two subscriptions is

$$O(|S.preds| * \frac{N_p}{N_a})$$

where N_p is the total number of all predicates, N_a is the total number of all attributes, hence $\frac{N_p}{N_a}$ is the average number of predicates associated with one attribute.

4.3.5 Intersecting-based Routing

In content-based routing, subscriptions are forwarded toward reverse path of intersecting advertisement. Thus, we need to find intersecting subscriptions when receiving a new advertisement and find intersecting advertisements when receiving a new subscription. Since advertisement share the same representation of subscription, these two algorithms are similar. Algorithm *getIntersectingSubs* describes the procedure to get a set of intersecting subscriptions upon receiving a new advertisement.

Algorithm *getIntersectingSubs*(A)

1. $SubSet = \emptyset$
2. **for** each predicate $p(\mu) \in A$
3. **for** each predicate $p_i(\mu_i)$ where $p_i.attr = p.attr$
4. **if** $(compute\_overlap(\mu, \mu_i) > 0)$
5. $SubSet = SubSet \cup p_i.subSet$
6. return $SubSet$

4.4 Parameterizations Selection for A-ToPSS Model

The A-ToPSS model offers its users great flexibility and leaves room for tuning a wide range of default parameters. It is often a challenge to select the right membership function parameterizations, the exact number of membership functions to represent one dimension, the appropriate aggregation function or the right thresholds. Users may like to define

their own specifications according to the information of other people's requirements. It will be more convenient if we could provide them with the aggregated knowledge of current data in the system. For example, a user wants to get to know the average price for a second-hand car and buy one with a price relative to the notion of cheap in the system. Therefore, his specification for the price may vary according to the average price. If the average price is 10K dollars, he will define his expectation between 8K to 11K. However, if the average price is 5K, his expected price will decrease to [4k; 6k].

However, A-ToPSS is used in a context where many subscribers (potentially millions) seek the right information. Consequently, much information about what defines certain concepts in specific domains is readily available, such as an "average understanding" of what constitutes a "cheap" price of a popular electronics gadget available in an online auction. If this information could be exploited, better default parameter choices could be determined for subscribers and publishers of such a system.

Therefore, we propose a density estimation approach to determine default value settings for membership functions based on historical data (i.e., from past subscriptions and publications). We demonstrate our approach on real data traces that we have collected from an online auction site. The real data traces serve as models of subscriptions and publications that have been submitted in the past (and will be made available in the future). In the density estimation we do not differentiate between subscriptions or publications, but mine for default parameterizations of the membership functions underlying both entities. This is possible due to the link of a fuzzy set and a possibility distribution, explained in further detail in [59]. The mined parameterizations are then used to provide default values for imperfect concepts.

The feasibility of membership function mining depends on the distribution of the data in data set. Since we don't know what distribution the data displays beforehand, we can't predefine any function to represent the distribution of the data. Therefore, we propose a function mining approach based on the density estimation of the data.

The function mining is performed for each attribute, thus it is based on one dimensional data. The x-axis values are the distribution of the data for one attribute and the goal is to get y-axis value for each x value to represent the degree of its membership. We assume that the number of concepts to describe the characteristics of one dimension is given. For example, the dimension price, could be represented by three concepts "cheap", "fair" and "expensive". Each of these concepts is represented by a parameterized membership function, which can be adapted to a specific understanding by modifying its parameters. We focus on estimating the representation of each parameterized membership function.

There are two steps to estimate the representation of a membership function. First, given k, the number of membership functions of one attribute, find the domain for each membership functions. In other words, given a list of data x_i, how to partition the list into k clusters so that the distance between clusters are maximized and the inner distance inside one cluster are minimized. Since the data in our case is one dimension, we compute the distances between each pair of neighbors, choose the $k-1$ largest distances as the separators to partition the list (as shown in Figure 4.9). Next, the exact expression of the function is estimated for each membership function based on the data density. We use a similar density estimation as in probability and statistics. The basic idea is to count the number of data points within a small region around each x-value and then divide it by the length of the region to get the density. To make sure no y-value larger than 1, we normalize the $f(x)$ by dividing the maximum $f(x)$. The detailed algorithm is described in Algorithm *DensityEstimation*.

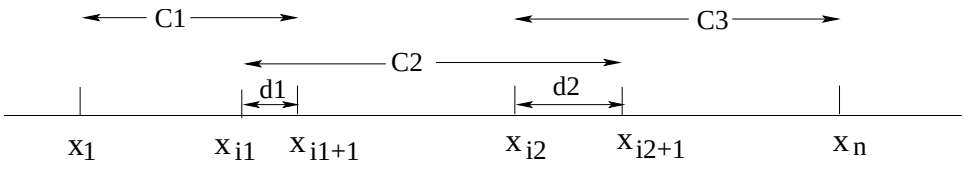


Figure 4.9: partition the list into k clusters

Algorithm *DensityEstimation*(V, k)

Input: a vector V of a list of data x_i, number of membership functions k

Output: $f_j(x), j = i, \cdots, k$ for each membership function

1. sort(V) so to get $x_1 \leq x_2 \leq \cdots x_n$
2. calculate the distance between adjacent data $d_i = |x_{i+1} - x_i|$
3. find the $k-1$ largest distance, $\{d_j | d_j = x_{i_j+1} - x_{i_j}, j = 1, \cdots, k-1\}$
4. separate the list into k clusters, each cluster j has the domain $C_j : [a_j, b_j]$ where $a_j = x_{i_j}$ and $b_j = x_{i_{j+1}+1}$
5. **for** each cluster $C_j : [a_j, b_j]$
6. find the maximum distance $d = max\{d_i | d_i = |x_{i+1} - x_i| \wedge x_i \in [a_j, b_j]\}$
7. **for** each variable x
8. $I_x = [max(a_j, x - d/2), min(b_j, x + d/2)]$
9. $A_x = x_i | x_i \in I_x$
10. $f_j(x) = |A_x|/d$
11. normalize: find the maximum $f_j(x_i)$ $M = max(f_j(x_i))$
12. $f_j(x) = |A_x|/(d * M)$

4.5 Experiments

4.5.1 Experiment Setup

In this section we evaluate our approach to model uncertainties in content-based routing. All experiments are run on a Linux system with 2GB RAM and a 2.4GHz microprocessor. First, we validate our density estimation algorithm to determine the membership function. Next we evaluate the performance of the approximate matching algorithms on a single router. Finally, we show the approximate content-based routing experiments on a network.

4.5.2 Model Parameterizations Mining

To validate our density estimation algorithm for membership function learning, we collect a set of real data from an online apartment renting web site. The information posted by publishers listed the type of apartment, rent range and other binary attributes of the apartment including air conditioning, pet allowed etc. We classify all the data into three categories according to apartment type: one-bedroom, two- bedroom and three-bedroom, eliminate binary attributes and keep only the qualitative rent value since the function mining algorithm can only be applied to qualitative data. In the collected data set, there are two types of values for rent price: discrete point value and interval value. The dataset size for these two types are summarized in 4.1. The dataset of two-bedroom price is the largest one, thus we use this dataset to demonstrate model parameterizations mining process.

Table 4.1: Data Classification and Dataset Size

Type	**Point Value**	**Interval Value**
one-bedroom	89	318
two-bedroom	180	1178
three-bedroom	87	592

Since the range of the interval is relative small compare to the value, we take the median as the representative and thus we get a set of discrete data points for rent attribute. The distribution of the set of data is shown in Figure 4.10. From the figure, we can see that most data drop in the range between $500 and $2000. There are very few users subscribe their interest above $3000.

To get a membership function to analytically represent each attribute of price, we applied the density esitimation algorithm to the price dataset.

Figure 4.11, Figure 4.12 and Figure 4.13 show the three membership functions mined

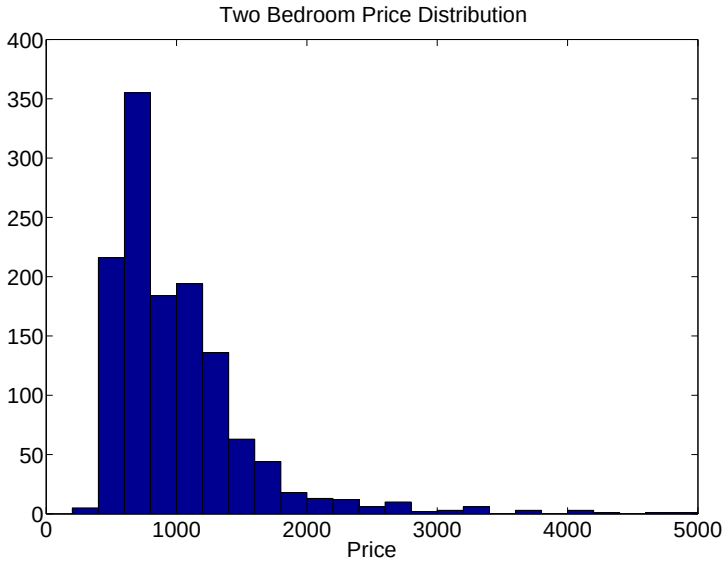


Figure 4.10: Two Bedroom Price Distribution

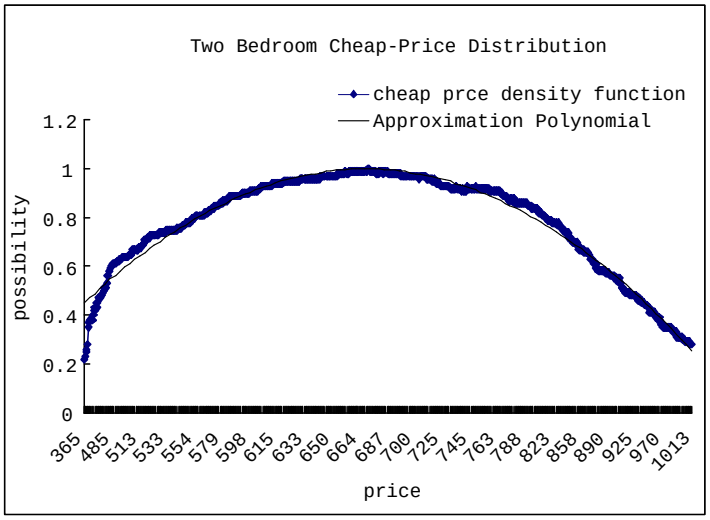


Figure 4.11: The function representing the cheap rent for two-bedrooms apartment

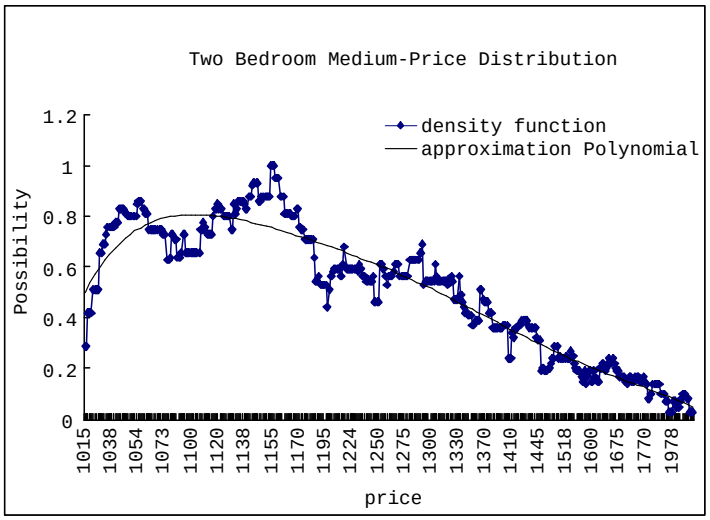


Figure 4.12: The function representing the medium rent for two-bedrooms apartment

for the rent of two-bedroom apartments. We partition the rent into three clusters: cheap, medium and expensive. After applying Algorithm *DensityEstimation*, we smooth the functions using polynomial approximations. The figures show that the density functions for all three clusters basically follow a bell-shaped function distribution, which can be used in our approximate matching semantics, although it is not very smooth for the second cluster and has some fluctuations.

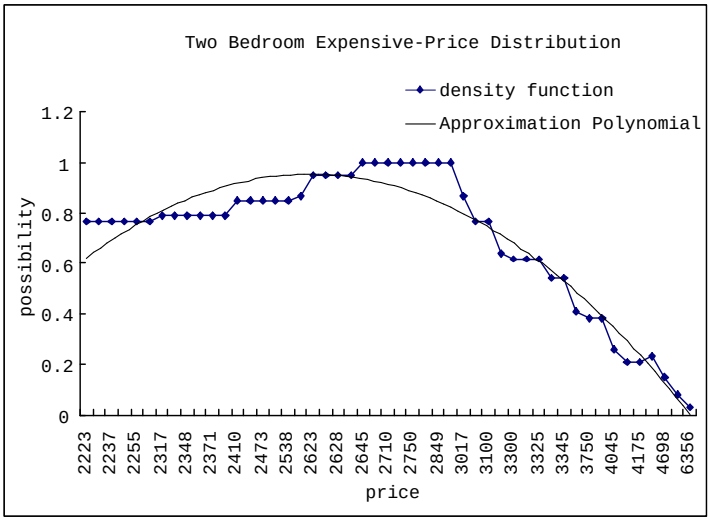


Figure 4.13: The function representing the expensive rent for two-bedrooms apartment

4.5.3 Evaluation of Approximate Matching on a Single Router

In this section, we evaluate the performance of the approximate matching algorithms with respect to time and memory. The objective is to confirm the efficiency of the algorithms and compare the crisp publish/subscribe model with the approximate model. We also examine the trade off between matching precision against the space used for storage.

Experiment Framework

We are using a synthetic workload so that we can independently examine various aspects of the approach. The algorithm is implemented in C. To render the approximate and crisp cases comparable, we generated crisp subscriptions and publications based on approximate ones. For subscriptions, we define three interval types of crisp predicates derived from the approximate ones: *optimistic, pessimistic* and *middle*. There are three ways to determine the lower bound and upper bound of the interval. If m_1, m_2, m_3, m_4 are the four parameters for the representation of the approximate predicate then those three interval types are defined in Figure 4.14.

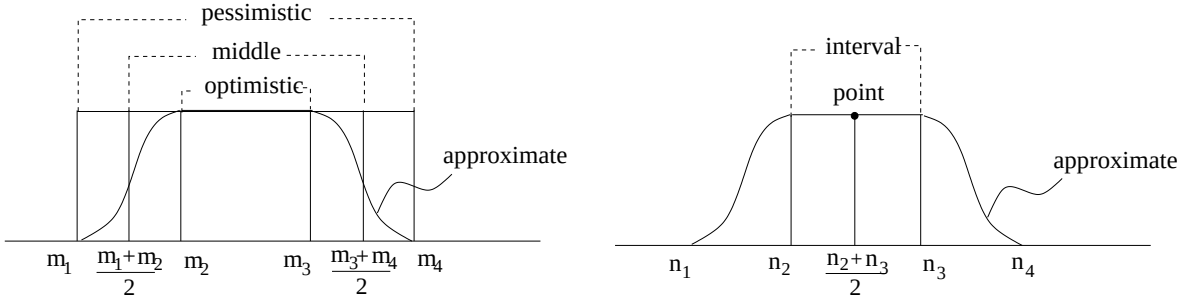


Figure 4.14: Definition of different subscription and publication types

Publications are generated similarly (cf. Figure 4.14). We have two choices when generating crisp publications on the basis of approximate publications: *point* and *interval*. They refer to the types of the value for each attribute in the publication. *Point* type is defined to be consistent with the publication language data model in crisp publish/subscribe system so that they are comparable. *Interval* type serves to compare the

difference between an interval representation and a fuzzy set representation for an uncertain constraint. Since we define three choices to generate interval subscriptions, we can compare the effects of different lower bound and upper bound of the interval in the subscriptions. Therefore, we only generate one interval type for publications.

Table 4.2: A-ToPSS Workload parameters

parameters	**value**	**description**
n_t	42	size of predefined attributes
n_v	[2,5]	number of predefined fuzzy concepts for each attribute
$Size_P$	4	number of attributes in publication
$Size_S$	2	number of predicates in subscription
N_{sub}	[100, 100,000]	number of subscriptions
N_{pub}	10	number of publications

The workload parameters used in the experiments are shown in Table 4.2. A subscription is a list of predicates. Predicates are determined by an attribute name and a fuzzy set. Predicate attribute names are drawn from a predefined set of names. The same set is used to generate publications. The total number of names available is determined by n_t. For each attribute name, we provided a set of fuzzy concepts (the number of those concepts is n_v) to be drawn as predicate's lingual fuzzy value. In this experiment we use a trapezoidal fuzzy set membership function. The function is defined by 4 points, randomly selected from the domain, governed by a uniform distribution, to form the membership function representation. The membership functions of covered subscriptions are generated based on covering subscription's membership functions, where the 4 points of the covered function are selected from within the interval of the covering function.

Performance Evaluation

To evaluate the performance, we classify the implementations into 3 pairs according to different emphasis: 1. algorithms: regular matching vs. improved matching algorithm; 2. matching result representation: *float-wise* (4 bytes) vs. *bit-wise* (8 bits or 4 bits); 3. the data structure for the association between predicates and subscriptions: matrix-based vs. list-based. We consider the following metrics: subscription loading time, matching time and memory used. The matching time measurement starts just before the publication has been submitted to the system and ends right after the system responds.

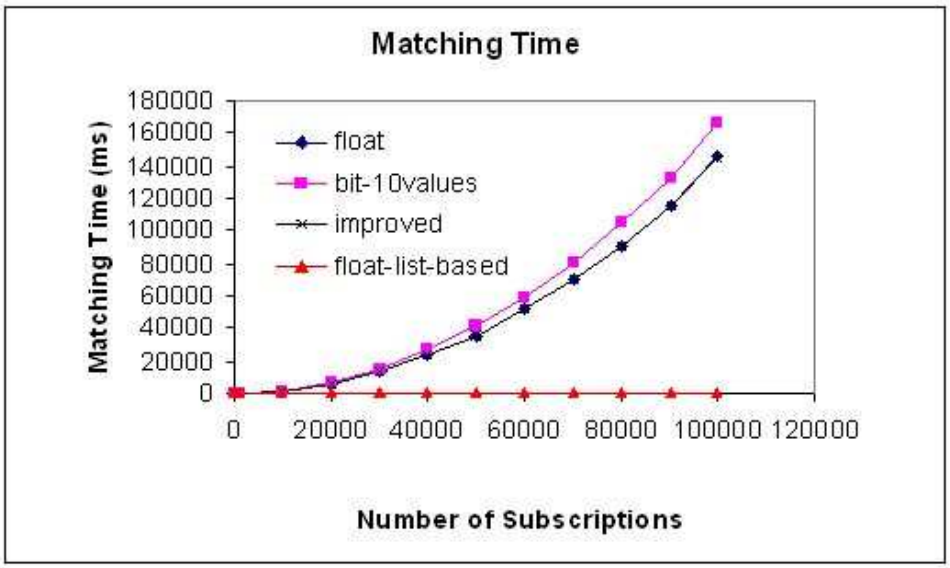


Figure 4.15: Matching Time

Figure 4.15 compares the matching time across all implementations. The matching time depends on the number of predicates associated with the same attribute and the number of subscriptions that include those matched predicates, hence matching time increases with increasing the number of subscriptions. In Figure 4.15 we compare the float-wise, bit-wise and improved matching implementations. The advantage of the improved predicate matching algorithm is not distinguished since the subscription evaluation step takes much more time than predicate matching. The bit-wise implementation runs slower

than the float-wise because it needs more computation to set the bit values. To show the benefits of the improved predicate matching algorithm, we ran the predicate matching process only and Figure 4.16 showed that the improved algorithm runs faster.

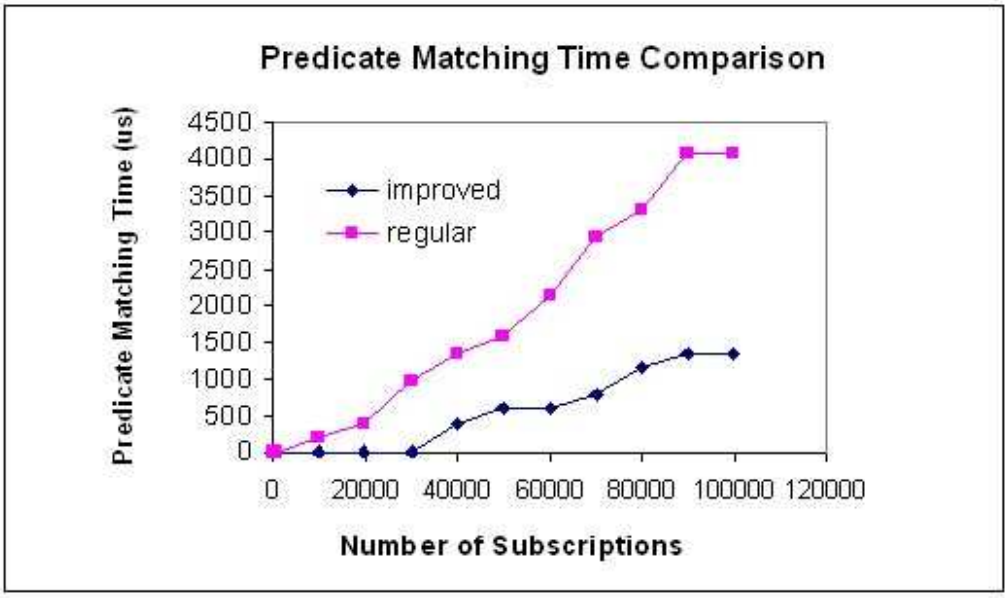


Figure 4.16: Predicate Matching Time

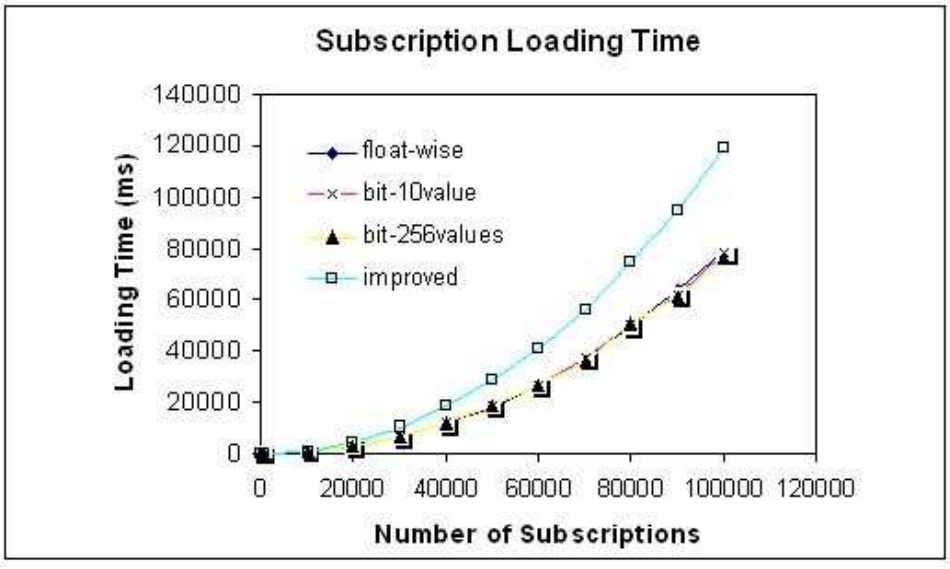


Figure 4.17: Loading Time

Figure 4.17 compares the loading time among different algorithms. Contrary to the

matching time, the improved algorithm needs more time than the other three. This is because predicates need to be inserted into a sorted list based on the 4 points of the function. This is a tradeoff between the loading time and matching time. In a real application, most subscriptions stay in the system for a long time and the matching time is more important to the user. With the high publication submission rate, it is better to process the matching quickly and respond as soon as possible.

Figure 4.18 shows memory utilization for the float-wise and bit-wise algorithms. The difference shows up only in the storage of matched result of predicates and subscriptions, so we only consider the space used here. We can see that bit-wise one uses less than the float version due to the space saved by using several bits instead of 4 bytes.

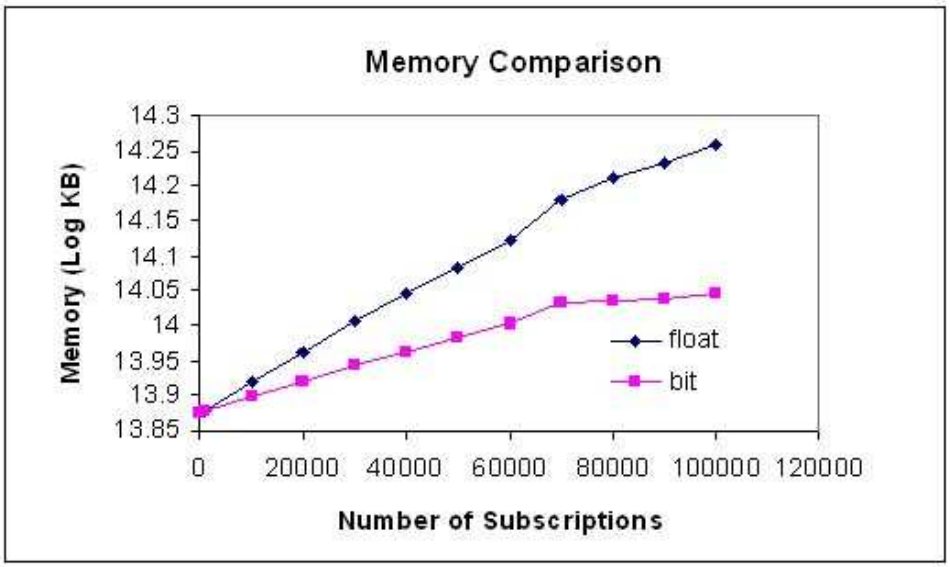


Figure 4.18: Memory Used

In our experiment, the workload is generated randomly, thus the number of subscriptions that contain the same predicate is very small compared to the total number of subscriptions. Therefore, both the matching time and memory using the list-based approach is much less than the matrix-based approach considering the size of the list for each predicate is much smaller. The results are shown in Figure 4.19 and 4.20. In the case where each predicate is contained in most subscriptions, the matrix-based version

should be much better because access to the table is faster.

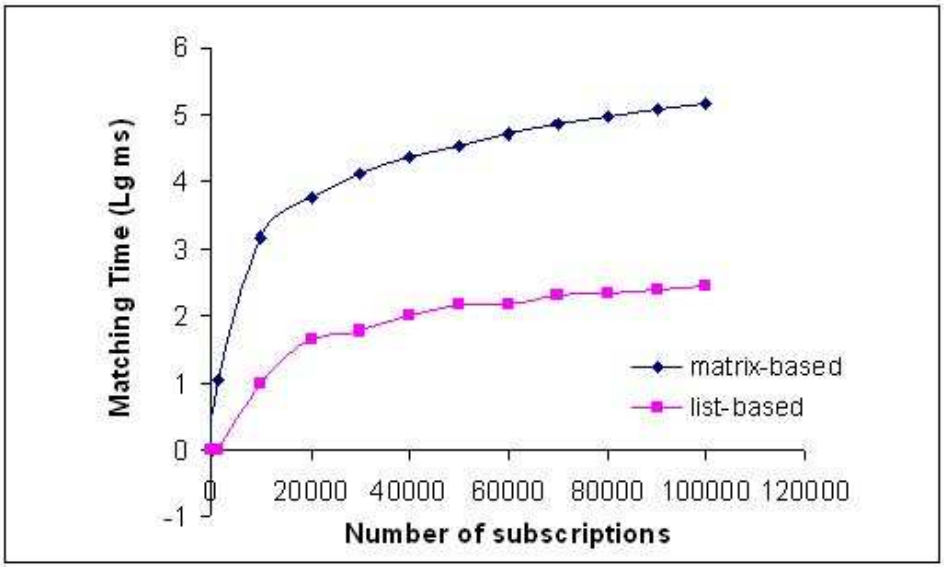


Figure 4.19: List-based Matching Time

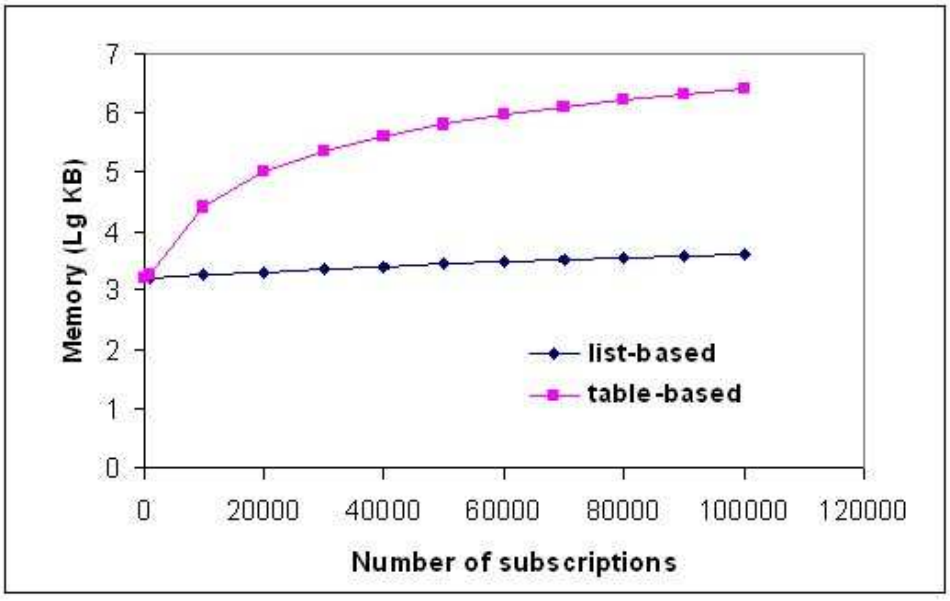


Figure 4.20: List-based Memory Used

The decrease in space using bits instead of float results in a loss of precision in the matched results. A performance measure *precision* is defined as:

$$precision = \frac{\sharp\text{Correct Subscriptions Returned}}{\sharp\text{Subscriptions Returned}}$$

In publish/subscribe systems, correctness means that the matched subscriptions the system returns are exactly what the users want. For example, a user wants to be notified when her subscription matches with a degree larger than 0.8. In the 10 value bit-wise implementation (0,1 and eight equal parts in between), those matched degrees between 0.75 and 0.8 are represented by the same bit pattern as those between 0.8 and 0.875. If users are only satisfied with the latter ones, there is an error since the system will return all subscriptions whose degrees are between 0.75 and 0.875. Compared to the float-wise implementation which always return the correct data, the bit-wise version will also return some subscriptions that are not satisfied because of the precision loss. In our context, the precision is computed by

$$precision = \frac{\sharp\text{Subscriptions float-wise Returned}}{\sharp\text{Subscriptions bit-wise Returned}}$$

Figure 4.21 shows the precision of 8-bit-wise and 4-bit-wise implementations. We can see that the precision of the 8 bits version is stable around 98% and the 4 bits is stable around 96%. Considering the acceptance of users' error range in the real world, the decrease of the bits don't introduce much error.

Comparison Between Crisp and Approximate Model

In this set of experiments we compare the crisp and approximate publish/subscribe matching model with respect to the number of matches identified under different conditions. Table 4.3 shows the different numbers of matches based on the evaluation of a fixed number of approximate publications over different kinds of subscriptions and different thresholds. We use α as the thresholds to assess a minimal possibility and necessity beyond which a subscription is not counted as a match (i.e., $\omega_\Pi = \omega_N = \alpha$). We can see that for one type of subscription, the number decreases with increasing α, which indicates the threshold effect of α. With the same α, the pessimistic case results in the largest number of matches and the optimistic case results in the fewest matches. The

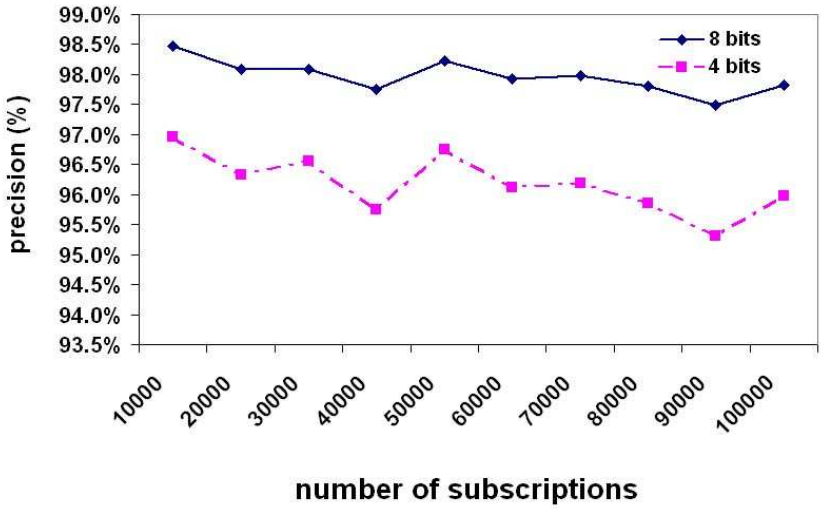


Figure 4.21: The trade off between precision and space

approximate case and the middle case do not exhibit much difference. This is due to the wider restriction of subscription, the greater the probability of being matched.

Table 4.3 then shows the numbers of matched subscriptions for different type of publication when the subscription type is fixed. When $\alpha = 0$, the approximate publication returns the most subscriptions and the point type returns the least. This is the same as for subscriptions. However, with the increase of α, the approximate publication matches a very small number of subscriptions, whereas point-valued publication matches the most. This is because α is used as the threshold for both possibilities and necessities. Think of the intuitive meaning of possibility and necessity we defined in the model section. For the approximate publication, the function restricting the attribute has a wider domain, thus it is more likely that the publication intersects with the complementary region of subscriptions, therefore the necessities is very likely to be 0, which makes it more difficult to reach the α threshold. For the point-valued publication, it is easy for such a value to be located in the core of the subscription function, thus more subscriptions are matched

Table 4.3: Comparison of number of matches for various types of subscriptions with approximate publications and number of matches for various types of publications with approximate subscriptions.

Subscription Type	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$
appro	4628	184	7
pessi	4628	804	281
middle	4438	184	39
optim	3763	47	7
Publication Type	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$
appro	4628	184	7
interval	3720	474	170
point	2960	1932	868

with high α for point-valued type than others.

Effect of Choice of Aggregation Functions

To compute the overall degree of match for each subscription, different operations can be chosen to aggregate the degrees of match of predicates (e.g., min, weighted average etc.). For example, when students are looking for housing close to campus, they will consider, both the price and the distance. One student may worry more about the price, another student may be more indifferent and be satisfied with a balanced average, while a third student maybe more location-sensitive. In the proposed approximate matching scheme one aggregation function evaluates the degree of match of all subscriptions in the system, which maybe influenced by different thresholds. However, it is also important to understand the effect different aggregation functions have on matching effectiveness. This effectiveness is evaluated through *precision* and *recall* metrics. Three popular aggregation

operations: *min*, *max* and *average* are compared. The definition of precision is given before, recall is defined as:

$$recall = \frac{\sharp\text{Correct Subscriptions Returned}}{\sharp\text{Correct Subscriptions}}.$$

The F-measure is a common metric for the evaluation of information systems. It relates precision and recall. It is computed as follows:

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

The relationship of the set of matching subscriptions using different aggregation operations is shown in Figure 4.22. The experiment runs by distributing user's aggregation expectation uniformly over 4 choices: min, max, average and weighted average (assign a weight to each predicate.) The correct data set should contain subscriptions whose overall degree, computed according to user's expectation, are larger than the threshold (ω_Π, ω_N). The data set returned contain subscriptions whose overall degree, computed by only one uniform function (either min, max or weighted average), is larger than the thresholds. Among the set we returned, there maybe some subscriptions whose overall degree is less than the threshold if computed according to the user's expectation, which is a positive error. Similarly, outside the data set we got, there maybe some subscriptions that are not returned to the user, but the overall degree is larger than the threshold, which is the negative error. Figure 4.22 shows the comparison of the F-measures on these operations. It can be observed that all operations have high F-measures (around 95%), while the result of the average aggregation performs best.

4.5.4 Evaluation of Approximate Routing on a Network

Experiment Framework

We run experiments to evaluate the effects of approximate covering and merging algorithms. To evaluate the effectiveness of the approximate covering and merging algorithms,

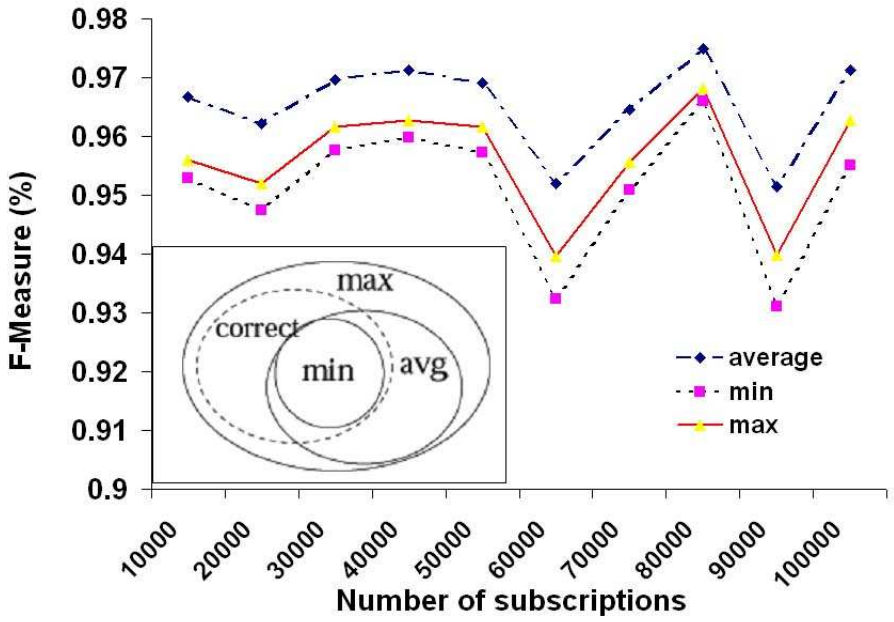


Figure 4.22: F-measure on aggregations

we measure subscription insertion time, matching time and routing table size. The routing table size is calculated based on the final number of subscriptions stored in the system. The insertion time is calculated as the average over the last 1000 subscriptions inserted. The matching time is computed as the average over the last 1000 publications matching operations. For merging, we also evaluate the number of false positives and false negatives introduced by the imperfect merger. The following factors influence the performance of the algorithms: number of subscriptions stored in the routing table, subscription covering ratio, and merging threshold. We examine the effect of these factors. For each experiment, we vary one parameter and fix the others to their default values as specified in Table 4.4.

Approximate Routing Performance

In following two figures, we evaluate the routing table size and the matching time, as the number of subscriptions increases. We compare the matching performance for the

Table 4.4: Workload parameters for experiments on a network

parameters	value	description
$Size_P$	10	number of attributes in publication
$Size_S$	4	number of predicates in subscription
N_{sub}	100,000	number of subscriptions
N_{pub}	20,000	number of publications
$R_{covering}$	0.5	subscription covering ratio
T_{merge}	$(0.5, 1, 0)$	threshold $(T_{R_{overlap}}, T_{\Pi}, T_N)$ for merging

algorithm without covering, with covering, and with merging. Figure 4.23 shows that the routing table size is reduced significantly by covering and reduced even more by merging. The results also show that the size of the routing table increases linearly with the increase in the number of subscriptions, The rate of the increase differs for the three algorithms. The covering algorithm decreases the speed with which the routing table increases in size. The merging algorithm decreases the speed even further.

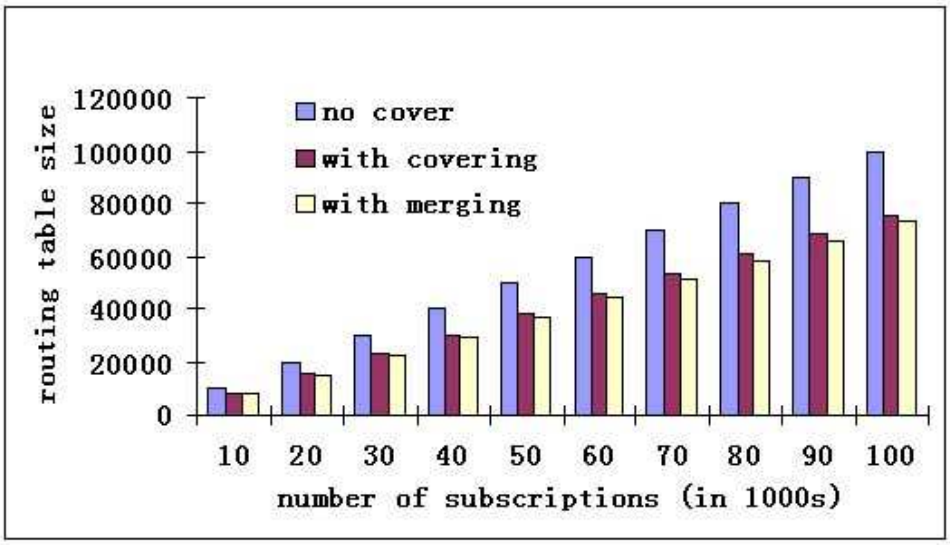


Figure 4.23: Routing Table Size vs. #subscriptions

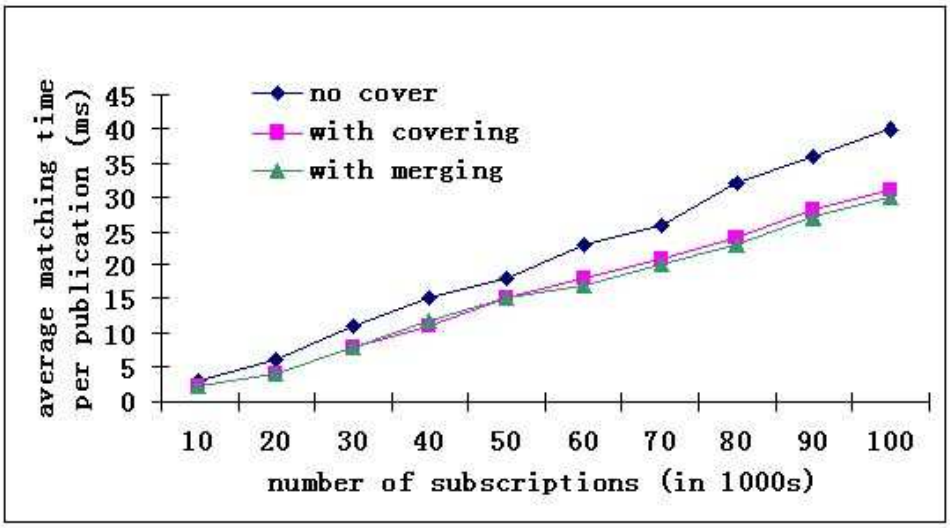


Figure 4.24: Matching time vs. #subscriptions

Figure 4.24 shows the matching time for increasing number of subscriptions. The matching time depends on the number of predicates with the same attribute as publication and the number of matched subscriptions. With the increase of the total number of subscriptions, the predicates under one attribute increases, and also the number of matched subscriptions. Thus, the matching time increases with the number of subscriptions. However, using the covering algorithm, some covered predicates and some matched subscriptions will be dropped since they are covered by others. Thus, there are fewer subscriptions in the routing table than in the original workload and it takes less time to match. Merging is similar, but the number of subscriptions is reduced even further and matching is even faster.

Figure 4.25 shows the routing table size when the subscription covering ratio varies. The workload contains 100,000 subscriptions. The larger the covering ratio in subscriptions, the more subscriptions are covered and hence dropped. Thus, the smaller the routing table and fewer subscriptions that need to be forwarded into the network. The effect is a reduction in matching time, as shown in Figure 4.26. The merging algorithm is a further optimization. However, the routing table size resulting from merging mainly de-

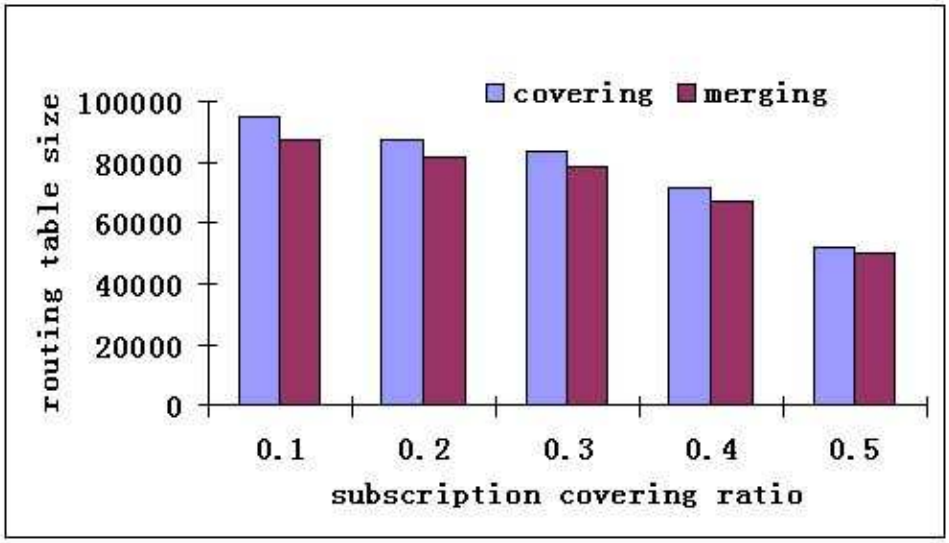


Figure 4.25: Routing table size vs. subscription covering ratio

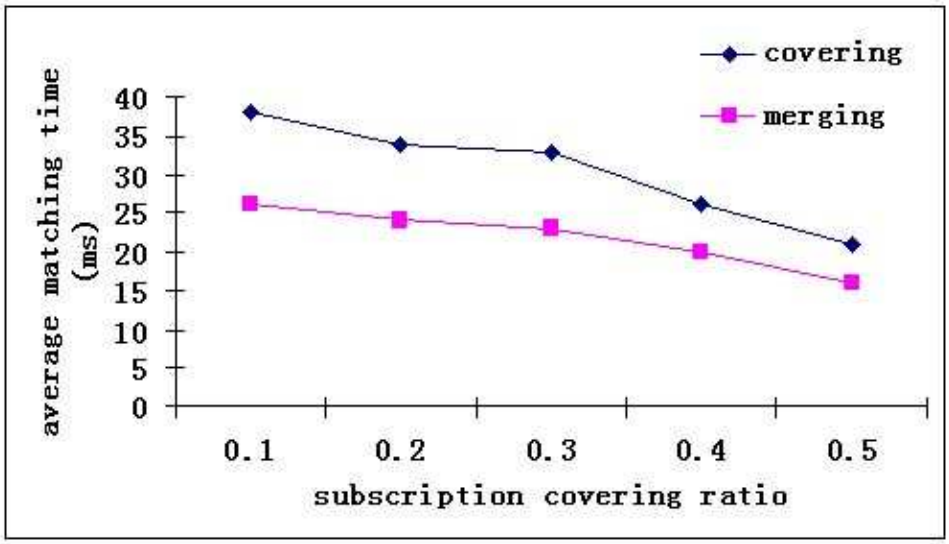


Figure 4.26: Matching time vs. subscription covering ratio

pends on the merging threshold set in the merging algorithm, not on the ratio of covering (this will be demonstrated later.)

Covering and merging can reduce the routing table size, but it requires more processing time to check the cover relation among all subscriptions and to perform merging. This is a trade off between matching time and subscription processing time. To evaluate

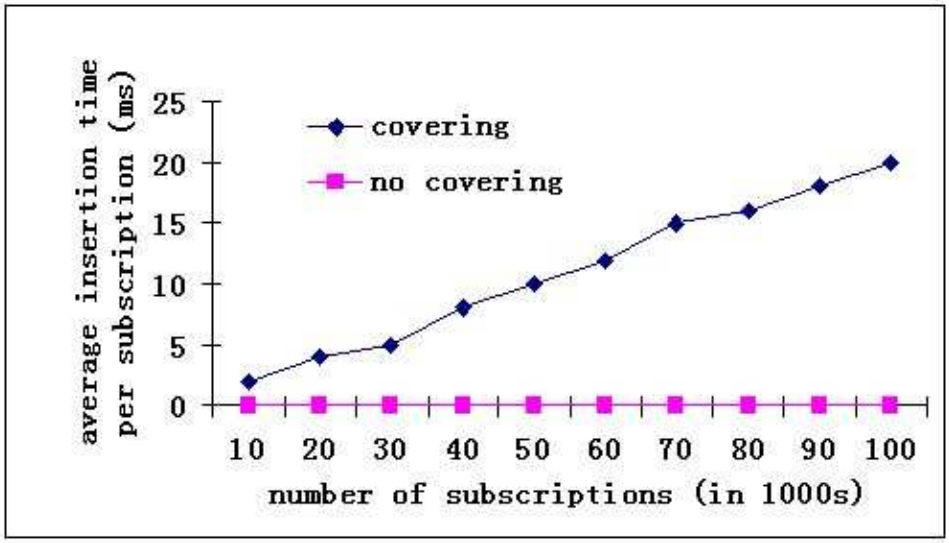


Figure 4.27: Insertion time vs. #subscriptions

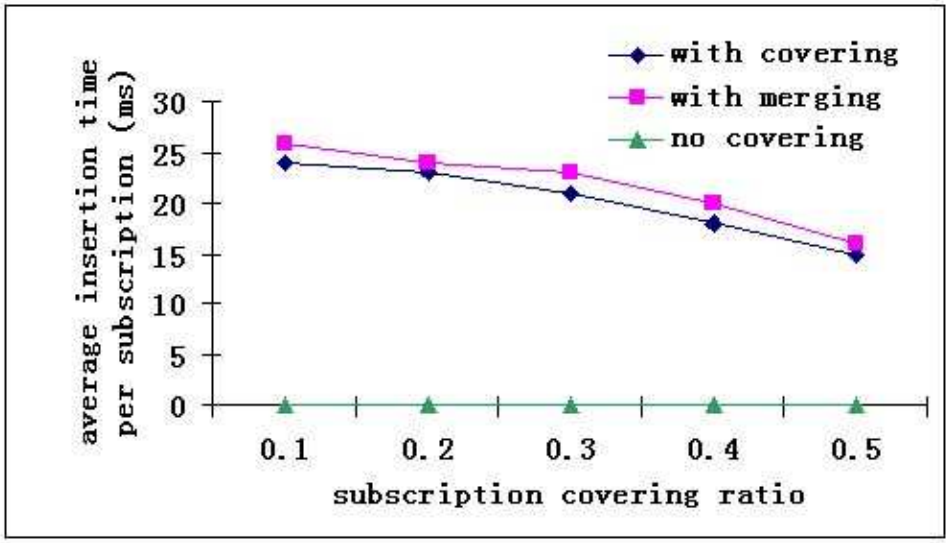


Figure 4.28: Insertion time vs. subscription covering ratio

this trade off, we also measure the average insertion time of one subscription for the covering and merging algorithm. We first fix the covering ratio and increase the number of subscriptions. The results are shown in Figure 4.27. Not surprisingly, the subscription insertion time with covering is larger than that without covering, due to the required covering computations. Without covering, the time is $O(1)$. With the covering algorithm,

the average insertion time for one subscription grows with the increase in the number of subscriptions, which validates the time complexity analysis of the covering algorithm for a constant covering ratio. The relation of the insertion time and the covering ratio is also shown in Figure 4.28 where the number of total subscriptions is fixed and the covering ratio increases, therefore the routing table size decreases and the insertion time decreases as well.

We also observe that the insertion time for the merging algorithm is similar to that of the covering algorithm. This is because our merging computation is done at the same time as the covering computation and does not require extra time. It only requires one more operation to merge subscriptions, given their similarity is larger than the threshold. However, this time may compensate for the decrease of routing table size. Therefore, the insertion time of the covering algorithm and the merging algorithm do not differ a lot.

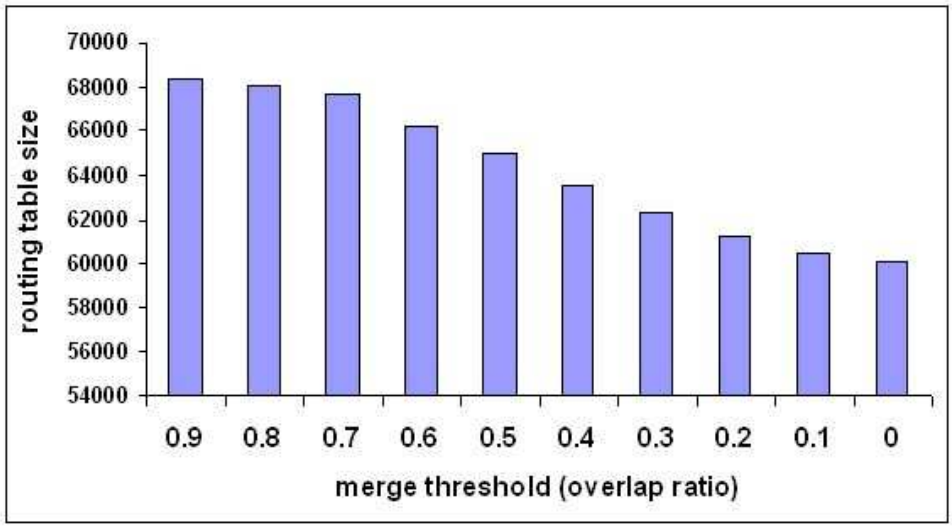


Figure 4.29: routing table size vs. merge percentage

The reduction of routing table size due to merging is determined by the merging threshold assigned in the algorithm. The results are shown in Figure 4.29. The smaller the threshold, the more subscriptions will be merged and the smaller the routing table. The extreme case is that each pair of subscriptions has no overlap.

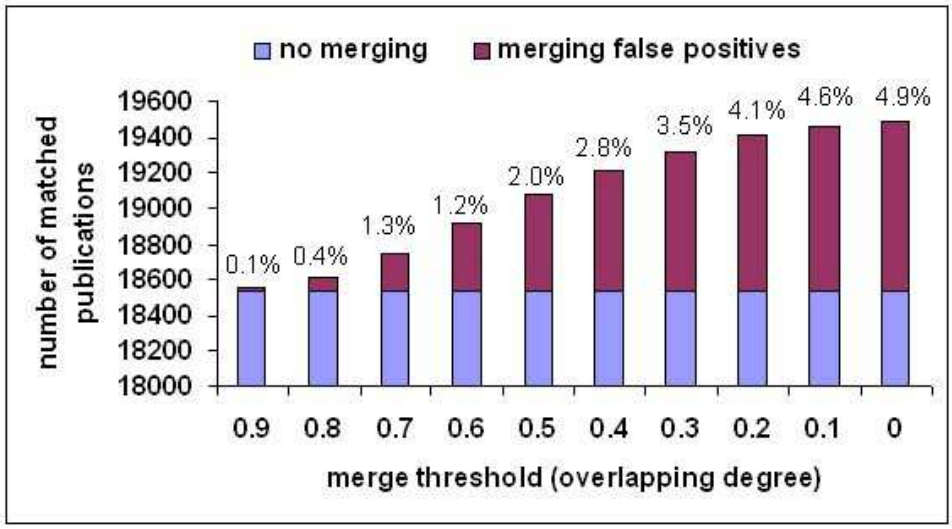


Figure 4.30: false positive vs. merge percentage

The merging of subscriptions may result in the introduction of false positive (i.e., unmatched publications that are forwarded into the network, but do not actually match the individual subscriptions.) The more subscriptions are merged, the larger the potential for false positives. We experiment with the possible impact of false positives based on a workload of 10,000 subscriptions and 20,000 publications. Figure 4.30 shows the number of false positives with the decrease of the merging threshold, which allows more subscriptions with little similarity to be merged and results in more non matched publications to be returned (in terms of the original workload.) The number of false positives is small and, therefore, tolerable, especially considering the decrease in the routing table size.

Effects of the Merging Choice on Predicate Matching Threshold

As we mentioned in the merging-based routing, there are many choices to aggregate matching thresholds of two predicates such as *min*, *max* and *average*. Different choice may result in different set of matched publications returned. In this section, we will experimentally evaluate the effects of these aggregation functions. A matched publication (against the merged subscription) is a correct return if it matches either of the original

subscriptions. Otherwise it is false positive. A false negative refers to a publication that should match one of the original subscriptions, but filters out (i.e., does not match) by the merged subscription. Use of *min* won't miss any correct matched publication, but may get some false positives. Use of *max* can reduce the number of false positives, but some correct publications may be missed. *average* is a compromise way to balance the effects of these two choices.

To measure false positives, we use *precision* which is defined as

$$precision = \frac{\#\text{correct matched publications returned}}{\#\text{all matched publications returned}}.$$

The measure of false negatives is evaluated by a measure *recall* which is defined as

$$recall = \frac{\#\text{correct matched publications returned}}{\#\text{all correct matched publications}}.$$

We use the F-measure which is a common metric to combine *precision* and *recall* in information systems to evaluate the effects of the above aggregation function. It is computed as

$$F - measure = \frac{2 * precision * recall}{precision + recall}$$

Figure 4.31 shows the comparison of the F-measure on the choices of these aggregation functions of thresholds for merged predicates. It can be observed that all choices have high F-measure (larger than 85%), while the result of the *average* and *min* are better than *max*. These two choices can achieve 97% for F-measure.

4.6 A-ToPSS System Implementation

To demonstrate the viability of the approximate publish/subscribe model, we implemented the Approximate Toronto Publish/Subscribe System (A-TOPSS). In this chapter, we describe the overall system architecture of A-TOPSS and features supported by the web interface. We will also explain the functions of a control panel that is used to adjust experimental values and monitor the behavior of the system.

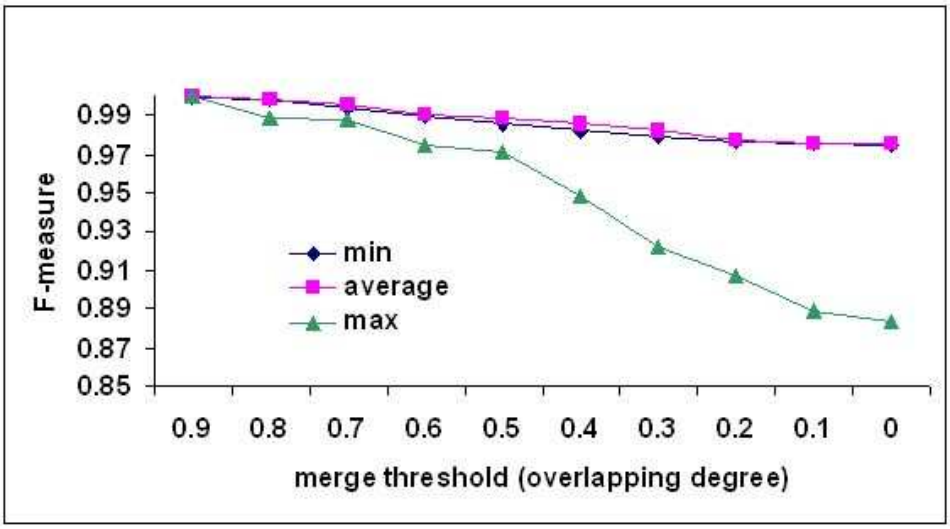


Figure 4.31: Effects of choices to aggregate matching thresholds

4.6.1 System Architecture

The main challenge in applying publish/subscribe systems to real world applications lies in the design of efficient matching algorithms that exhibit scalability. At Internet-scale, such a system has to be able to process millions of subscriptions and react to thousands of publications. The Approximate Toronto Publish/Subscribe System is implemented based on this consideration. The architecture of A-ToPSS for a real-life application is shown in Figure 4.32. Publishers and subscribers send requests through a web server (e.g., Apache) to the system. The requests include personal information registration, subscribing their interests and publishing data information. Subscriptions and publications are processed by a matching engine. At the same time, all of users' information passes through a script engine (e.g., PHP, JSP or Metahtml, etc.), and is stored in a database. The matching engine matches publications against subscriptions and returns the matched subscriptions to a notification engine. The pervasive notification engine sends different types of notifications (e.g. e-mail, ICQ, TCP/UDP, etc.) to the subscribers according to their requests.

As an experimental platform, A-ToPSS demonstrates two parts to evaluate different

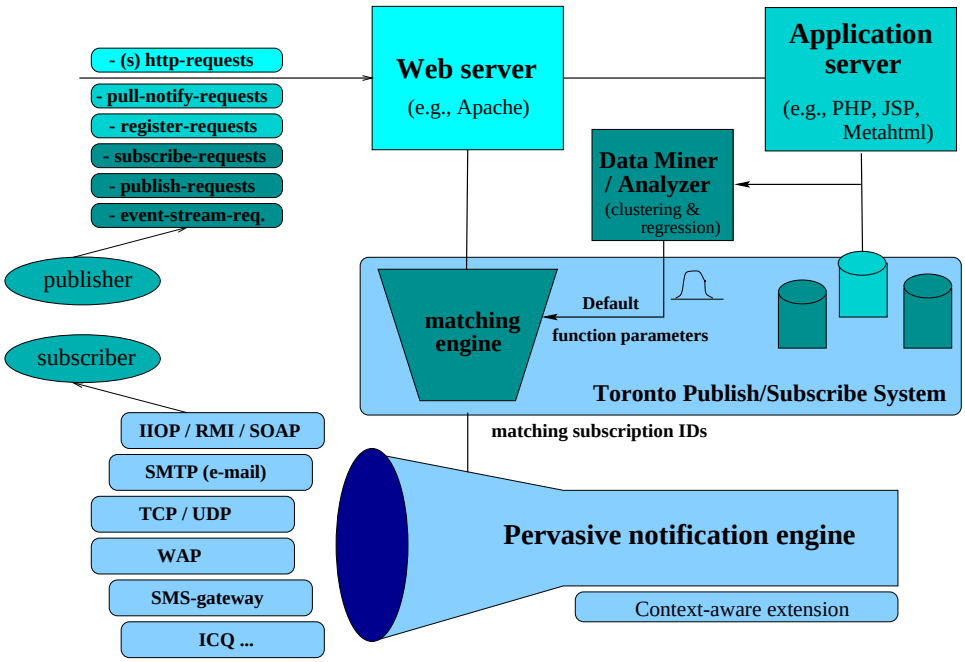


Figure 4.32: Overall Architecture of Publish/Subscribe System

subscription and publication language models and approximate matching algorithms. One involves the normal operations (e.g., subscribing, publishing and matching) in a publish/subscribe system. The other focuses on the experimental system operations to investigate the effects of different parameters in controlling our system. The demonstration system setup that integrates these two parts together is depicted in Figure 4.33.

The approximate matching engine receives subscriptions and publications simultaneously from a workload generator and a web server. The workload generator is running in the background to simulate a large number of other information providers and consumers in the real world. Once a publication comes, the approximate matching engine is triggered to match the coming publication against all the subscriptions in the system and send the matched subscriptions to the notification engine. This consists of the normal system operations part. An experiment control panel that manages system parameters sends the changes of parameters to the matching engine. And a monitoring panel is used

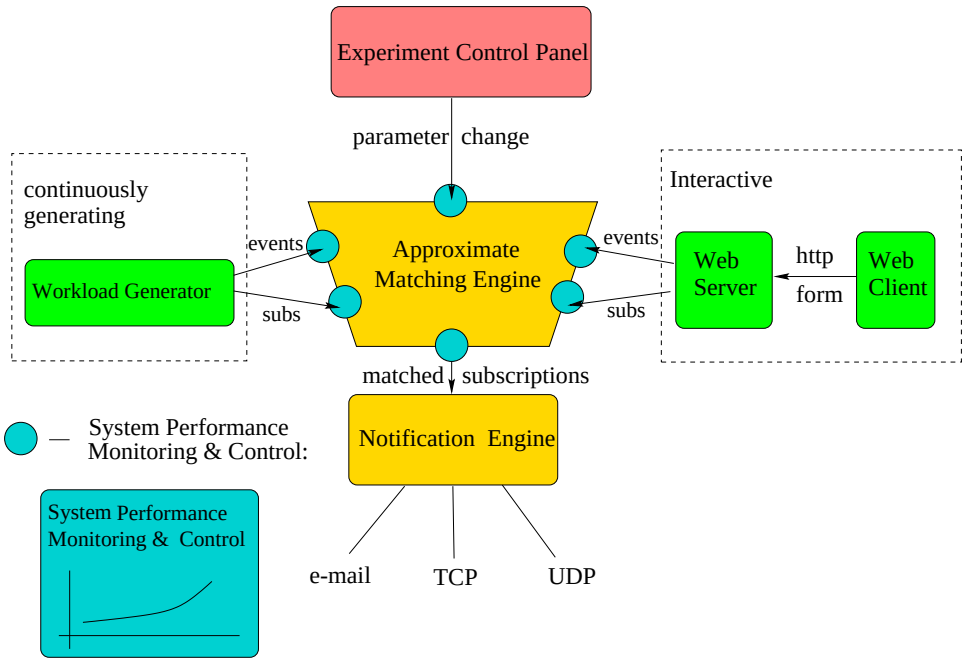


Figure 4.33: Demonstration Setup of Approximate Toronto Publish/Subscribe System (A-ToPSS).

to monitor the effect of these parameters on system performance. This consists of the experimental system operations part.

The workload generator can continuously generate new subscriptions and publications according to the specification specified in a configuration file. The workload generation of subscriptions and publications is controlled by two threads. There are other threads dealing with the receival of subscriptions and publications from the web server. The information transmission between the matching engine and the web server is realized by two data buffers: one for publications and the other for subscriptions. The threads in the web-server side put the publication or subscription into its individual buffer upon receiving a piece of information. The threads in the matching side read the buffers at a certain rate, take out the information if there is any, convert into structured data and stored them into the system. There is also a thread for controlling the sending of notifications (i.e., matched subscriptions) to the web server and a thread for handling the

deletion of information. In a real application, every subscription and publication is only valid for a period of time. We use a thread to control the deletion of invalid subscriptions and publications. The detail of the implementation design is shown in Figure 4.34.

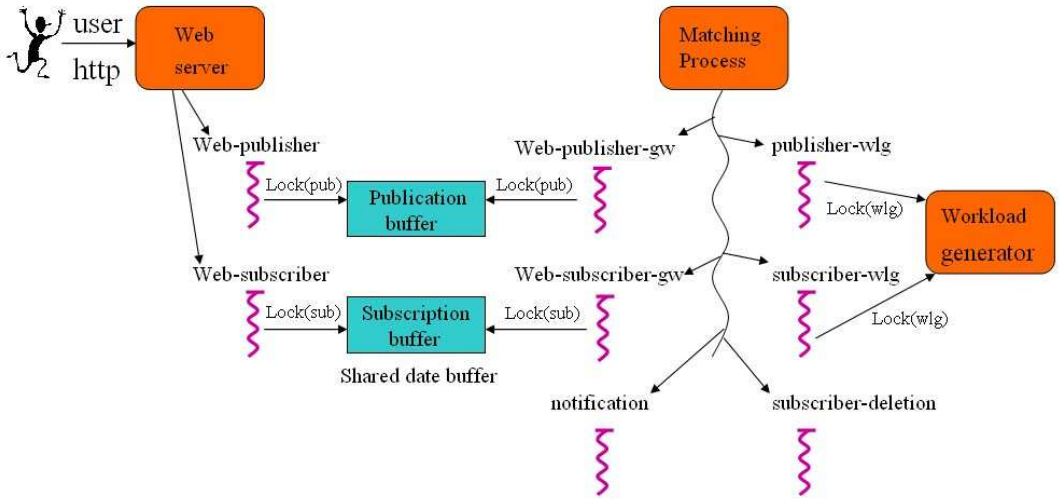


Figure 4.34: A–ToPSS implementation design

4.6.2 Web Interface

A-TOPSS provides a web interface for users to interact with the system. The interactive user interface is implemented by Meta-HTML web programming language. Meta-HTML is a powerful, extensible server-side programming language specifically designed for working on the World Wide Web. It resembles a hybrid of HTML and Lisp languages and has a huge existing function library, including supports for sockets, image creation, perl, GNU plot, etc. It is extensible in both Meta-HTML and other languages (C, etc.).

A-ToPSS offers four classes of normal operations: registration, subscribing, publishing and notification. The first time a user visits the web interface, registration is required to access the information resource. The user need to create an ID and set a password for herself. Personal information such as name, address is optional. However, the contact information relevant to the notification must be provided in order to successfully receive

notifications. For example, email address must be provided by the user if she wants to receive notifications via email. These are administrative operation, which is common to most web applications. Next, we will describe features specific to to publish/subscribe systems.

For simplicity, we will explain the operations for subscribing as an illustration. Operations for publishing are similar, and we won't elaborate here. There are two types of users in the system - administrators and regular users. Only administrators have the privilege to create new subscription types, edit the existing ones or delete them. Subscription types are templates for subscriptions. These templates specify the number of predicates and whether an attribute accepts crisp or approximate value. Before the modification or deletion of a subscription type, system will check whether any subscription is using under this type. Subscription types can only be edited when no subscription is defined under it.

The user-level operations on subscriptions are designed for the ordinary users. Subscribers can add a new subscription, edit or delete the subscriptions they defined before. When adding a new subscription, the user first chooses a type, then our system will ask users to input corresponding information according to the requirements specified by the subscription type. For crisp subscriptions, users need to provide attribute names, operators (e.g., $>, <, =, \neq$ and $\neq$) and values (e.g., integers, floats, strings, etc.). For approximate subscriptions, it is more complicated. Other than attribute names, users need to provide the number of approximate constraints for each attribute. For example, the "price" attribute may have 3 approximate constraints that are "expensive", "reasonable" and "cheap". For the representation of each constraint, the web interface gives the flexibility for users to describe predicates. A user chooses among a family of functions to represent the uncertain information. Figure 4.35 shows a screen shot of the subscription entry panel of our system, where a user can view and adapt the membership function representing her predicate.

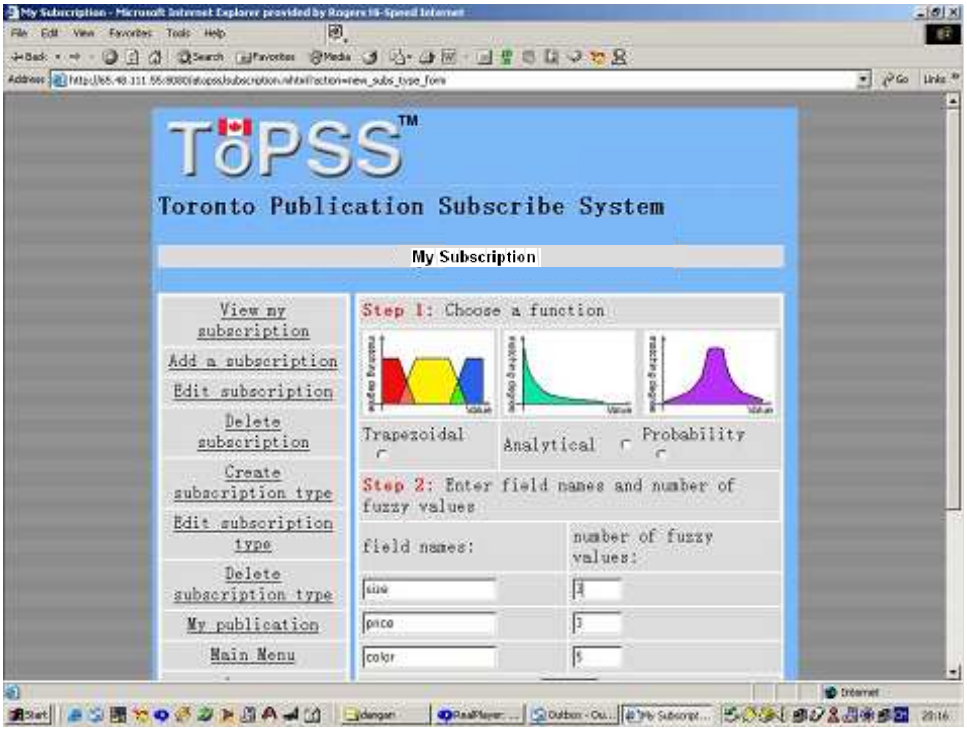


Figure 4.35: The power user's interface for defining approximate subscriptions.

After users submit subscriptions and publications, their information will be stored in a database, at the same time, transmitted to the matching engine to be processed. After the matching, matched subscriptions are sent back to the web interface and stored into the database. For the moment, A-ToPSS supports notification only by a pull model. When a user click the "notification" button, the results of matching of her subscriptions will be displayed on the web. A link to the publication which matches her subscription is also offered. The user can browse the provided information for further use. If any subscription or publication is deleted, the match related to it will be broken and won't be sent back to users.

4.6.3 Control and Monitoring Experiments

There are many variables, such as users' satisfaction thresholds and publication rate, that may affect system behavior. In order to illustrate the effects of these parameters have

on the performance of the system, we developed a control panel for adjusting the values of system parameters and a monitoring panel for display system metric and observing the system behavior in real time. Both the control panel and the monitoring panel are written as Java applets.

To demonstrate the differences between the crisp and approximate publish/subscribe models, for each model we deploy an experiment control panel (a java applet) where users can manage the change of parameters, and a monitoring panel (a java applet) that observes and displays system metrics. Figure 4.36 is a screen shot for part of the control panel. On the control panel, users can adjust the following parameters (for both crisp

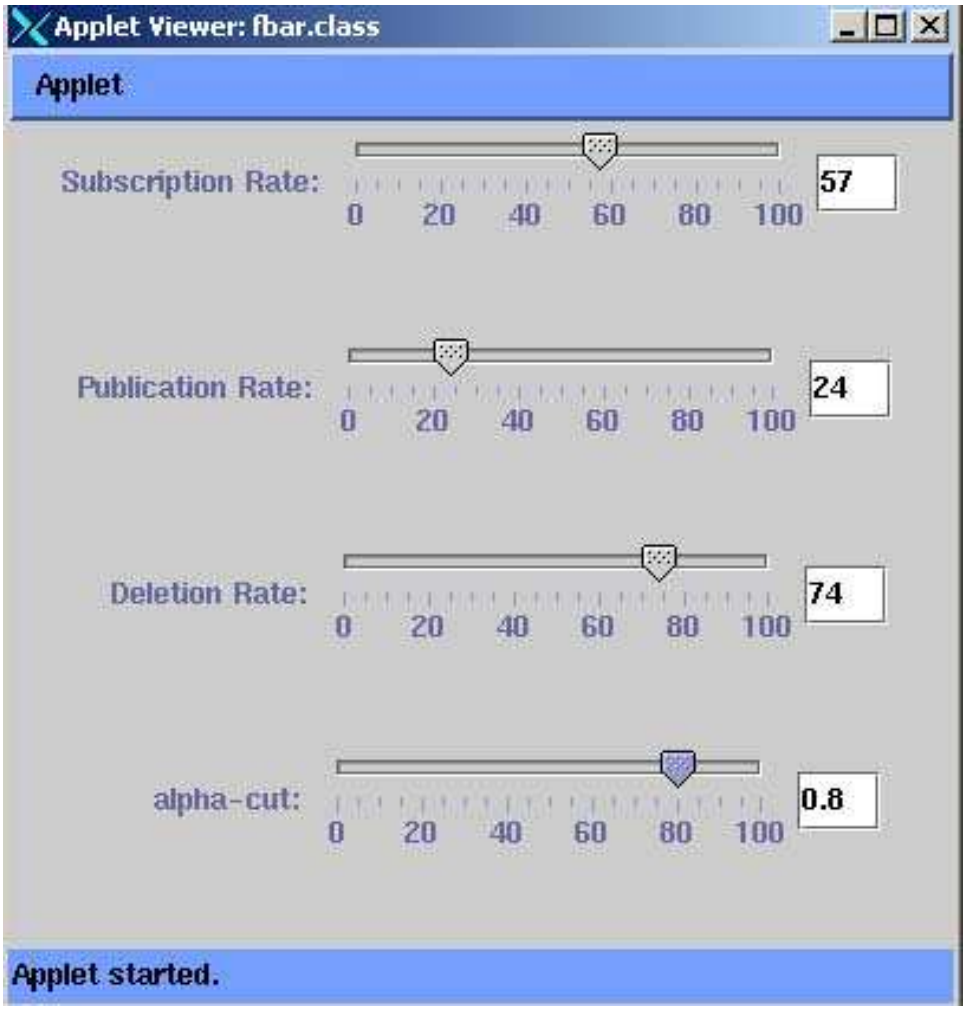


Figure 4.36: The control panel

model and approximate model):

- rate of subscription generation
- rate of publication generation
- rate of subscription deletion
- rate of publication deletion
- thresholds of users' satisfaction

Since the number of predicates and subscriptions in the system is quite large, it is difficult to control the thresholds for each predicate or subscription. In the control panel, we use the one pair of thresholds for all subscriptions to check their overall matching degrees. The control of the representation of membership functions is implemented in the normal system operations part. Users can choose a form from a function family and adjust the shape of the function according to their own specification. The effects of the representation of functions on the number of matched subscriptions is still in progress.

On the monitoring panels, the following metrics are observed and displayed:

- subscription loading time
- matching time
- number of matched predicates
- number of matched subscriptions

These metrics are taken at monitoring and control points indicated in Figure 4.33. This part aims at both experimenting with the matching model to demonstrate and exploring its degrees of freedom. We can see that with the increase of the subscription thresholds, the number of matched subscriptions decreases, as what we expect. Figure 4.37 shows the monitoring panel.

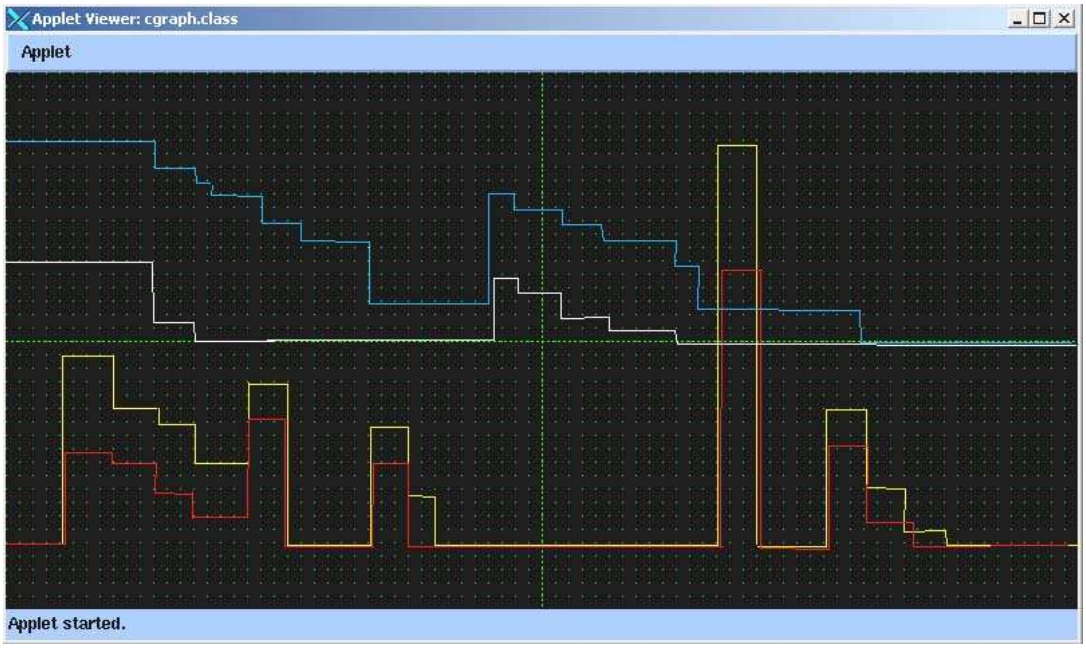


Figure 4.37: The monitoring panel

Chapter 5

Filtering of Graph-Based Metadata on Computing Cluster

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 providers. In this chapter we develop algorithms to enable efficient filtering of RSS documents, which is in a graph structured data format, on a computing cluster.

The current RSS feed aggregators use pull-based architectures, where the aggregator pulls RSS feeds from a web site that hosts the feed. 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 paradigm follows such an architecture and is well suited for structuring of large and dynamic systems such as RSS feed filtering, for instance. 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 [3, 34, 5, 28]. 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.

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 [28]. 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. In this chapter, we describe G-ToPSS, a graph-based publish/ subscribe architecture for dissemination of RSS feed.

At internet scale the, challenges of filtering graph-structured data as required by a RSS feed aggregator include large query populations, overlap among queries, high data rates and large number of matches. To address these challenges, in this chapter we also develop algorithms to effectively distribute RSS filtering over a computer cluster,

demonstrating improved scalability by increasing the number of compute nodes in the cluster.

In this chapter, Section 5.1 describes the architecture of a publish/subscribe system for a compute cluster. The G-ToPSS publish/ subscribe model supporting graph-based data matching is developed in Section 5.2. Section 5.3 develops the data structure and two novel subscription indexing algorithms and a pipelined filtering algorithm designed for the compute cluster. Section 5.4 presents our experimental evaluation. Finally a real world application implementation is described in Section 5.5.

5.1 Architecture of the Graph-Based Metadata Matching Engine

In a clustered system, we organize a set of machines into a two level network. One is selected as the front-end *server*. The remaining machines are the *back-end nodes* that connect to the server directly. Figure 5.1 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. An index is built up 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.

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.

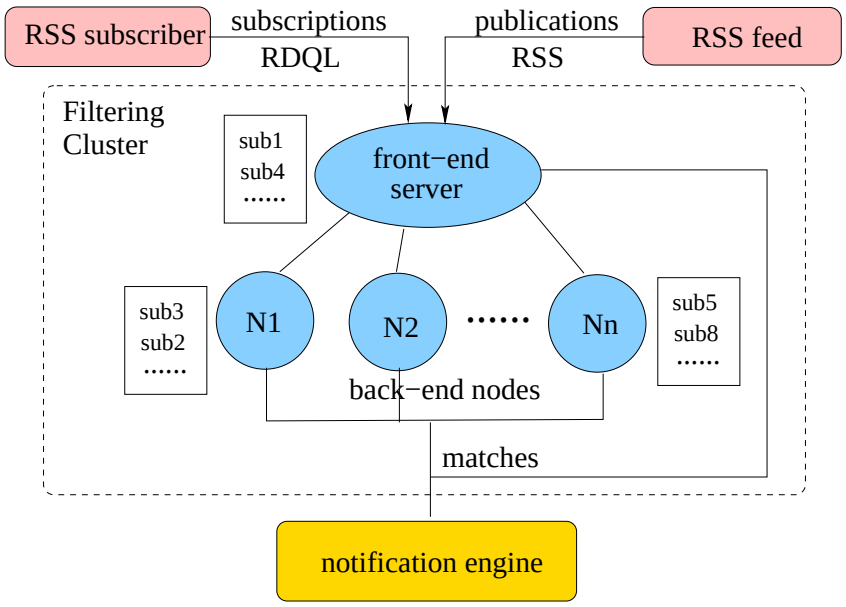


Figure 5.1: Publish/Subscribe Cluster

5.1.1 Subscription Partitioning

The server is responsible for partitioning subscriptions among the back-end nodes in order to maximize throughput. All events enter the system via the server[1]. The server acts as an *index* of subscription partition.

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. In this chapter, we developed

[1]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 others as slaves. The master is responsible for doing index updates, while slaves only do index lookups.

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.

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 hotspots by distributing the filtering among a number of back-end nodes. The server index is created based on the semantics of the subscriptions and access frequency. 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 events are most likely to be part of the filtering result set. Consequently, we distribute the job of determining those results among many back-end 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 $S_1 \succeq S_2$. If we denote by E_1 and E_2 the set of publications that match subscription S_1 and S_2, respectively, then $E_2 \subseteq E_1$. 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. 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.

Access Frequency Partitioning: Subscriptions that are not in a containment relation may still relate to each other based on the event result set in which they appear.

Subscriptions that frequently appear together in the same event result set, can be merged into a new subscription. 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 $S_M \succeq S_2$.

5.1.2 Pipelined Filtering

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 the publication will be delivered to client nodes for further filtering.

The system throughput on a cluster pub/sub can be increased, compared to the 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.

5.2 Graph-Based Publish/Subscribe Model

5.2.1 Language and Data Model

In this section, we describe the four components of the graph-structured data model: publications, subscriptions, matching semantics and ontology support. Publications are RDF documents. Subscriptions are queries for filtering of RDF documents following certain patterns. Our subscription language model is similar to RDQL (RDF Query Language), but the difference is that RDQL is a *typed* language featuring variables on

labels for nodes (classes) and edges (properties). However, our G-ToPSS model only supports variables on node labels and opts to include ontology information in a separate taxonomy. We refer to our subscription language as *GQL*.

Publication Data Model

A G-ToPSS publication is an RDF document, which is represented as a *directed labelled graph*. By the specification of RDF semantics by Pat Hayes, an RDF graph can be represented by a set of triples *(subject, property, object)*. Each triple represents by a node-edge-node link (as shown in Figure 5.2). *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*.

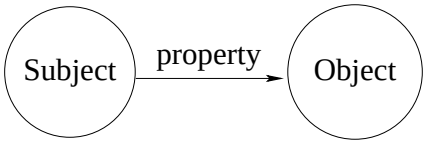


Figure 5.2: RDF triple graph

For short, we represent the triple *(subject, property, object)* simply as (s, p, o). Formally, a G-ToPSS publication is defined as a set of triples:

$$p = \{(s_1, p_1, o_1), (s_2, p_2, o_2), \cdots, (s_n, p_n, o_n)\}.$$

Figure 5.3 illustrates a university web page about the G-ToPSS project under the supervision of Prof A. Notice that this publication is in a graph format. Prof A got tenure in the same year as when the project G-ToPSS was published. The same object (2005) is used in these two links pointed by two different nodes. Such graph-structured data cannot be processed by XML filter, but can be supported by our G-ToPSS matching engine.

The example publication as shown in Figure 5.3 can be represented formally as:

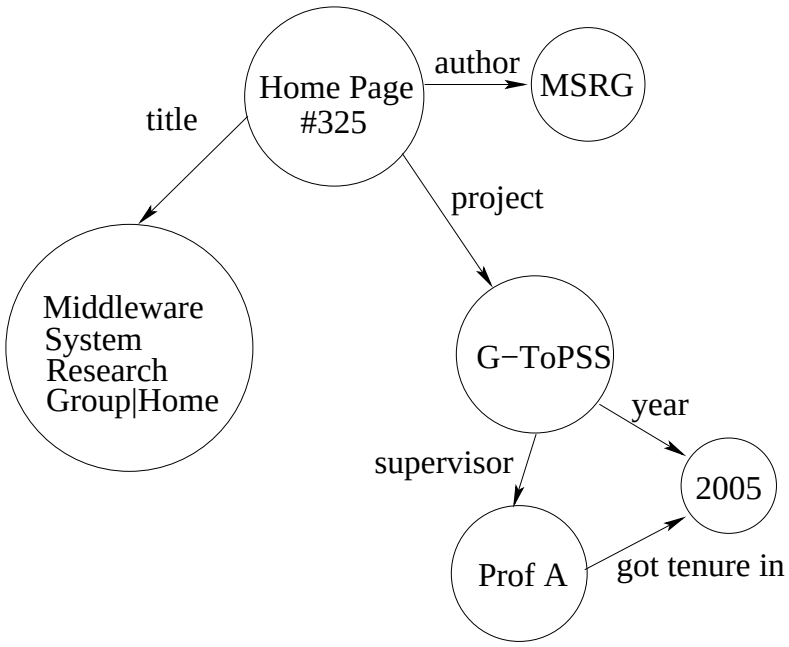


Figure 5.3: G-ToPSS Publication Example

```
Webpage = {(HomePage#325,project,G-ToPSS),(HomePage#325,author, MSRG),
           (HomePage#325,title,Middleware System Research Group),
           (G-ToPSS,supervisor,Prof A),((Prof A, got tenure,2005),
           (G-ToPSS,year,2005)}
```

Subscription Language Model

A G-ToPSS subscription is a *directed graph pattern* specifying the structure of the publication graph with optional constraints on 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 represents 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. $\leq$ is "descendantOf" which means that variable $?x$ is an instance of a descendant of class v. $\geq$ is "ancesterOf" which means that $?x$ is an instance of an ancestor of class v. $=$ means that $?x$ is the direct instance of class v (i.e., a child of v).

In summary, we represent the 5-tuple *(subject, property, object, constraintSet (subject), constraintSet (object))* simply as $t = (s, p, o, (s, op_s, v_s)?, (o, op_o, v_o)?))$. Formally, a G-ToPSS subscription is defined as a set of 5-tuples:

$$\begin{aligned} s &= \{t_1, t_2, \cdots, t_m\} \\ &= \{(s_1, p_1, o_1, (s_1, op_{s_1}, v_{s_1})?, (o_1, op_{o_1}, v_{o_1})?), \cdots, \\ &\quad (s_m, p_m, o_m, (s_m, op_{s_m}, v_{s_m})?, (o_m, op_{o_m}, v_{o_m})?)\} \end{aligned}$$

For example, Figure 5.4 illustrates a subscription that specifies interest in a web page which is about the G-ToPSS project supervised by Prof A who got tenure before the year 2006. The same subscription can also be represented declaratively by the following RDQL:

```
SELECT * FROM Webpages WHERE
      (HomePage#325,project,G-ToPSS) AND
      (G-ToPSS,supervisor,Prof A) AND
      ((Prof A, got tenure,?x) AND (?x,<,2006))
```

The subscription in Figure 5.5 is looking for a web page about a new project after 2004. 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 about

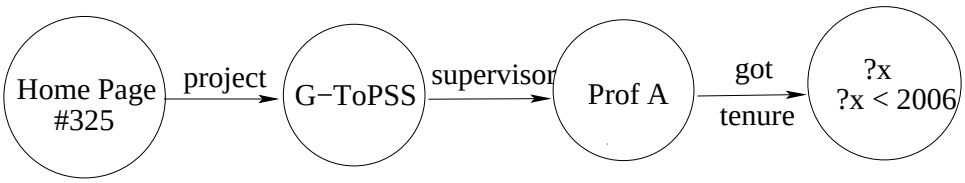


Figure 5.4: Subscription S_1

HomePage which is a descendant of the "Academia" class is going to match (refer to Figure 5.6). This subscription can be represented by the following RDQL:

```
SELECT * FROM Webpages WHERE
       (?y,project,G-ToPSS) AND
       (G-ToPSS,year,?z) AND
       ((?y, <=, Academia) AND (?z,>,2004))
```

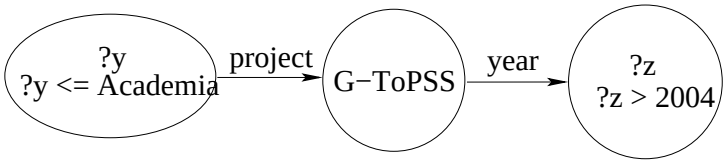


Figure 5.5: Subscription S_2

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. Graph pattern G_S is *embedded* in G_P if G_S is isomorphic to a subgraph of G_P and all constraints of G_S are satisfied.

Conretely 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 5.4 is matched by the publication in Figure 5.3 since the publication contains the same links (*Home Page* ♯325*, project, G-ToPSS*), (*G-ToPSS, supervisor, Prof A*), and (*2005 < 2006*); thus (*Prof A, got tenure,*?x(?x,<,*2006*)) are satisfied.

Ontology Support

An 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. An RDF schema supports constrain *is-a* relationship on *properties* (i.e., represented by the edge between *subject* and *object*). However, to simplify the system design, we only support the taxonomy information about *subject* and *object* nodes in our G-ToPSS model. As explained in the following section, structure matching and constraint matching are separate stages in the matching algorithm. It is straight forward to extend the current model to support other RDF schema semantics (e.g., *subPropertyOf*, *Datatype*, etc.).

G-ToPSS allows the designer to use multiple inheritance in the taxonomy, with the restriction that the taxonomy must be acyclic. The taxonomy contains the hierarchy of all classes and lists all instances of a class. 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 5.6, we show an example of a class taxonomy about an academic webpages system. Boxes represent classes and circles represent instances. Class “Academia” includes two subclasses: “Research Lab” and “University”. Class “Middleware Group” includes “Pub/Sub System Development” and “Aspect Oriented Software Development” two subclasses. The document instance “Home page ♯325” belongs to both “UofT” and “Pub/Sub System Development”.

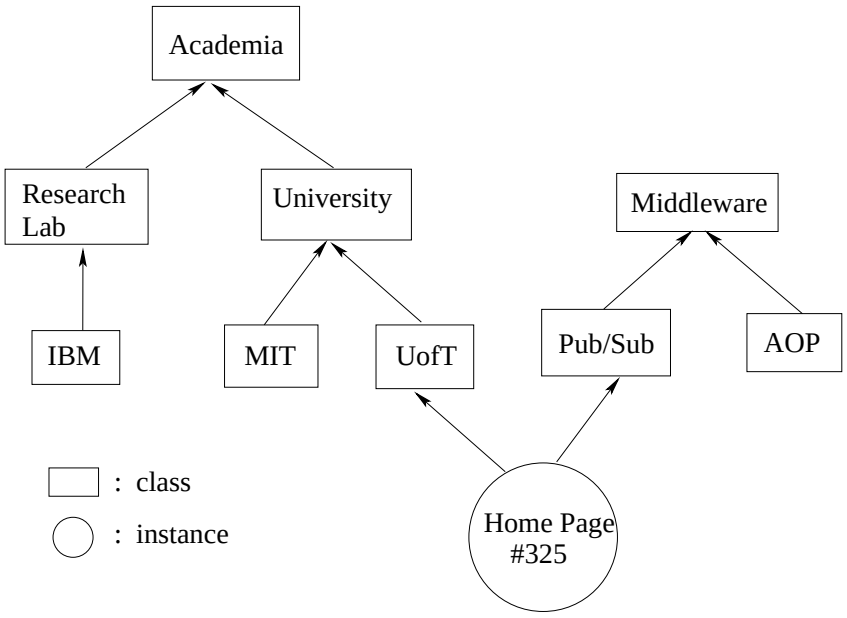


Figure 5.6: Example taxonomy

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

5.3 Data Structures and Algorithms

To exploit overlap between subscriptions we integrate all subscriptions into a single graph. We denote the graph containing all subscriptions as G_M. Figure 5.7 shows an example of G_M which combines two subscriptions S_1 and S_2 as shown in Figure 5.4 and Figure 5.5. After we integrated both S_1 and S_2, the matrix graph G_M may contain cycles or nodes with multiple parents. For example, in Figure 5.7, node $\star 1$ represents both variable $?x$ in subscription S_1 and variable $?y$ in subscription S_2 and it is pointed by two different parental nodes, forming a graph-structured query language model, which is more general and expressive than a tree-structured data and query model (i.e., XML and XPath).

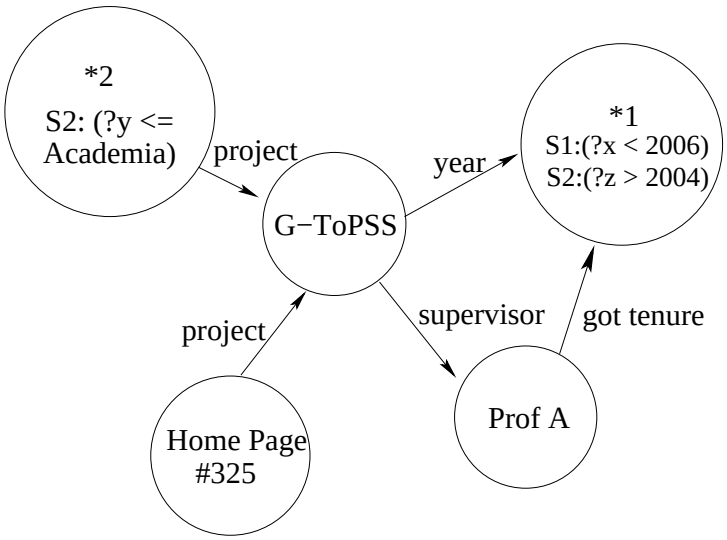


Figure 5.7: Matrix G_M contains both S_1 and S_2

Given all subscriptions, G_M and 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} that are subscriptions, in G_M that match some subgraph of G_P.

This matching problem is a specialization of subgraph isomorphism [91]. The subgraph isomorphism problem is defined as follows: given graphs G_1 and G_2, identify all subgraphs of G_2 which are isomorphic to G_1. The general subgraph isomorphism [48] only concerns the structure of two graphs (i.e., the adjacency relationship of verticies). However, in the graph-based publish/subscribe model, pattern graph maps to a data graph if the topology (structure) of the two graphs matches and all variable constraints (literal and ontology) are satisfied. Thus, the G-ToPSS matching problem is specilized by imposing more restrictions on verticies labels, edges labels and variable constraints in addition to the structure restriction. Based on this specialization, we propose a data structure to index all subscriptions and an efficient matching algorithm whose complexity is linear with the number of matches.

5.3.1 Data Structures

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.

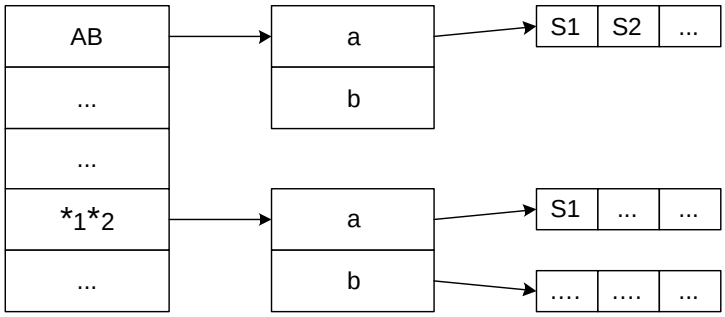


Figure 5.8: Data Structure

Figure 5.8 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 example, Figures 5.4 and 5.5 show two subscription graphs containing variables and the merged subscription graph, G_M, in Figure 5.7.

The data structure from Figure 5.8 allows us to store uniquely labeled 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 5.7, we use one node labeled 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.

We use a graph G_M to contain all subscriptions. Next, we discuss how G_M is created when inserting subscriptions. Suppose 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 $0 < i < |G_S.\star|$. $G_M.\star$ is the number of stars in 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. $newTable(A, B)$ 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. $\quad G_M.\star = G_S.\star$
3. **for** each edge $E \in G_S.edges$
4. $\quad T2 = G_M.T1.getTable(E.v, E.w)$
5. $\quad$ **if** ($T2$ is null)
6. $\quad\quad T2 = G_M.T1.insert(E.v, E.w)$
7. $\quad E2 = T2.getEdge(E)$
8. $\quad$ **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 inserting 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 $G_M.subs$.

5.3.2 Filtering Algorithm on a Single Node

Original Filtering Algorithm

In this section, we explain how to perform matching using the subscription graph G_M when a publication arrives. G_P is the publication graph (the number of edges in G_E is m). G'_P is a completed graph containing vertices $E.v$, $E.w$, $\star_i$ such that $0 < i < |G_M.\star| + 1$. All nodes in G_P are unique. *SubSet* contains all subscriptions that have at least one edge in G_M that are referenced by G_P. *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 $match(G_P)$

1. **for** each $E \in G_P.edges$
2. create a fully connected graph G'_P
3. **for** each edge $E2 \in G'_P$
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 ∈ 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 ∈ SubSet
14.     if (S.edgeCount ≥ |S.edges|)
15.         S.edgeCount = 0
16.         b = E.smEdge.bindingTable|E ∈ S
17.         for every edge E2 ∈ S.edges − E
18.             b = b ⋈ E2.smEdge.bindingTable
19.         for every row R ∈ 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 corresponding 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, it can be matched by edges v_1v_2, $v_1\star_i$, $\star_iv_2$ and $\star_i\star_j$ in G_M. There are three actions to perform on these potentially matching edges. (1) Add v_1v_2 into the binding tables of all matching edges so that they can be used in the second stage. (2) Increase the counters of subscriptions associated with these edges. (3) Put the subscriptions into *Subset* as matching candidates. 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 5.9 provides 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.

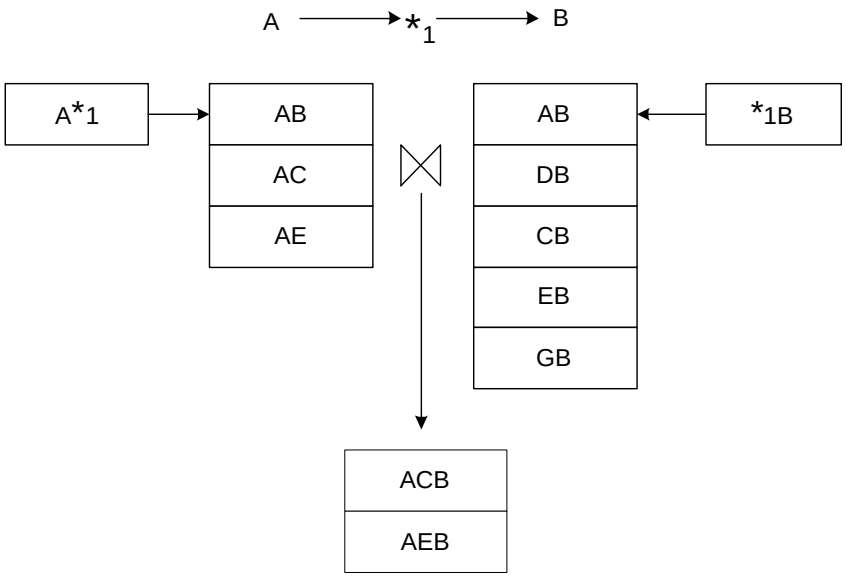


Figure 5.9: Binding table join

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)

1. **if** $op = LT$ **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 5.7, for subscription s_2, $\star_2$ is matched by node "2005" since $2005 > 2004$ and $\star_1$ is matched by node "Home Page ♯325" since it is descendant of class "Academia."

Optimized Filtering Algorithm

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). As an example, Figures 5.7 shows how two subscriptions are combined into one subscription matrix graph. Given all 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.

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 that contains all the pairs of vertices that the edge between them has label E_i. Each entry of the second table points to a list of subscriptions that contain the edge between this 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 5.10. 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.

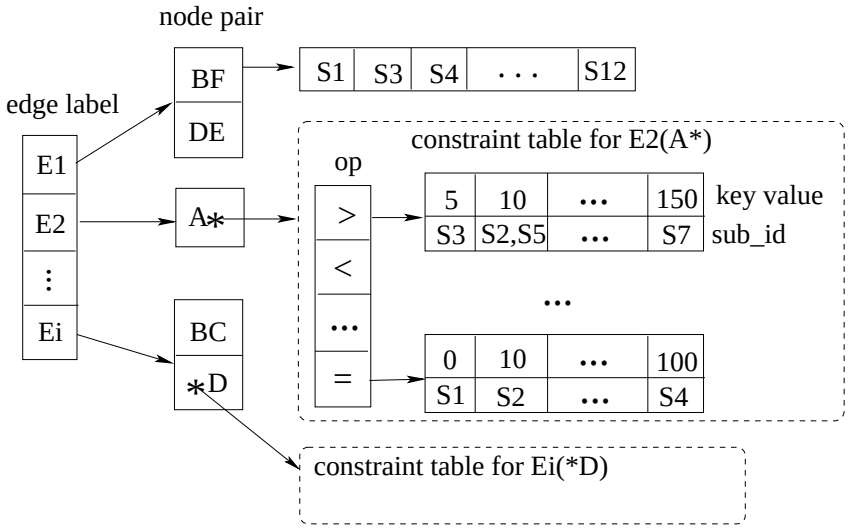


Figure 5.10: G-ToPSS Data Structure

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 $improvedMatch(G_P)$

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. $e'.bindingTable+ = (v, w)$; $SubSet = SubSet + s$
8. $result = \emptyset$
9. **for** each subscriptions $s \in SubSet$
10. join binding table $e.bindingTable$ for each edge $e \in s.edges$
11. **if** the final binding table is not empty
12. result = result + s

Algorithm $improvedMatch$ is the optimized algorithm for matching publications against indexing subscriptions on the server to get the satisfied indexes, which is used to determine which client the publication will be sent. 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 "$\neq$" 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 look-up, hence speeding up the matching.

Filtering Algorithm 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, the space complexity is

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

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: For the procedure of insert a subscription into the system, he insert(G_S) algorithm iterates for every edge in the coming subscription, locate the corresponding list associated with the edge and add an entry of the coming subscription into the list. Thus, the insert algorithm depends on the number of edges for each subscription and the time complexity is

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

To form the graph G_M which contains all subscriptions, we have to insert subscriptions one by one. Therefore, the time to load a batch of subscriptions at a time is $\sum_{s_i} |G_{S_i}.edges|$. Since the number of edges in each subscription is very small, the time complexity of loading subscription is

$$O(number\_of\_subscriptions).$$

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

$$O(l^k)$$

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 includes two parts: checking the constraints for class variables according to the class taxonomy and checking the constraints for literal variables according to publication literal values. The time for the first part 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

$$O(mk^2) + O(n_1 * l^k + n_2 * k * d^t + n_2 * k * \gamma)$$

where n_1 is the number of subscriptions in $SubSet$, n_2 is the number of subscriptions among $SubSet$ having possible bindings and γ is the unit time of constraint checking for one literal variable. 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). $n_1 \simeq n_2 \simeq n$ where n is around the number of matched subscriptions. Also in the real applications, $m << n$. Therefore, the overall matching time is linear with the number of matched subscriptions:

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

Complexity Analysis for Optimization: The matching algorithm consists of two steps. The time for the first step depends on the number of edges in the publication. In the second step, for each subscription in $SubSet$, a join operation is performed for each pair of connected edges The computation of this 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 time of the join operations for one subscription to determine satisfying bindings for variables 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 and $SubSet$ can be considered as approximately equal to $|N_{structure\_matched\_subs}|$, compared to the large number of subscriptions that need to be filtered. Furthermore, $|N_{structure\_matched\_subs}|$ is approximately equal to the number of finally matched subscriptions. The overall matching time is thus linear in the number of matched subscriptions: $O(ratio_{match} * number\_of\_subscriptions)$.

From the above analysis, we conclude that the matching time to process a publication on a single node depends on the number of matched subscriptions. This leads to the intuition behind our indexing algorithms – to partition the subscriptions that are possibly matched together into disjoint sets to achieve maximum throughput.

5.3.3 Indexing Algorithms for a Cluster

The goal in designing a cluster-based publish/subscribe system is to partition the subscriptions that are possibly matched together into disjoint sets and delivered to different clients so that filtering can be processed concurrently and maximum throughput can be achieved. In this section, we first give a theoretical analysis behind our indexing algorithms. Then we describe two indexing algorithms to fulfill subscription partition. 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. At last, we present a pipelined filtering algorithm on the cluster.

Theoretical Analysis

Intuitively, we would cluster similar subscriptions together to speed processing publications. That is also the main goal for all existing subscription clustering papers such as [78, 69]. However, it is not the best solution for our matching algorithm. From our earlier work [73], we know that the filtering on a single machine depends on the number of matches. Therefore, in order to increase the system throughput, subscription indexing aims to distribute subscriptions that are possibly matched by the same publication to different machines. We will show below why subscription distribution is superior to subscription clustering.

Suppose there are a front server and k back-end nodes in the system. In order to achieve the maximum throughput, we assume there are k publications and the ith publication matches m_i subscriptions. Since the matching time depends on the number of matches, we assume the filtering time for a single match is t, then the filtering time of publication i is $m_i * t$ if all m_i matched subscriptions are located in one machine. Given these assumptions, next we will give a theoretical analysis for subscription clustering model and subscription distribution model, separately.

In the subscription clustering model, all similar subscriptions should be clustered together. Thus the optimal subscription partition is that each node process one publication and all its matches are stored in the same node so that all k publications can be processed in parallel. Thus the filtering time of the whole system (all k publications) depends on the processing of the publication which has the most matches, and the system filtering time would be

$$\max_i(m_i) * t.$$

In the subscription distribution model, subscription partitioning aims to distribute subscriptions that are possibly matched by the same publication to different machines. Based on this goal, for each publication, we distributed its matched subscriptions uniformly to k nodes. So each node ends up with m_i/k matches for the ith publication.

Since each back-end node contains m_i/k matched subscriptions for the ith publication, it has to be sent to every node to be processed in parallel and the filtering time would be $m_i/k * t$. Similar things happens for other publications. Thus the total system filtering time would be

$$\sum_i \frac{m_i}{k} * t.$$

Comparing the two filtering times $\max_i(m_i) * t$ and $\sum_i \frac{m_i}{k} * t$, it is obvious that $\sum_i \frac{m_i}{k} * t \leq \max_i(m_i) * t$. Therefore, the subscription distribution model is better than the subscription clustering model. In the next subsections, we will describe two concrete indexing algorithms to achieve the goal of subscription distribution.

Precise Indexing Algorithm 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 5.11 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 clients. In other words, there is no containment relation among the subscriptions in the index.

The subscription index based on containment is built incrementally with the incoming subscriptions. When a new subscription S coming into server. If there it is contained by other subscription in the index, S will be sent to a client. If there are other subscriptions are contained by S, these subscriptions will be partitioned equally into disjoint and sent to each client. Algorithm *containmentIndexing* describes the detailed operation to index

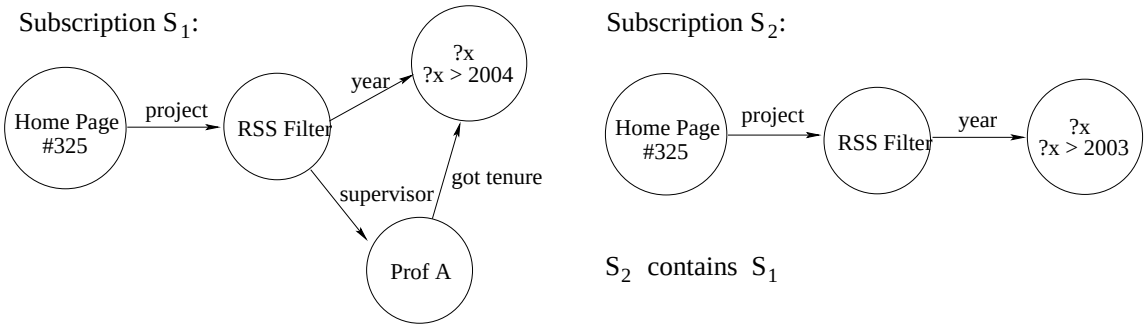


Figure 5.11: Example of Subscription Containment

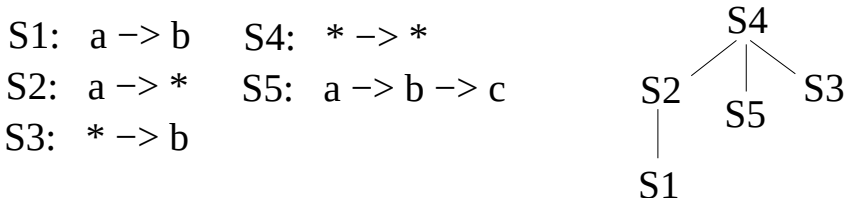


Figure 5.12: Example of Subscription Containment Tree

an incoming subscription S.

Algorithm *containmentIndexing*(S)

1. $R_{contained} = null$
2. //S is contained by other subscription
3. **if** (server.containmentChecking($S, R_{contained}$))
4. client o = server.getNextClient();
5. server.send(S, o)
6. subscription $S_c = R_{contained}$.getNextSub()
7. *client_set* O = server.index.get(S_c)
8. add o into O
9. server.index.put(S_c, O)
10. **else if** ($R_{contained}! = \emptyset$)
11. *client_set* O = server.index.get(S)
12. **for** each $S_i \in R_{contained}$
13. client o = server.getNextClient();
14. server.send(S_i, o)
15. add o into O

16. server.index.put(S_c, O)

17. **else** server.insert(S)

Let Σ be the subscription set in the index server, S is the new incoming subscription. This indexing method has the following property for a new subscription coming into the index server, we have the following property: If the new subscription is contained by a subscription in the index, no other subscription in the index can be contained by the new subscription. On the other hand, if the new subscription contains an existing subscription, it will not be contained by any other subscription in the index. This is formalized in the following theorem:

Theorem: For the new subscription S, (1) if $\exists S' \in \Sigma$ such that $S' \succeq S$, then $\forall S'' \in \Sigma$, we have $S \not\succeq S''$; (2) if $\exists S' \in \Sigma$ such that $S \succeq S'$, then $\forall S'' \in \Sigma$, we have $S'' \not\succeq S$.

Proof: We first prove (1). Suppose $\exists S'' \in \Sigma$ such that $S \succeq S''$, since $\exists S' \in \Sigma$ such that $S' \succeq S$, then we have $S' \succeq S''$. This contradicts the condition that there is no containment relation among the subscriptions in the index. Similarly for (2), suppose $\exists S'' \in \Sigma$ such that $S'' \succeq S$, since $\exists S' \in \Sigma$ such that $S \succeq S'$, then we have $S'' \succeq S'$. This contradicts the condition that there is no containment relation among the subscriptions in the index.

Query containment for relational databases is one of the most thoroughly investigated problems in database theory. Recently, conjunctive queries over trees also attracted quite some attention [40, 64]. There are many complexity results for XPath containment ranging from PTIME to Undecidable depending on the expressiveness of the language. In G-ToPSS model, we use the same definition of query containment to define subscription containment, but the subscription (query) containment is defined on labeled graphs, rather than XPath queries. Since the subscription index based on containment is built incrementally with the incoming subscriptions, we don't need to find out all containment relations among a set of subscriptions from the beginning. From the theorem, if the new

subscription is contained by a subscription in the index, no other subscription in the index can be contained by the new subscription. On the other hand, if the new subscription contains an existing subscription, it will not be contained by any other subscription in the index. Based on these observations, we propose an algorithm to determine all subscription containment relations for the newly coming subscription in linear time.

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*

Algorithm *containmentChecking* is executed for each new subscription that is to be added to the index. If G_S is already contained by any existing subscription the algorithm stops. Otherwise, a set of subscriptions that are contained by G_S will be returned. There are two stages.

In the first stage of the algorithm, we use the edge label to get a set of subscriptions that contain the same edge as the incoming subscription. for each incoming edge $e(v, w)$ in S, we use the edge label to get a set of candidate edges $e2(v2, w2)$ in G_M, which has the same edge label but the two end nodes $(v2, w2)$ may be different from (v, w). For each subscription that contains $e2$, its counter is incremented by 1.

Therefore, if subscription S_i in G_M is contained by the arriving S, the condition that $S_i.edgeCnt \geq S.edges$ and $S_i.edges \geq S.edges$ is satisfied. On the other hand, if S_i contains the arriving S, the condition that $S.edges \geq S_i.edges$ and $S_i.edgeCnt \geq S_i.edges$ is satisfied. At the end of the first stage, only the subscriptions that satisfy either of the above two conditions will be processed in the next stage.

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 contained by S_i or S, then a containment relation is detected.

Figure 5.13 illustrates an example of checking containment 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 5.13. Joining of these three tables produces one entry

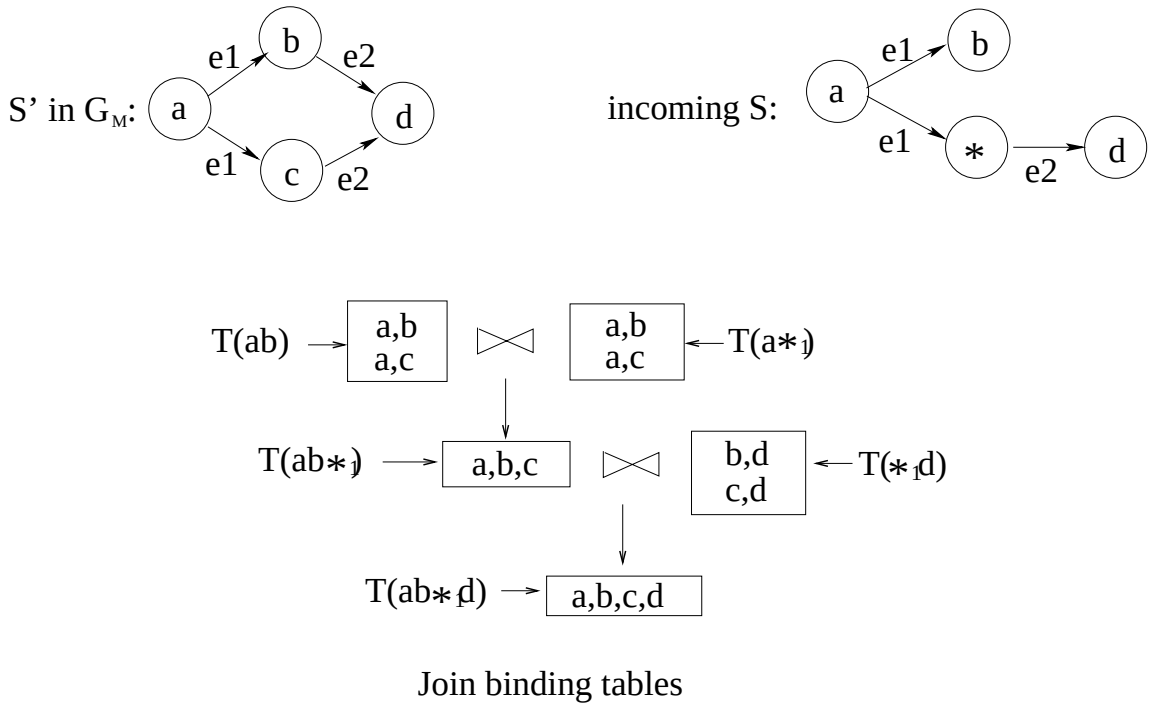


Figure 5.13: Computing containment relation

(a, b, c, d) in the result. Since (a, b, c, d) is contained by $(a, b, \star, d)$, we conclude that S' is contained by S.

Complexity Analysis: 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 containment relations is linear with the number of contained subscriptions, $O(ratio_{overlap} * number\_of\_subscriptions)$.

Imprecise Indexing Using Merging

The static structure relationship among subscriptions is one aspect that we explore to partition subscriptions. The second indexing algorithm identifies those subscriptions that

are frequently part of the same result set at run-time. Such subscriptions can be determined by analyzing run-time filtering statistics based on the matched event result set in which they appear. The identified subscriptions are removed from the index and partitioned into disjoint sets at the clients (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.

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 and a threshold T, we say P_1 and P_2 are similar if $\frac{|P_1.edges \cap P_2.edges|}{|P_1.edges \cup P_2.edges|} \geq T$.

The server keeps statistics on subscription access frequency through result set membership, as identified by an event. The event matrix E_M contains a set of *publication summaries* that have been processed. The publication summaries contain aggregate information about similar publications.

The statistic collection is done separately from the matching process and its effect on the matching process is negligible. The statistic collection could be performed outside of the cluster by duplicating the event stream (or some sample of it) and the index, as that is the only information that is required for the collection process. While the performance of the collection process is not important, we assume it does not have real-time result goals, and that it can even be done off-line. In the future, we are going to investigate sampled approach to statistic collection that introduces inaccuracies, but has better scalability. 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. $\quad\quad p_{new} = \text{merge}(p, p')$

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 merging index 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*. *mergeSubs* are removed from the index, and a new, merged, subscription is added to the index. *mergeSubs* are then partitioned into disjoint sets (and distributed to clients). The merging procedure is described in Algorithm *mergeIndexing*.

Algorithm *mergeIndexing*(C)

1. p = server.getLargestMergePub(E_M)
2. *mergeSubs* = server.match(p)
3. **while** ((*mergeSubs.size* > 1) **and** (*index.size* > C)) **do**
4. remove the first element S_i from *mergeSubs*;
5. **while** (there is any subscription S_j that the merged result $S_M = S_i \cap S_j$ is not empty) **do**
6. remove S_j from *mergeSubs*
7. client o = server.getNextClient();
8. server.send(S_i, o)
9. *client_set* O = server.index.get(S_i)
10. add o into O
11. server.index.put(S_i, O)
12. client o = server.getNextClient();
13. server.send(S_j, o)
14. *client_set* O = server.index.get(S_j)
15. add o into O
16. server.index.put(S_j, O)
17. $S_i = S_M$
18. server.containmentIndexing(S_M)

Complexity Analysis: The merge indexing algorithm 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 $max(mergeSubs.size, index.size - C)$.

For *how* to merge, the simple form of a perfect merger is defined as union of subscriptions: $S_M = S_1 \vee S_2$. Since each subscription is a graph in our model, the perfect merger does not reduce the number of subscriptions and the routing table size. On the other hand, due to the occurrence of variables in subscriptions, it is difficult to find a simple connected graph to represent a union of two graphs.

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.

Figure 5.14 shows an example of the merger. This merging rule satisfies the definition of a merged subscription and the representation is in a simple form which significantly reduces memory use. However, it forms an imprecise merger and introduces false positives. That's why this indexing algorithm based on merging is called imprecise indexing. For example, the publication shown in Figure 5.14 matches the index subscription (merger), but it does not match any of the original two subscriptions.

5.3.4 Index Maintenance

In a publish/subscribe 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. For example, an unsubscription of a previously contained subscriptions is no longer in the index (it was effectively removed due to the containment relation.) More importantly, in order to build a correct routing path after

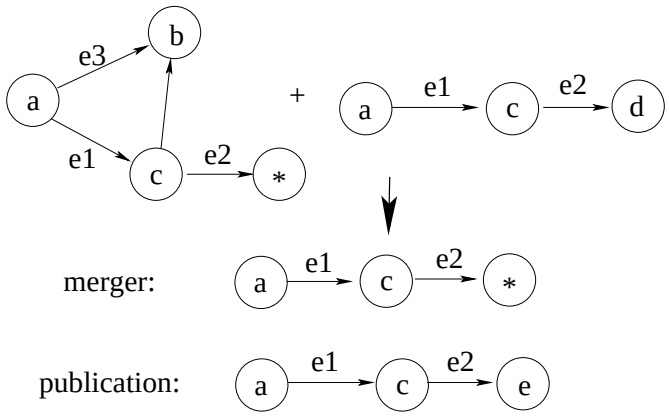


Figure 5.14: Merger of Subscriptions

unsubscribing, the contained subscriptions needs to be recovered when deleting the containing subscription. The same is done for unsubscribing merged subscriptions. Based on the containment and merging definition, we can easily obtain the following propositions.

Proposition 1: If subscription S is in the server index, S satisfies one and only one of the following properties: (1) S has no relation with other subscriptions; (2) S is a root of a containment tree; (3) S is a root of a merging tree.

Corollary 1: For an unsubscribe(S) message, if S is in the index, S is either an independent subscription or a root of a containment tree.

Proposition 2: The server index is complete, there is no other containment relations.

Proposition 3: If subscription S is not in the index, S satisfies one and only one of the following properties: (1) S is an inner node (or leaf) of a containment tree; (2) S is a leaf node of a merging tree.

Proposition 4: The leaves of a merging tree are either independent subscriptions or a root a containment tree.

When receiving an $unSub(S)$ message, it is simple if S is not in the index. We just forward $unSub(S)$ message to all client and the client which contains S will delete it. Deleting subscription from client 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.

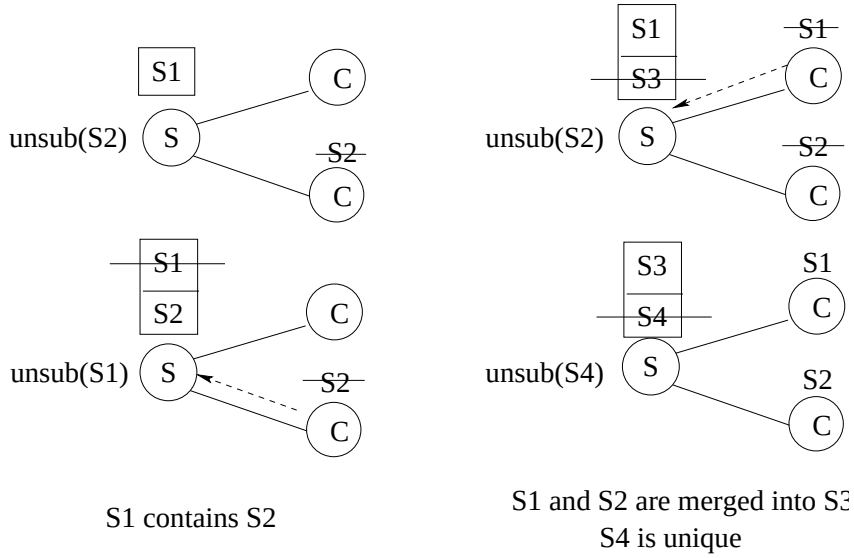


Figure 5.15: Deleting Subscriptions

Based on the above propositions, next we will discuss the detailed operation of subscription deletion for each indexing algorithm. When receiving an $unSub(S)$ message, we need to update the subscription index, containment trees and merging trees. For subscription index, delete S from the index if it exists; otherwise forward $unSub(S)$ to all clients, 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 subscription index. If S is an inner (or leaf) node, the index remains unchanged, delete S from client. 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 5.15 shows examples to *unSub* a subscription which has containment relation. S_3 is a merger of S_1 and S_2 and the server receives a $unSub(S_2)$ message. Then S_3 will be deleted from

the index and S_1 is moved from the client to the index (at the server). For the unique subscription S_4, just simply remove from the index.

5.3.5 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 Algorithm *filtering*.

Algorithm *filtering*(G_P)

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

Complexity Analysis: The goal of indexing and cluster filtering is to fulfill parallel processing, thus to increase the system throughput compared to a centralized setting. The increased throughput is obtained by trading off the cost of indexing subscriptions. The total filtering time depends on the server index lookup time and back-end node parallel matching time. From the above analysis, the matching time to process a publication on a single node depends on the number of matches. Suppose the time to find one match for

a publication is t, and the ith publication has m_i matches, then sequentially processing p publications in a centralized system cost $\sum_{i=1}^{p} m_i$ time. However, on a computer cluster with k back-end nodes, the pipelined filtering time is decreased to $p + \frac{1}{k}\sum_{i=1}^{p} m_i$ time, where p is the index lookup time at server, the other part is the parallel matching time on clients. When k is small, the total filtering time decreases linearly to the proportion of the increase of k. When k is large, the total filtering time is asymptotic to p as $k \to \infty$.

In the next section, we will examine the performance of indexing algorithms and cluster filtering algorithms to verify the above analysis.

5.4 Eexperiments

In this section we evaluate our approach to filter graph-based data on a computing cluster. After a brief description of the experiments setup. First we show the filtering performance on a centralized system. Next we evaluate the indexing algorithms required by a cluster environment. Finally, we experimentally evaluated the performance of the filtering system in a cluster environment and compared it to a centralized system to show that the throughput scales linearly with the number of cluster nodes.

5.4.1 Experiment Setup

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 5.1.

$Size_P$ 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

Table 5.1: 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

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 5.1. A publication is generated first. For example, for publication of size (k,m) we first generate a simple path of length $k-1$ and then we generate $m-k+1$ edges between random pairs of the k nodes.

Subscriptions are generated in four steps. 1.$ratio_{match}$ subscriptions that match the publication are generated by randomly selecting a subgraph of the publication. 2. Using same technique, overlapped subscriptions are generated as subgraphs from one big graph. 3. $N_{sub}*(1-overlap_s)$ non-overlapping subscriptions are generated randomly in the same way that the publication was generated. 4. 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 the others to their default values as specified in Table 5.1.

5.4.2 Filtering Performance on a Single Node

Number of subscriptions: Figure 5.16 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.

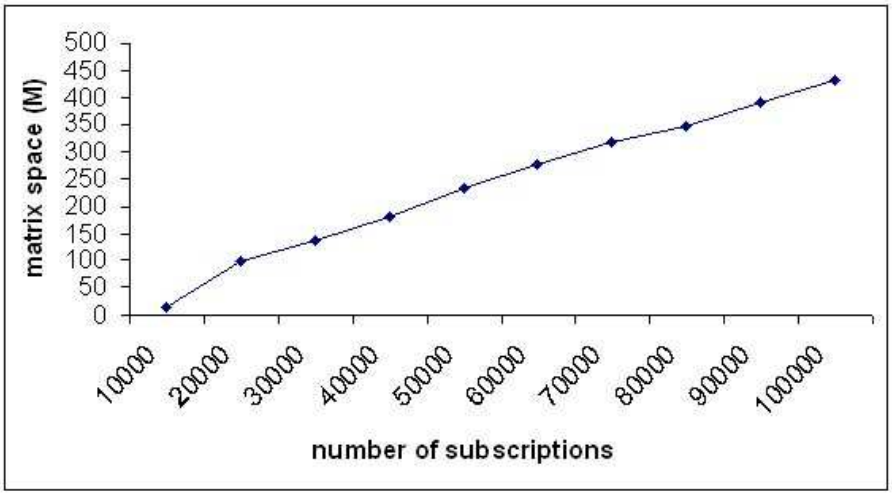


Figure 5.16: Memory vs. #subscriptions

Figure 5.17 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 increases.

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

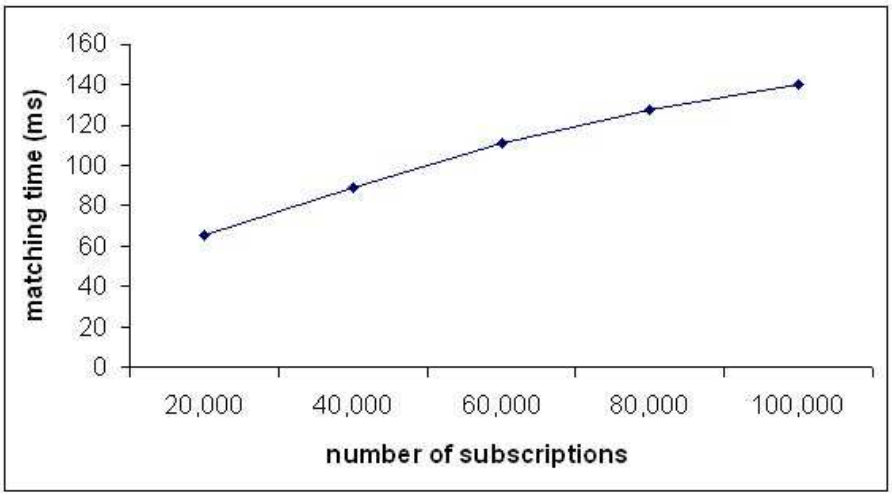


Figure 5.17: Matching time vs. #subscriptions

Subscription size: Figure 5.18 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.

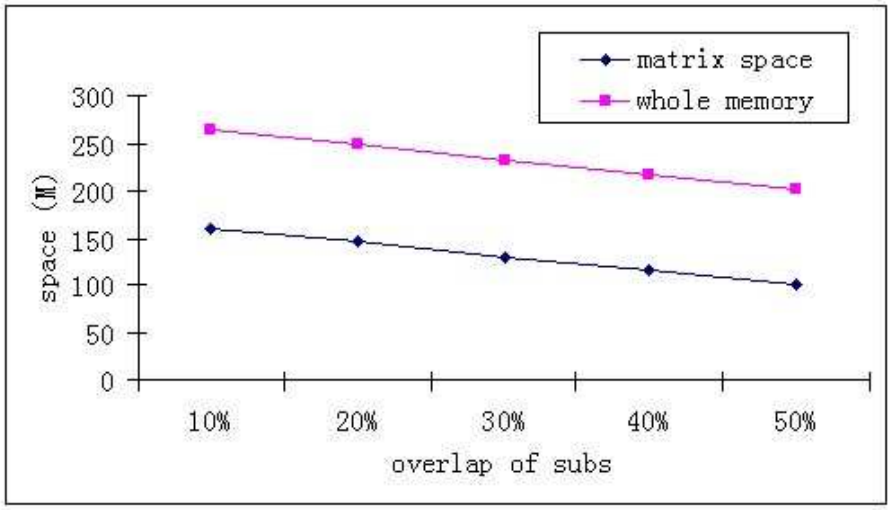


Figure 5.18: Memory vs. subscription overlap

Figure 5.19 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

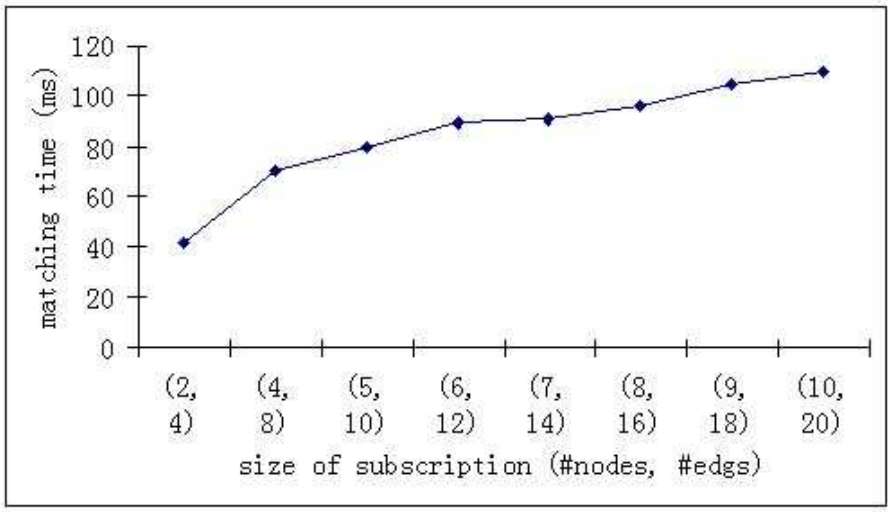


Figure 5.19: Matching time vs. subscription size

8), the time almost doubles. On the other hand, as the number of edges increases slowly, so does the increase of 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 5.20 shows the effect of increasing the number of subscriptions that match the publication. As this number grows, the time to match grows very rapidly. This is mainly due to increase in time to calculate all the bindings for each subscription.

G-ToPSS vs. Alternatives: In Figure 5.21 we compare the performance of our algorithm to the OPS algorithm [92]. As the graph shows, OPS matching time increases very rapidly 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 [48], 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

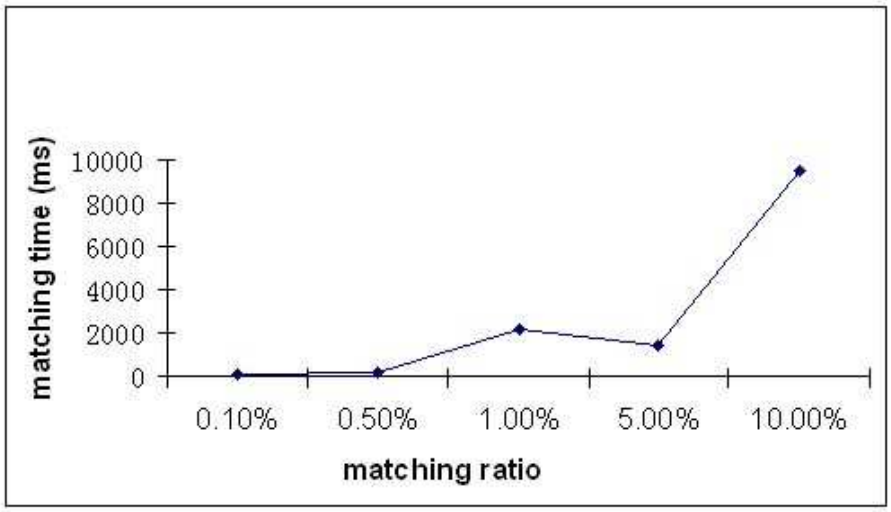


Figure 5.20: Matching time vs. matching ratio

graph and the nodes in a RDF publication are unique.

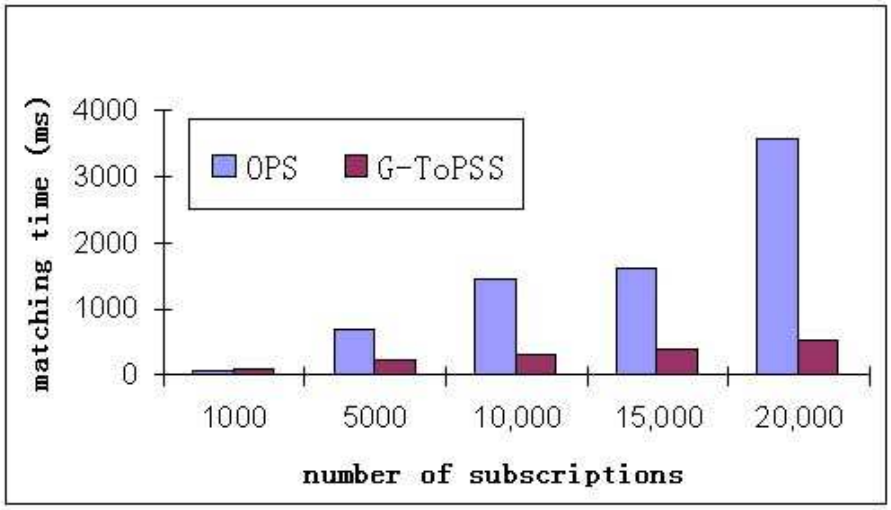


Figure 5.21: G-ToPSS vs. OPS

Figure 5.22 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.

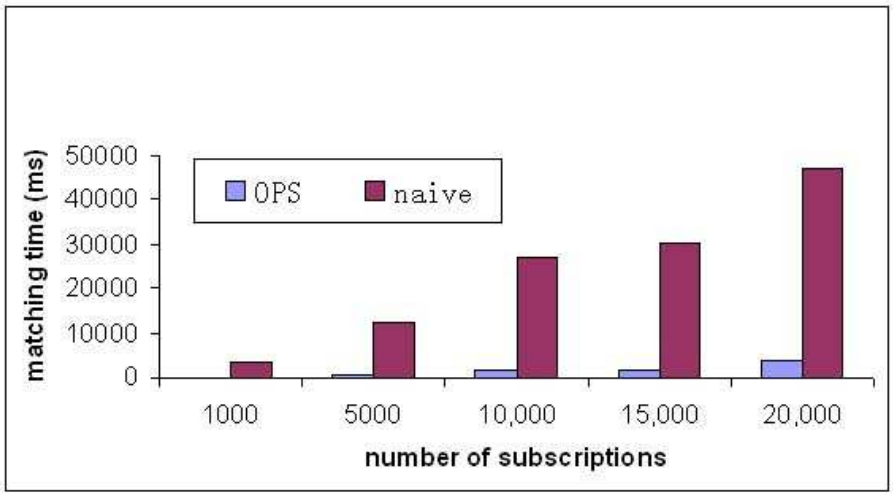


Figure 5.22: OPS vs. naive

5.4.3 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 5.23 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 [73]. Our algorithm is especially useful for the workload where most subscriptions have same

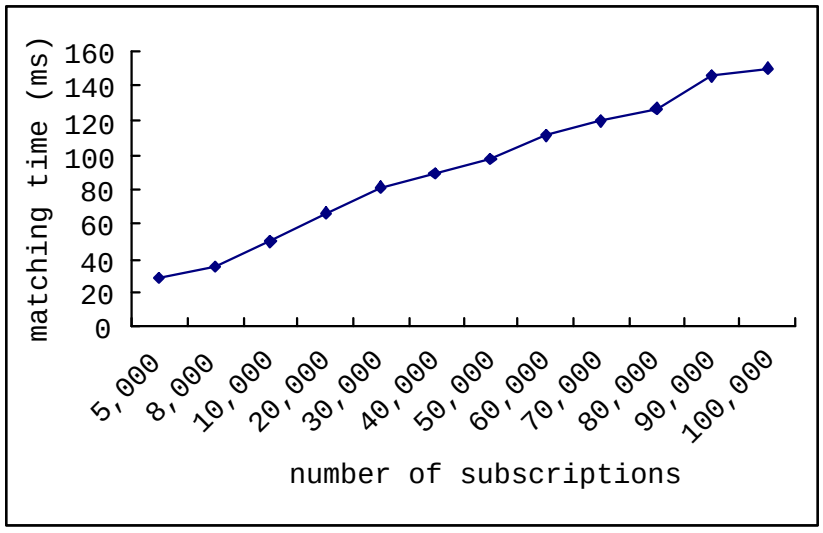


Figure 5.23: index lookup time vs. #subscriptions

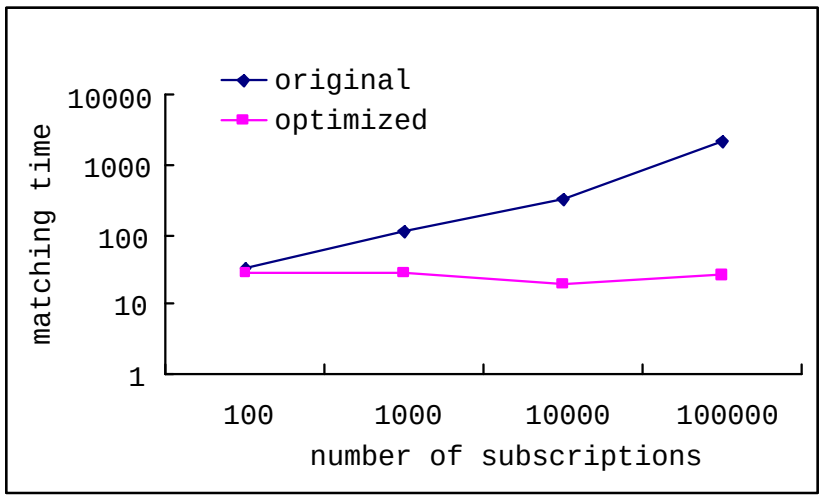


Figure 5.24: GToPSS vs. Optimization

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 5.24 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 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 5.25 shows that the subscription index size on the server is reduced to 50% by indexing.

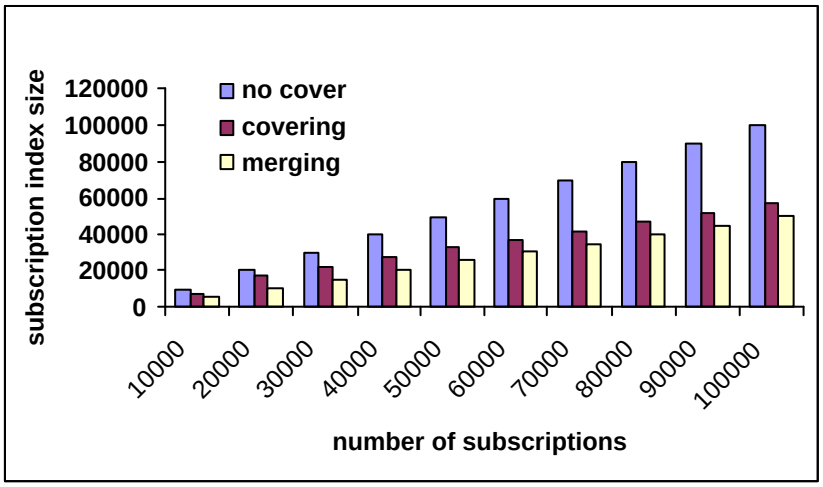


Figure 5.25: server index size vs. #subscriptions

Figure 5.26 shows the matching time when the number of matches in the workload varies. The matching time is dependent on the number of matches in the routing table. Without checking the covering relation, all the matched subscriptions will be read from the workload into the routing table. However, using the covering algorithm, some of the matched subscriptions will be dropped since they are covered by others. Thus, there are fewer matches in the routing table it takes less time to match. Merging is similar. but

the number of matches is reduced even further and matching is even faster.

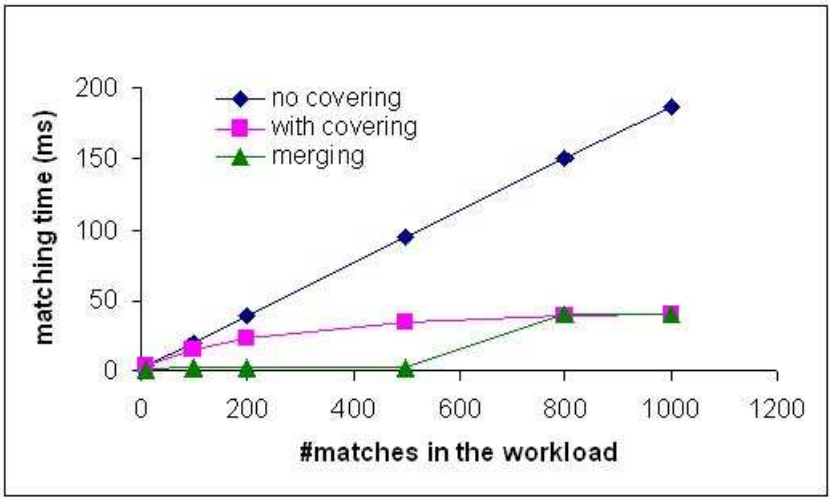


Figure 5.26: Matching time vs. #matches

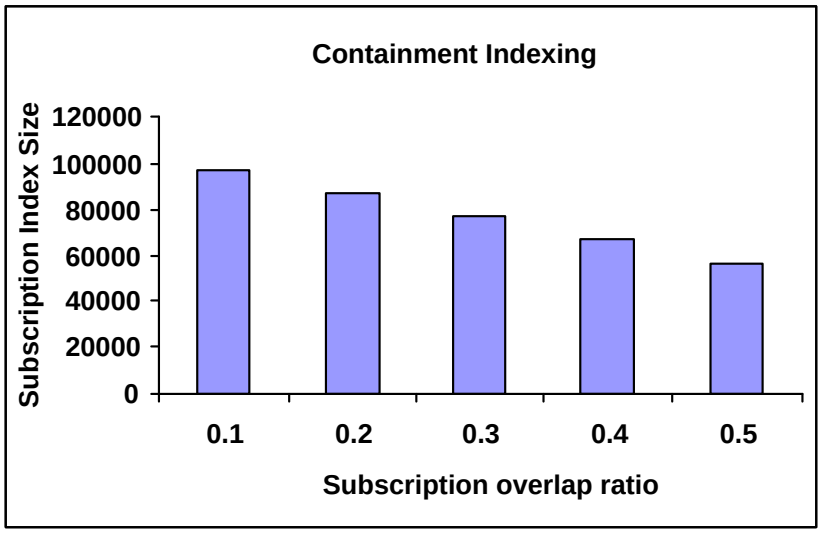


Figure 5.27: routing table size vs. overlapping ratio

Figure 5.27 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 clients; 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 5.28 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 $number\_of\_subscriptions * (1 - merge\_percentage)$. [2]

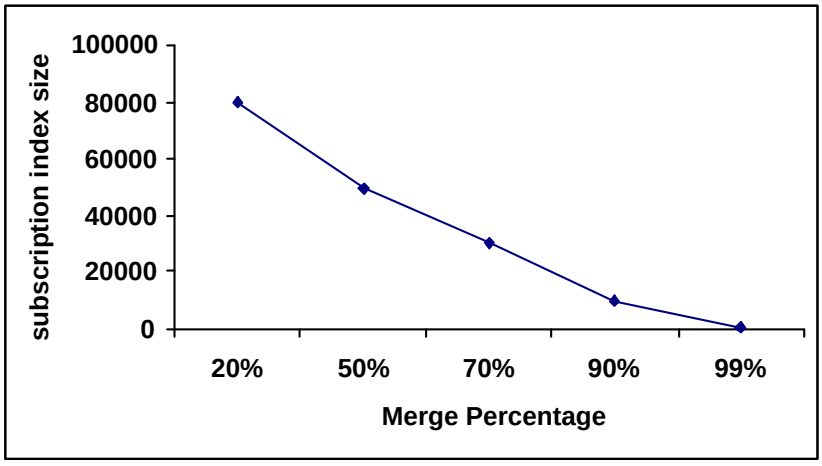


Figure 5.28: index size vs. merge percentage

The merging of subscriptions may result in the introduction of false positive (i.e., unmatched publications that are forwarded into client nodes, but do not actually match the individual subscriptions. They are not forwarded to real subscribers, since they are filtered out by the client 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 5.29 shows that there

[2]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.

are only 0.13% false positives for a 90% merge percentage.

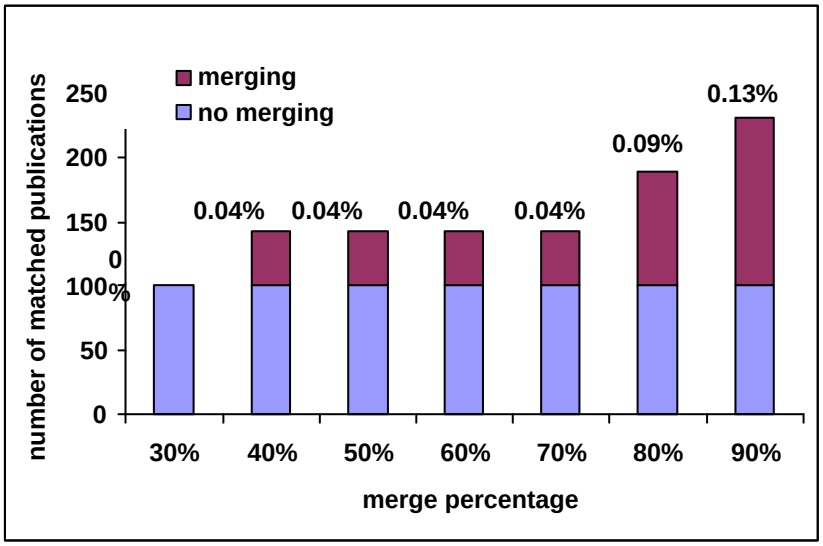


Figure 5.29: false positive vs. merge percentage

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 5.30, 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 5.3.3, 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 5.31 increases with the increase of merging percentage.

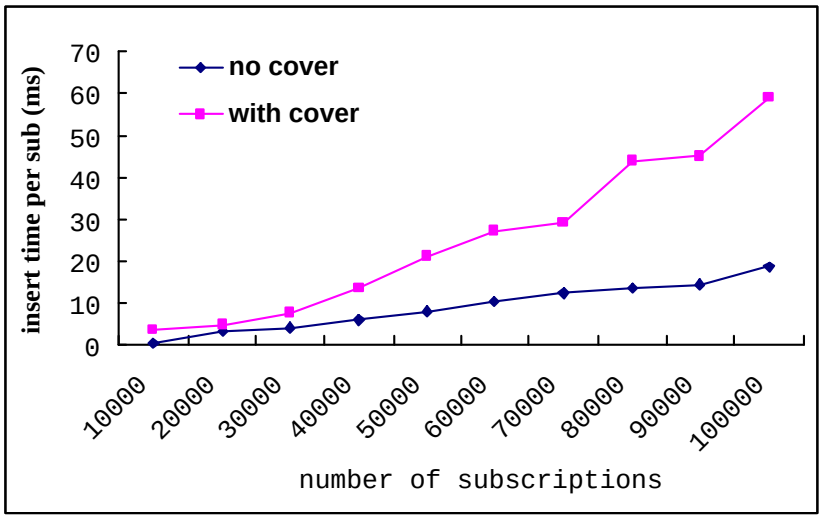


Figure 5.30: Insertion time vs. #subscriptions

5.4.4 Filtering Performance on a Cluster

The goal of using a cluster-based architecture for filtering is to increase the system throughput. The increased throughput is tradeoff by the cost of distributing (i.e., indexing) subscriptions among the cluster nodes. The following parameters are factors that influence the system throughput: number of nodes in the cluster, number of subscriptions, number of matches, subscription overlapping ratio and merge percentage. We examine the effect of these factors on both indexing cost and system throughput. In the experiments, we use a workload of 1 million subscriptions.

System throughput depends on the degree of filtering parallelism. The parallelism degree depends on the number of nodes in the cluster. In the following experiments, $cluster\_size$ equals to the number of back-end nodes. $cluster\_size$ equals 0 represents a centralized architecture, there is only one node which contains the entire workload. The indexing algorithms only work when $cluster\_size > 0$.

First we examine the performance of containment indexing. Figure 5.32 shows the effect of cluster size and number of total subscriptions on the average number of subscriptions stored in each node. The figure shows that the number of subscriptions per

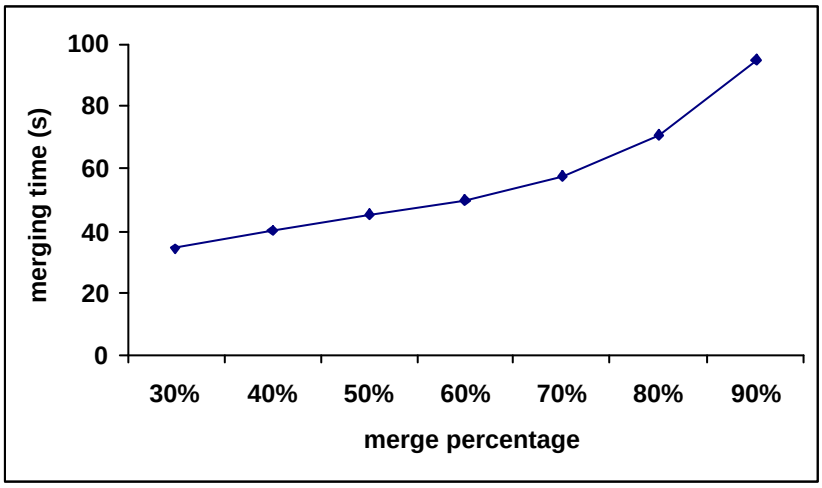


Figure 5.31: Merging time vs. merge percentage

node decreases proportionally with the increase of cluster size.

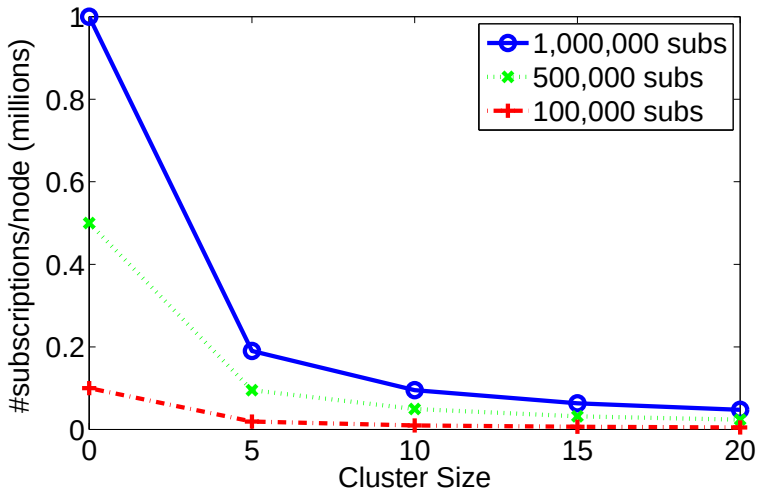


Figure 5.32: Number of Subscriptions per node vs.cluster size

Figure 5.33 shows the effect of overlapping ratio and cluster size on the subscription indexing time (or loading time in a centralized architecture). Obviously, in a centralized architecture, there is no containment checking, therefore, the loading time is the smallest. In a cluster architecture, the loading time decreases with the increased number of cluster node. The decreased indexing cost is due to the parallel processing. For a fixed cluster

size, the indexing time decreases with the increase of subscription overlapping ratio. With larger overlapping ratio, there are more overlapped subscriptions, and more commonality among subscriptions, thus less time needed for inserting a subscription.

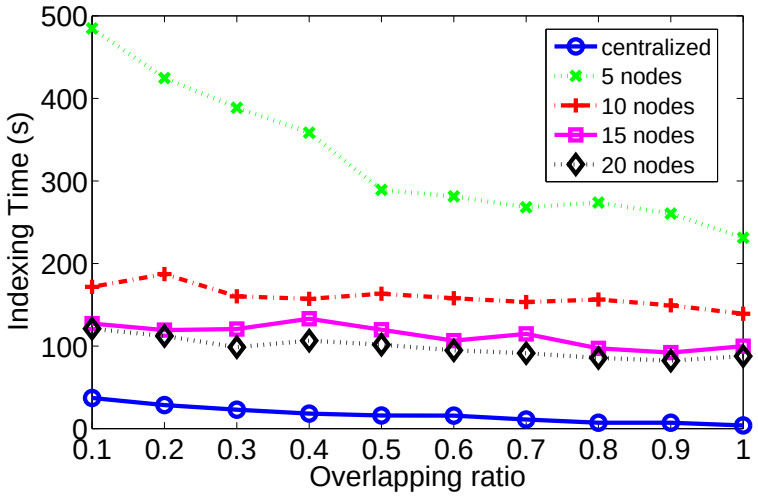


Figure 5.33: Indexing time vs.overlapping ratio

Next we examine the system throughput when using containment indexing in the cluster. In the experiments, we fixed the workload at $1,000,000$ subscriptions. We run the same workload on the clusters with different number of nodes and different percentage of matches per publication. The figure shows that for fixed percentage of matches, the system throughput increases linearly with the increase of cluster size. This is due to the parallel filtering in the cluster architecture. From [73], we know that the matching time on one node depends on the number of matches. Therefore, we can see that the system throughput decreases with the increase of matching percentage in figure 5.34.

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 5.35 shows the effect of the merge percentage and cluster size on system throughput for a workload with $1,000,000$ subscriptions and

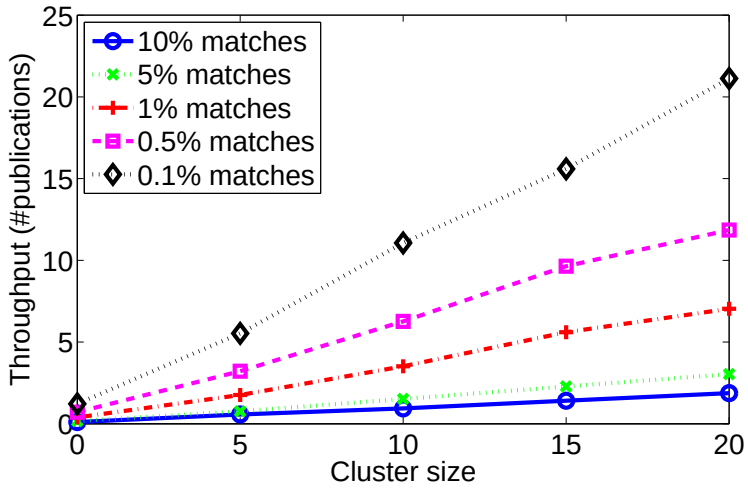


Figure 5.34: System throughput vs.cluster size for containment indexing

among which there are 1% matches. The figure validates the above argument: the benefit of the cluster filtering system increases with the increase of merge percentage and the cluster size. However, the increase speed slows down for larger merge percentage. The merge percentage is defined as the threshold to allow merging process, not the real percentage of subscriptions being merged. Although the merge percentage is set high, there may not have enough similar subscriptions to be merged. Therefore, the throughput increasing speed slows down for large merge percentage.

5.5 G-ToPSS System Implementation

5.5.1 Web Application

Recent years have seen a rise in the number of unconventional publishing tools on the Internet. Tools such as wikis, blogs, discussion forums, and web-based content management systems have experienced tremendous rise in popularity and use; primarily because they provide something traditional tools do not: easy of use for non computer-oriented users and they are based on the idea of "collaboration." It is estimated, by `pewinternet.org`,

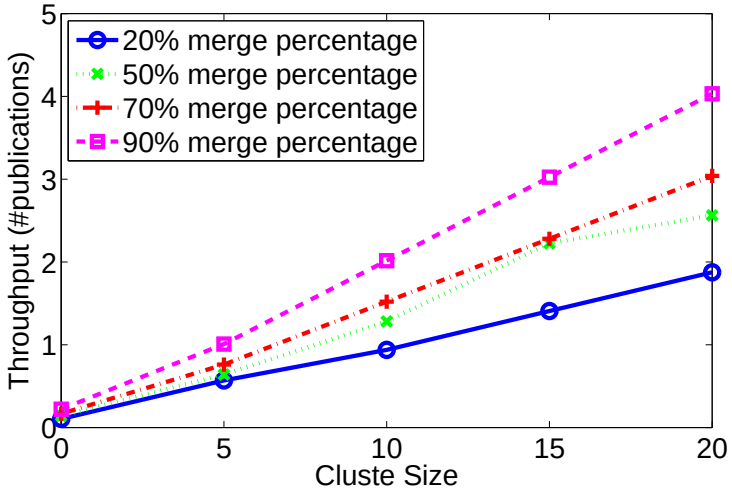


Figure 5.35: Throughput vs. cluster size for merging indexing

that 32 million people in the US read blogs (which represents 27% of the estimated 120 million US Internet users) while 8 million people have said that they have created blogs.

Web-based collaboration is the common idea for this new breed of content-management tools. The center piece of such tools is a web page that is being used as an area where multiple users participate in content creation. More significantly, the collaboration enabling tool used is the web page itself (accessed through the all-pervasive web browser).

With these new web applications, there rose a need for users to stay informed about changes to the content. In general, users want to be updated about daily news headlines of interest to them, or be notified when there is a reply in a discussion they participate in, or their favorite web personality has updated his/her blog (online diary etc.).

RSS[3] is quickly becoming the dominant way to disseminate content update notifications on the Internet. `pewinternet.org` reports that 6 million people in the US use RSS aggregators (a service/application that monitors large numbers of RSS feeds).[4]

Web-based content management systems (CMS) have also grown in popularity mainly

[3] `web.resource.org/rss/1.0/spec`

[4] Reported by Pew Internet & American Life Project (www.pewinternet.org), an organization that produces reports that explore the impact of the Internet on families, communities, the daily life. Also reported by "RSS at Harvard Law" (blogs.law.harvard.edu/tech/)

because they are based on the publishing tools just described, but also because they are much easier to use and maintain than traditional CMS.[5] Like traditional CMS systems, they provide content access control, user profiles, persistent storage, web access, RSS authoring, advanced content management, content routing and taxonomic content classification.

In this section, we describe an extension to content management systems, CMS-ToPSS, for scalable dissemination of RSS documents, based on the publish/subscribe model. To illustrate the effectiveness of the system, we extend an existing open-source web-based content-management system, Drupal (`drupal.org`) to use CMS-ToPSS in a manner that is transparent to end users, yet provides an efficient content-routing architecture.

5.5.2 System Architecture

CMS-ToPSS consists of three main components: The Drupal module (a content management system), the G-ToPSS filtering service and connector between them. The overall architecture is shown in Figure 5.36. The Drupal module acts as a client to the filtering service. The module does not require any changes to Drupal, and any Drupal installation can experience the benefits of CMS-ToPSS by simply retrieving and installing the module.

G-ToPSS filtering service is accessible via XML-RPC and can be accessed by the XML-RPC client. The CMS-ToPSS connector reads RSS feeds and serializes them into publications and subscriptions as input to G-ToPSS. In Figure 5.37 we show an example of an RSS feed (ie.e, a publication) and a subscription is shown in Figure 5.38. Both publications and subscriptions are RSS feeds. And subscriptions are differentiated by the key word *GQL* in *title* and the query can be taken out from *description*.

Upon receiving a publication and subscriptions, G-ToPSS performs the matching

[5]Mid Market Web CMS Vendors Pull Ahead. Brice Dunwoodie. `CMSwire.com`

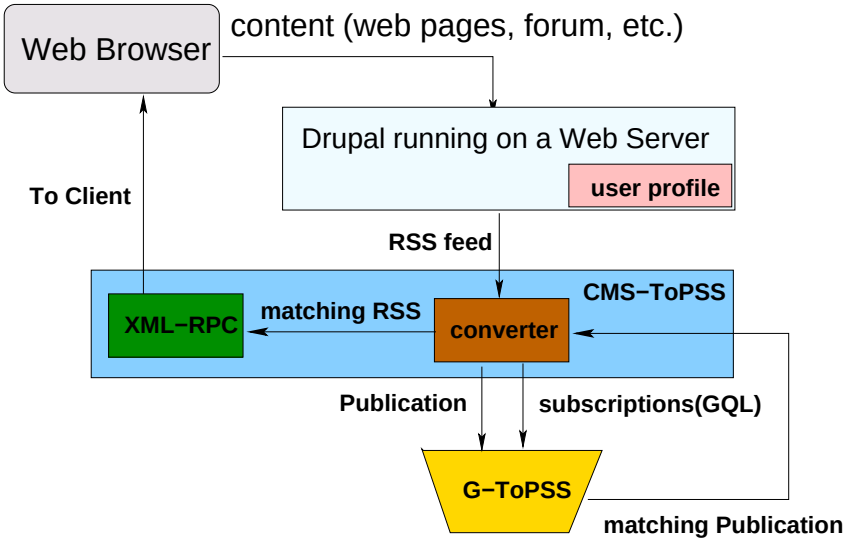


Figure 5.36: CMS-ToPSS system architecture

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://purl.org/rss/1.0/">
    <channel rdf:about="http://www.xml.com/xml/news.rss">
        <title>XML.com</title>
        <link>http://xml.com/pub</link>
        <description> XML.com features a rich mix of information and services for
the XML community. </description>
        <image rdf:resource="http://xml.com/universal/images/xml_tiny.gif" />
        <items>
            <rdf:Seq>
              <rdf:li resource="http://xml.com/pub/2000/08/09/xslt/xslt.html" />
              <rdf:li resource="http://xml.com/pub/2000/08/09/rdfdb/index.html" />
            </rdf:Seq>
        </items>
    </channel>
    <image rdf:about="http://xml.com/universal/images/xml_tiny.gif">
        <title>XML.com</title>
        <link>http://www.xml.com</link>
        <url>http://xml.com/universal/images/xml_tiny.gif</url>
    </image>
    <item rdf:about="http://xml.com/pub/2000/08/09/xslt/xslt.html">
        <title>Processing Inclusions with XSLT</title>
        <link>http://xml.com/pub/2000/08/09/xslt/xslt.html</link>
        <description> Processing document inclusions with general XML tools can
be problematic. This article proposes a way of preserving inclusion information
through SAX-based processing. </description>
    </item>
    <item rdf:about="http://xml.com/pub/2000/08/09/rdfdb/index.html">
        <title>Putting RDF to Work</title>
        <link>http://xml.com/pub/2000/08/09/rdfdb/index.html</link>
        <description> Tool and API support for the Resource Description Frame-
work is slowly coming of age. Edd Dumbill takes a look at RDFDB, one of the most
exciting new RDF toolkits. </description>
    </item>
</rdf:RDF>
```

Figure 5.37: RSS feed example

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://purl.org/rss/1.0/">
    <channel rdf:about="http://www.xml.com/xml/news.rss">
        <title>XML.com</title>
        <link>http://xml.com/pub</link>
        <description> XML.com features a rich mix of information and services for
the XML community. </description>
        <image rdf:resource="http://xml.com/universal/images/xml_tiny.gif"/>
        <items>
            <rdf:Seq>
              <rdf:li resource="http://xml.com/pub/2000/08/09/xslt/xslt.html"/>
              <rdf:li resource="http://xml.com/pub/2000/08/09/rdfdb/index.html"/>
            </rdf:Seq>
        </items>
    </channel>
    <item rdf:about="http://xml.com/pub/2000/08/09/rdfdb/index.html">
        <title>GQL</title>
        <link>http://xml.com/pub/2000/08/09/rdfdb/index.html</link>
        <description>
             SELECT ?x
             WHERE (Homepage325, Project, G-ToPSS), (G-ToPSS, Supervisor,
Arno Jacobsen), (G-ToPSS, YEAR, ?x)
             SUCHTHAT (?x > 2000) </description>
    </item>
</rdf:RDF>
```

Figure 5.38: Subscription example

between them and the outputs are notifications which are also serialized as RSS feeds over the converter back to the client via XML-RPC. Each subscription that a user submits is, in fact, a distinct RSS feed (containing items matching the user's subscription).

The Drupal module performs both subscribing and publishing based on user interaction with Drupal CMS. User can easily generate an RDF document using our template and publish to G-ToPSS. Also user can form a subscription with specified constraints from the interaction panel and send it to G-ToPSS. The module serializes all content changes in Drupal using RSS and sends them to the G-ToPSS filter service. The filtering service forwards the document to the interested clients which could be other XML-RPC clients as well as other Drupal modules. Note that the G-ToPSS filtering service can serve multiple Drupal sites.

In addition to publishing all content changes in RSS, the Drupal module also extends different kinds of Drupal content with change notification capabilities. For example, users can subscribe to receive notifications when they have replies on the discussion forum, or when a certain web page in Drupal has been updated. The Drupal module registers these kinds of subscriptions with the G-ToPSS filtering service transparently to the user.

A user, using a web browser, accesses a Drupal site that is extended with the module described in this chapter. The user can choose to receive notifications for content of her choice (e.g., discussion forum replies, web page updates etc.) Drupal supports convenient taxonomic content classification, which can be directly mapped to a G-ToPSS ontology. In this case, the user will get notifications only when both the content and taxonomic constraints of her subscription are satisfied. The users can also create content (e.g., participate in a discussion form or create/update a web page) to trigger notifications. The users' subscriptions are stored as part of their Drupal profile. Via the profile web page, users can review their notification requests as well as see all notifications received for those requests.

We also allow users to subscribe directly on the RSS content by expressing their

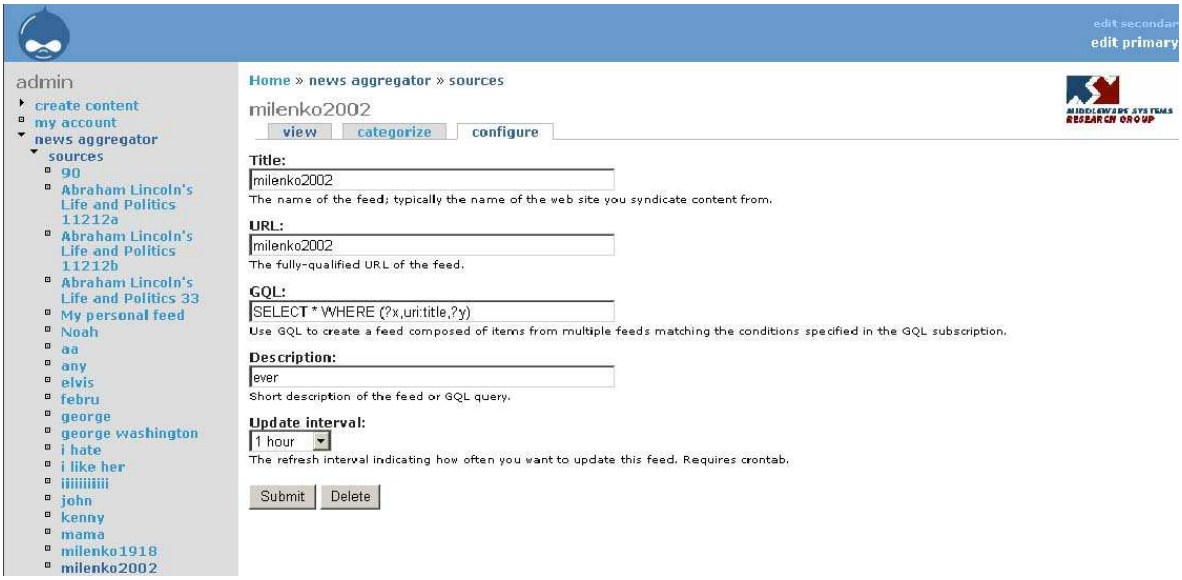


Figure 5.39: CMS-ToPSS User API

subscriptions in G-ToPSS's SQL-like subscription language (GQL). The subscriptions and their results are also shown as part of the user profile. The screenshot of the user interface is shown in Figure 5.39.

Chapter 6

Probabilistic Publish/Subscribe System

Efficient Pattern matching over event streams is increasing being employed in many areas and has aroused significant interest in industry [84, 24]. Much effort has been devoted to developing efficient matching algorithms [96, 2]. Up to now, all the systems developed to date perform a fully evaluation for each subscription pattern. The match result will not be reported until the whole subscription has been evaluated. Especially in the domain of complex event processing, action waits to be taken until all subscribed events have occured, there is no response at intermediate state.

Many applications of complex event processing are monitoring systems such as flight alert system, intrusion detection,credit card fraud detection, traffic monitoring system. It would be more flexible for these systems to take action if a match of the defined monitoring pattern could be predicted in advance rather than wait until the situation becomes worse. Take the intrusion detection system as an example, if a series of suspicious activities is predicted as a potential intrusion with a large probability, the system would responds by resetting the connection or by reprogramming the firewall to block network traffic from the suspected malicious source.

To the best of our knowledge, there is no research prototype has the feature to predict the occurrence of complex event associated with a probability based on the partial match status. In this thesis we will propose a model by leveraging the Markov Model to offer this ability.

In this chapter, Section 6.1 describes a probabilistic publish/subscribe model for complex event processing supporting expressive operators. Section 6.3 develops the data structures and the algorithms constitute our matching and predicting functionality. Finally, the system architecture is presented in Section 4.1.

6.1 System Model

In this section we describe the language and data model underlying our probabilistic publish/subscribe approach. The objectives are to allow subscribers to express interests in complex constellations of events over event streams allowing the subscriber to join individual events through Boolean operators and constrain interests via explicit and implicit temporal conditions. The language is the basis for algorithms to probabilistically match subscriber interests and to predict matching results from partially detected events.

6.1.1 Publication Data Model

Publications describe real-world events and are defined by a set of attribute value pairs:

$$e_i = \{(a_1, v_1) \cdots (a_n, v_n)\}.$$

Unlike conventional publish/subscribe approaches, the model in this chapter operates over event streams, where an *event stream* is interpreted as infinite sequence of events, and each event represents an occurrence of interest at a designated point in time. As is common in the publish/subscribe literature, we are using the terms *event* and *publication* synonymously. Each event contains an attribute that records when it occurred. Events are assumed to be processed in the order they occurred.

For example, Table 6.1 shows the events logged by a typical operating system monitoring facility. Whenever a login attempt is registered by the system, the event is recorded with a time stamp, a status message, and the IP address from where the attempt was made. We add an event ID, as a unique identifier for each event.

Table 6.1: Event stream of login history

EID	Time	Status	IP
e_0	2007/02/14/12:38:10	denied	128.100.2.15
e_1	2007/02/14/12:42:10	denied	128.100.2.15
e_2	2007/02/14/12:42:10	denied	128.100.2.15
e_3	2007/02/14/12:43:28	success	128.100.2.15
e_4	2007/02/14/12:43:56	logoff	128.100.2.15
e_5	2007/02/14/12:45:28	success	128.100.5.10

6.1.2 Subscription Language

A subscription expresses the interest of a subscriber in events. A *primitive subscription* is a subscription that is matched by a single event. In contrast, a *composite subscription* is a subscription that may require multiple independent events to be matched.

A primitive subscription is a conjunction of predicates. Each predicate defines a constraint over an attribute. More formally, a primitive subscription is as follows:

$$s = p_1 \wedge p_2 \wedge \cdots \wedge p_k,$$

where the p_i are Boolean predicates.

For example, a primitive subscription that monitors a failed login from a specific computer is expressed as follows:

$$s_{failed\_login} : (LOGIN = denied) \wedge (IP = 128.100.2.15).$$

A primitive subscription can be matched by multiple events. In the above example, s_{failed_login} is matched by e_1 and e_2. In contrast, a composite subscription expresses interest in *composite events*. Here, composite events represent constellations of events in the event stream, but not necessarily contiguous. Composite subscriptions correlate events from the event stream through temporal and logical operators. More formally, a composite subscription is defined as an expression over primitive subscriptions that are related by temporal and Boolean operators:

$$cs = R(s_1, s_2, \cdots, s_k),$$

where R denotes operators listed in Table 6.2.

Table 6.2: Composite subscription operators

Temporal		Boolean	
R	Description	R	Description
,	contiguous sequence	$\wedge$	conjunction
;	non-contiguous sequence	$\vee$	disjunction
@	explicit temporal operator		

The contiguous sequence operator, ",", requires its operand subscriptions to be matched by contiguous events in the event stream. The non-contiguous sequence operator, ";", is similar, but allows other events to occur between the events in the event stream matching the operand subscriptions, as long as the matched events occur in order. Since events in the event stream occur in order and, we assume, no events occur at the same time, there is an implicit temporal condition associated with both "," and ";" operators. That is the time interval between the events matching the operand subscriptions has to be greater than zero. One event cannot match two operands of the same composite subscription. In contrast, "@" is an explicit temporal operator; it adds an explicit temporal condition that requires the matching events to occur in the specified time interval.

In the intrusion detection scenario, from above, a possible intrusion can be modeled by a composite subscription as follows:

$$\begin{aligned} s_1, s_2, s_3: &\quad IP = \$x \wedge LOGIN = denied \\ s_4: &\quad IP = \$x \wedge LOGIN = success \\ cs_{intrusion}: &\quad s_1; s_2; s_3@(t(s_3) - t(s_1) < 5min), s_4 \end{aligned}$$

$cs_{intrusion}$ defines a potential intrusion as the occurrence of at least three failed login attempts followed by one successful login attempted, where the interval between the first failed attempt and the last attempt must be less than 5 minutes and all logins occur from the same IP address. The variable, x, in the IP predicate represents the join condition that all login attempts originate form the same IP address. The non-contiguous sequence operator used, allows for other events to occur in between.

The following shows a more general example.

$$\begin{aligned} s_1, s_2: &\quad IP = \$x \wedge LOGIN = denied \\ s_3, s_4: &\quad IP = \$x \wedge LOGIN = success \\ s_5: &\quad IP = \$x \wedge ACTION = passwd \\ s_6: &\quad IP = \$x \wedge ACTION = logoff \\ cs_{compromised} = &\quad s_1; ((s_2; s_3@(t(s_3) - t(s_1) < d)) \vee (s_4, s_5)); s_6 \end{aligned}$$

$cs_{compromised}$ defines an intrusion pattern bracketed by a failed login attempt at the start and a logoff action at the end; in between these events, either there are at least one failed login followed by one successful login, or one successful login followed by a password reset action. This composite subscription illustrates the combination of both temporal and Boolean operators.

6.1.3 The Matching and Prediction Problems

In this section we define the matching problem and the prediction problem for the probabilistic publish/subscribe model.

The matching problem is similar to the standard publish/subscribe matching problem: *Given a set of composite subscriptions CS and an event stream $\{e_i\}$, find all $cs \in CS$ such that for each s_i of cs there is an e_j that matches s_i and the occurrence time of all matching e_j satisfies the temporal conditions specified in cs.*

For example, $cs_{intrusion}$ is matched by the event sequence e_0, e_1, e_2, e_3 from Table 6.1.

Next, we describe the prediction problem. At some time, t_{now}, events in the event stream may have matched some of the primitive subscriptions registered with the system. A composite subscription for which some of its primitive subscriptions are matched is said to be in a *partially matched* state. This is the case, for example, when s_1 and s_2 in $cs_{intrusion}$ are matched.

Our objective is to be able to predict that the probability a subscription, cs, will match is greater than a threshold, θ_{cs}, after processing N further events.

Say, based on past observations, we know that an intrusion occured half of the time after the first failed login and 80% of the time after two failed logins. Based on this, we conclude that the composite subscription, $cs_{instrusion}$, is matched with a probability of 0.5 after observing the first login failure and with a probability of 0.8 after the second failure.

That is given an event stream, we aim to calculate the probability that a composite subscription, cs, is matched some time in the future given its current partial matching state. The challenge is to efficiently calculate this conditional probability for all composite subscriptions. Thus, the prediction problem is as follows: *Given a set of composite subscriptions, CS, and an event stream, $\{e_i\}$, find all partially matched composite subscriptions $cs \in CS$ such that the probability of cs transitioning from a partial match to a full match, after processing N events, is greater than the threshold, θ_{cs}.*

6.2 Subscription Processing

In this section we describe the system architecture and subscription processing.

The probabilistic publish/subscribe system performs four principal matching tasks. These are the matching of primitive subscriptions, the matching of sequence and temporal operators, the matching of Boolean expressions underlying composite subscriptions, and the prediction of subscription matches based on partial matching state evolving over time. Figure 6.1 shows the matching engine architecture and hints at the processing flow of events through the system.

Before discussing the event processing flow in more detail, we describe how composite subscriptions are decomposed and represented in the system. A composite subscription is an expression tree. Intermediate tree nodes represent the operators and leaf nodes the expressions of primitive subscriptions.

The Boolean expressions defining the composite subscription are represented as Boolean trees and are managed by the *Boolean Matching Engine* (BME) in the architecture. The temporal subexpressions of composite subscriptions are represented as finite state machines and are managed by the *State Machine Engine* (SME). Individual primitive subscriptions, which are at the leaves of the tree, are managed by the *Atomic Subscription Matcher* (ASM). The decomposition of a composite subscription into finite state machines and Boolean trees is described in Sec. 6.2.4.

Input events are first evaluated against all primitive subscriptions stored in the SM. The resulting matches drive the state transitions of the SME and the matching of the Boolean operators of composite subscriptions in the BME. SME and BME produce three kinds of outputs. The first kind are the fully matched composite subscriptions. The second kind are referred to as *derived events*. These are events that are fed back from the SME to the BME and from the BME to the SME to trigger further state transitions and Boolean matches, respectively. The third kind are partial matches passed on to the *Prediction Engine* (PE).

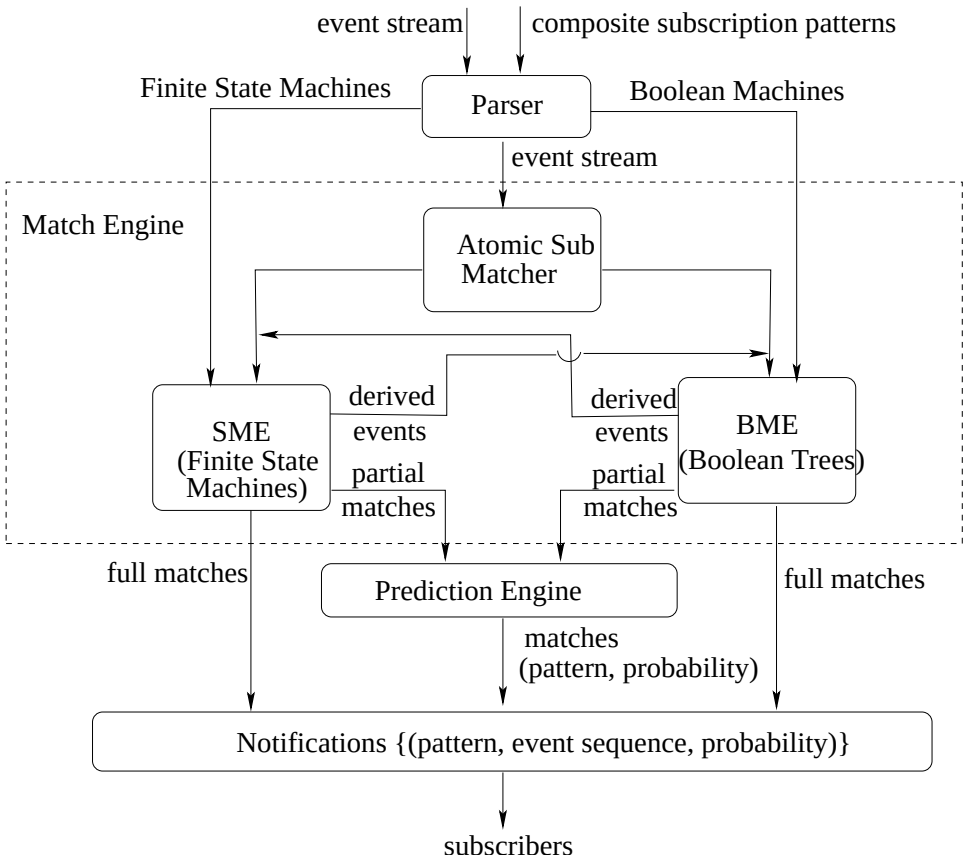


Figure 6.1: System Architecture

Primitive subscription matching and Boolean expression matching has been extensively studied [53, 23, 34] and will not be further discussed here. In our implementation we experiment with a Binary Decision Diagram (BDD)-based matcher for evaluating primitive subscriptions [54] and employ a variant of the Rete algorithm [53] for the stateful Boolean expression matching required by the BME. The BME needs to track partially matched composite subscriptions.

Here, we focus our discussion on the algorithms underlying the State Machine Engine (SME), which leverages finite state machines (FSM) [45] for the processing of sequence operators. Subscription expressions are graphs that represent finite state machines derived from the sequence operators. In the graph a node represents a state, an edge represents a state transition labelled by a primitive subscription whose match triggers

the transition. State transitions are triggerd by events.

Traditional algorithms for converting regular expression into FSMs can not be used due to the explicit temporal operator, @. Below, we describe the FSM construction and event processing algorithms for our composite subscription expressions. There are five cases to consider, as detailed in Table 6.3.

Table 6.3: FSM Construction Cases

Case Description	**Op.**	**Algo.**
Contiguous op w/ explicit time condition Contiguous op w/o time condition	, , @	1
Non-contiguous op w/o time condition	;	2
Non-contiguous op w/ time condition	; @	3
General	, ; @	4

6.2.1 Contiguous Sequence Operators

Constructing an FSM and processing it for expressions using the contiguous sequence operator is simple. It constitutes a building block for supporting the other operators. In our presentation, we assume the processing of one expression, deferring optimizations for the merging and joint processing of multiple FSMs to below.

Given an expression consisting of k primitive subscriptions $cs = s_1, s_2, \cdots, s_k$, the FSM has $k+1$ states capturing the intermediate states of matching the individual primitive subscriptions. N_0 is the initial state (no matches), N_1 represents the state that s_1 matches, N_2 represents the state that the previous two contiguous events match s_1 and s_2. N_{k-1} represents the state that the previous k contiguous events match $s_1, s_2 \cdots s_{k-1}$, and the last N_k represents the state that the whole expression is matched.

The key to constructing the FSM is the addition of all necessary transitions among

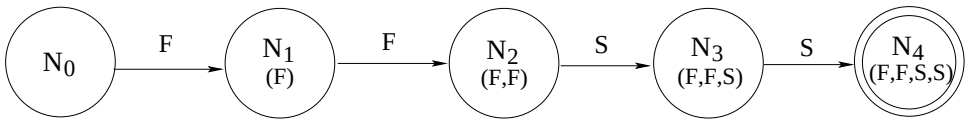


Figure 6.2: FSM for F, F, S, S

states so that the FSM can always match the largest partial state given any input. For each state $N_i, i = [0, k-1]$, we add a transition from state N_i to state N_{i+1} upon the match of the primitive subscription s_{i+1}.

During matching, the initial state is always checked for each new event to see whether a new partially matching instance of the FSM must be initiated. Therefore, if nothing matches the incoming event, all current FSM instances are simply discarded. The detail of the algorithm is described in *Build_State_Machine_For_Term*.

Algorithm *Build_State_Machine_For_Term*$(cs = s_1, s_2, \cdots, s_k)$

($*$ construct a state transition graph given a composite subscription containing only "," operators $*$)

1. construct $k+1$ states $N_0, N_1 = s_1$, $N_2 = s_1, s_2$, $\cdots$, $N_{k-1} = s_1, s_2, \cdots, s_{k-1}$, $N_k = s_1, s_2, \cdots, s_k$
2. **for** each state $N_i = s_1, s_2, \cdots, s_i (i = 0, \cdots k-1)$
3. add an edge s_{i+1} from N_i to N_{i+1}

Taking the expression $cs = F, F, S, S$ as example, there are five states N_0, N_1, N_2, N_3, and N_4. For example, state N_2 represents that the sub-expression F, F matches. If a successful login S event occurs, we move to the next state N_3. Figure 6.2 shows this example.

We treat the temporal condition defined by @ as an additional predicate of the associated primitive subscription. The above algorithm can be applied to build the FSM for expressions containing @ operators. Figure 6.3 shows an example for the expression $F, F, F@(t_{N_3} - t_{N_1} < d), S$.

To support the explicit temporal operator in event processing, each state is associated with an attribute that records the most recent transition into the state. When the time condition is evaluated for determining whether to take the transition, the value of this

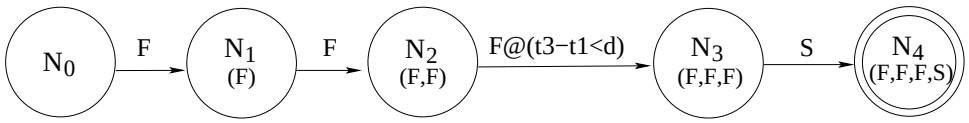


Figure 6.3: FSM for $F, F, F@(t_{N_3} - t_{N_1} < d), S$

attribute is used.

When a new instance of a state machine is created, separate time attribute instances are initialized as well. For example, given a state machine as shown in Figure 6.3, there would be three instances of the state machine if events matching $F_{@1}, F_{@3}, F_{@4}$ occurred. In one instance, the current state is N_1 with a time attribute value of 4; in another instance, the current state is N_2 and the previous state N_1 has a time attribute value of 3. In the last instance, the current state is N_3 and the previous state N_1 has a time attribute value of 1.

6.2.2 Non-contiguous Sequence Operators

There are two differences for the FSM that represents expressions with non-contiguous sequence operators. First, since events not contributing to matching an expression are allowed to occur during matching, the FSM must remains at the current state, even if the primitive subscription that triggers the transition to the next state is not matched. Second, if the next primitive subscription is matched, we need to take two transitions in order to track all matching possibilities. One transition leads to the next state, the other transition leads back to the state itself to allow future matches of the next primitive subscription to trigger the transition to the next state. Figure 6.4 shows an example and Algorithm *Transition_Between_Term* describes the detail of the graph construction. In the figure we use the "$\star$" symbol to represent any input.

Algorithm *Transition_Between_Term*($cs = s_1; s_2; \cdots; s_k$)

($*$ construct a state transition graph for composite subscription with only ";" operators $*$)

1. construct $k+1$ states $N_0, N_1 = s_1$, $N_2 = s_1; s_2$, $\cdots$, $N_{k-1} = s_1; s_2; \cdots; s_{k-1}$, $N_k = s_1; s_2; \cdots; s_k$

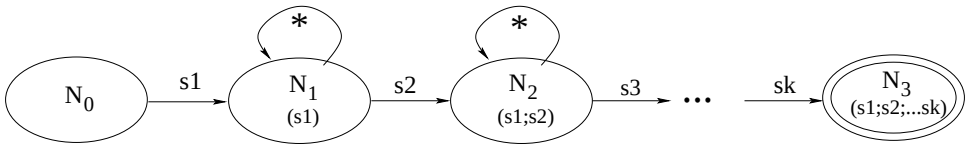


Figure 6.4: FSM for $s_1; s_2; \cdots; s_k$

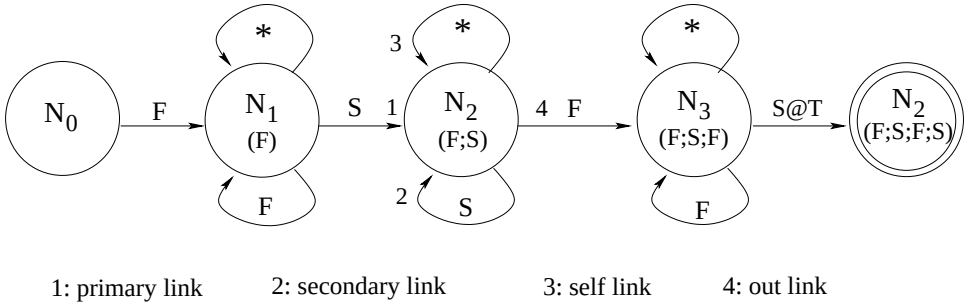


Figure 6.5: FSM for $F; S_1; F; S_2@(t_{S_2} - t_{S_1} < T)$

2. **for** each state $N_i = s_1; s_2; \cdots; s_i$
3. add an edge s_{i+1} from N_i to N_{i+1}
4. add an edge $\star$ from N_i to N_i

In order to support the explicit temporal operator "@", we need to record the matching time of the primitive subscriptions that are referred to by a time condition. We associate a time attribute with each state. Thus, we need to differentiate the transitions by which the state is reached. Taking Figure 6.5 as an example, there are three incoming links to state N_2. From state N_1, there is a link to state N_2 upon the match of S. This is the first time that N_2 is reached upon a match, and we call this link the *primary* link. From state N_2, there are two links back to itself. One is triggered also upon a match of S, and we call this link the *secondary link*. Since the non-contiguous sequence operator allows other events to occur in between, the transition will stay in the current state. This link is called a *self* link and and is triggered on all events except those that cause a transition of the primary or secondary links. Except the *self* links, both of the other links (*primary, secondary*) are triggered upon a match. The transition times should be recorded with these transitions for future evaluation of the associated time conditions.

6.2.3 General Sequence Operators

When the contiguous sequence operators and non-contiguous sequence operators are used together in one composite subscription, we decompose the composite subscription into multiple *terms* separated by the non-contiguous sequence operators and each *term* contains only contiguous sequence operators. For each term, we add transitions using Algorithm *Build_State_Machine_For_Term*, and we apply Algorithm *Transition_Between_Term* to add transitions between terms. Algorithm *Build_State_Machine* describes the detailed process.

Algorithm *Build_State_Machine*($cs = R(s_1, s_2, \cdots, s_m)$)

($*$ construct a finite state machine for a composite subscription containing ", ;" and "@" operators $*$)

1. Decompose the whole composite subscription into n terms that are separated ed by non-contiguous sequence operator, $cs = T_1; T_2; \cdots; T_n$; each term $T_i = S_{i1}, S_{i2}, \cdots, S_{ik_i}$ contains only contiguous sequence operator
2. $m = k_1 + k_2 + \cdots + k_n$
3. construct m states $N_0, N_1 = s_1, \cdots,$
4. $N_m = s_{11}, s_{12}, \cdots, s_{1k_1}; \cdots; s_{n1}, s_{n2}, \cdots, s_{nk_n}$, where each state represents that partial match state of one more primitive subscription
5. **for** each term $T_i = s_{i1}, s_{i2}, \cdots, s_{ik_i}$
6. *Build_Transition_For_Term*($T_i = s_{i1}, s_{i2}, \cdots, s_{ik_i}$)
7. **for** each state $N_j = \cdots; s_{i1}, s_{i2}, \cdots, s_{ik_i} (= s_j)$
8. add an edge s_{j+1} from N_j to N_{j+1}
9. add an edge s_j from N_j to N_j
10. add an edge $\star$ from N_j to N_j

Take Figure 6.6 as an example. This composite subscription is composed of three terms: (F, F, S), (F, S) and (F, S). There is a time condition between the F event in the last term and first F event in the first term. To construct the state transition graph for this composite subscription, we first create a chain of states where each state indicates incremental match status of one more match of a primitive subscription than the previous state. And for each term, we add transitions according to the rule for the contiguous

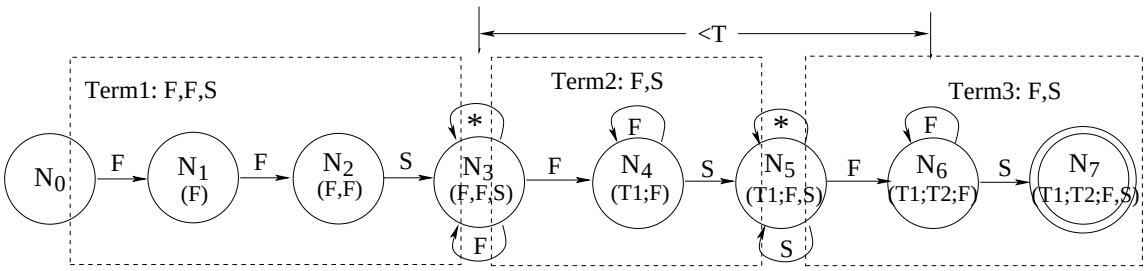


Figure 6.6: FSM for $F, F_1, S; F, S; F_2@(t_{F_2} - t_{F_1} < T), S$

operators. Then we add transitions between the states across different terms according to the rules for non-contiguous operators.

6.2.4 Combining Boolean/Sequence Operators

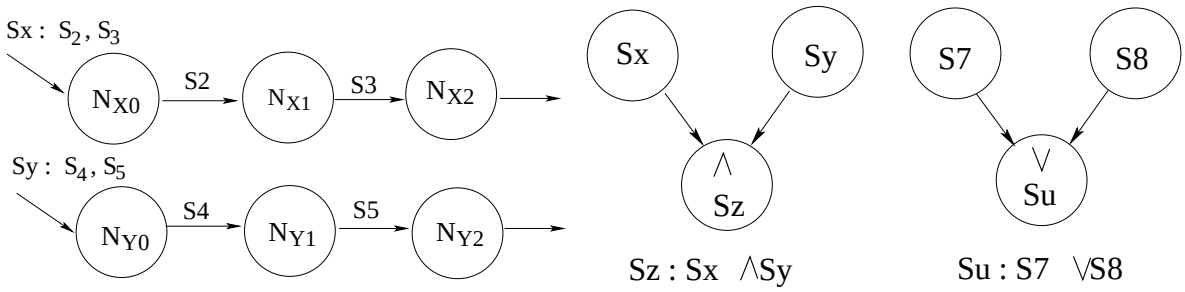
Given a general composite subscription that contains both sequence operators and Boolean operators, it will be decomposed using Algorithm *Build_Machines* to generate a set of state machines and Boolean machines. The decomposed state or Boolean machines may hierarchically compose another one. Then we call those decomposed machines as *child* machines, and the machine containing the decomposed ones as *parent* machines. The machine which appears at the top level represents the original whole composite subscription pattern, and we call it the *master* machine. The whole composite subscription is matched when the master machine reaches the matched state.

Algorithm *Build_Machines*($pattern = s_1 s_2 \cdots s_n$)

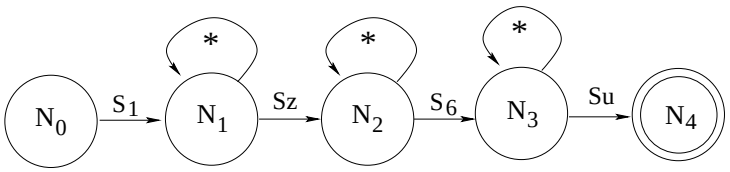
Input: a general composite subscription which is a String with n symbols, the symbol might be subscription id, sequence operator or Boolean operator

Output: a set of State Machines and Boolean Machines

1. symbols = new stack()
2. Machines = $\emptyset$
3. **for** each symbol σ in *pattern*
4. **if** $\sigma \neq')'$
5. symbols.push(σ)
6. **else**



(a) State machines and Boolean machines



(b) The Master State Machine for $cs = s_1; ((s_2, s_3) \wedge (s_4, s_5)); s_6; (s_7 \vee s_8)$

Figure 6.7: General composite subscription

```
7.          str = null
8.          σ = symbols.pop()
9.          while σ ≠' ('
10.                 str = σ + str
11.         if str.get(1).isBoolean()
12.             M = Build_Boolean_Machine(str)
13.         else  M = Build_State_Machine(str)
14.         Machines.add(M)
15.         new_symbol = M.reference
16.         symbols.push(new_symbol)
17. return Machines
```

Parentheses may be used to explicitly specify the operator precedence in a pattern. For example, the composite subscription $cs = s_1; ((s_2, s_3) \wedge (s_4, s_5)); s_6; (s_7 \vee s_8)$ can be represented by three state machines and one Boolean machines as shown in Figure 6.7.

When a composite subscription contains multiple state machines and Boolean machines, a special trigger event is created in order to trigger the transition between the state and Boolean matching engines. We define two types of trigger events. One is called *in trigger* which is a event sent by a parent state machine to start a child state machine. The other is called *out trigger* which is sent by a child state machine to a parent machine notifying a match. Once a child state machine is triggered, it remains active in order to detect more matches. Matches in the child state machine generate derived events that are consumed by the master state machine. The child state machine only terminates when the master state machine terminiates, i.e., the whole composite subscription is matched or times out.

6.2.5 Merging Multiple Graphs

Each composite subscription is represented by one or more state machines and Boolean machines. Machines that share common subexpressions can be optimized by merging their commonalities into a single machine. This helps save memory and improve the matching efficiency. Merging Boolean expressions has been studied and is not the focus of this thesis. Here we describe how to merge multiple state machines.

The principle of merging multiple state machines is to maintain the correctness of the merged state machine. Concretely speaking, given two state machines, two states N_1 in cs_1 and N_2 in cs_2 are chosen to be merged if at any time cs_1 arrives the state N_1, cs_2 arrives the state N_2. States N_1 and N_2 are said to be equivalent.

Definition: Two states N_1 and N_2 are *equivalent* if the following conditions are satisfied:
(1) The number of incoming transitions of N_1 and N_2 are equal.
(2) For each incoming transition, there exists an equivalent mapping between N_1 and N_2, that is for $\forall I_1 \rightarrow^x N_1, \exists I_2 \rightarrow^x N_2$ and I_1 and I_2 are equivalent.

Figure 6.8 shows the result of merging three state machines $cs_1 = a; b, c$, $cs_2 = a; b, d$ and $cs_3 = a$. State M_1 and M_2 are merge states representing the partial matches for both

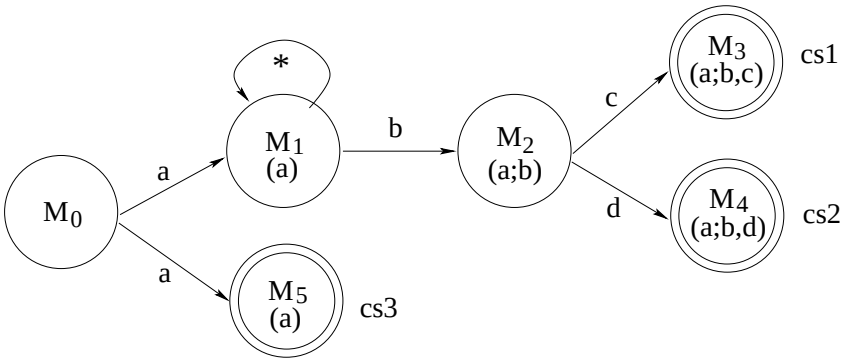


Figure 6.8: Merging Multiple State Machines

cs_1 and cs_2. However, state M_1 and M_3 cannot be merged although both of them will be triggered upon an occurrence of event a. M_1 and M_3 are not equivalent since there is another incoming transition (the $\star$ edge) associated with state M_1.

6.3 Event Processing

This section will describe the event matching and prediction algorithms.

6.3.1 Matching Algorithm

In our model, all composite subscriptions are represented as state machines and Boolean machines. Each primitive subscription match triggers a transition. The main challenge in the composite subscription evaluation is the management of time information for each matched primitive subscription and the efficient evaluation of the associated time conditions.

The discussion below considers the case where the explicit temporal operator references states within a finite state machine, and one where it references states across different state machines.

Time conditions within a finite state machine

Each time condition involves two primitive subscriptions. We refer to the first matched primitive subscription as the *referencing subscription*, and the latter matched primitive subscription as the *dependent subscription*. In the state machine, the state whose arrival is triggered by the referencing subscription is called the *referencing state* and the state whose arrival is triggered by the dependent subscription is called the *dependent state*. For example, suppose there is a time condition $@(t_{s_2} - t_{s_1}) < d$ between primitive subscriptions s_1 and s_2. s_1 is the referencing subscription and s_2 is the dependent subscription. During matching, all the primary and secondary transitions into the referencing state should be recorded so that the time condition can be evaluated later when the dependent state is reached.

In order to record the incoming transition time for individual states, we associate a time list $T_m(s_i)$ for each referencing state referred to by a time condition. Each entry in the list is a time recording when the state is reached (or when the primitive subscription triggering the arrival of this state is matched). New entries will be inserted into the time list when a *primary* transition or a *secondary* transition is taken. Since each state can be referred to by multiple time conditions and each time entry may satisfy a different set of time conditions, we associate a *time compatible Set* $T_c(s_i)$ with each entry containing the set of time conditions satisfied by this entry.

Figure 6.9 shows an example of the data structure we described above to support explicit temporal operators. There are three time conditions in the example involving three subscriptions s_1, s_2 and s_3 (also three states N_1, N_2, N_3). We create a time list for each state. Assume that the incoming events matched the subscriptions in a time sequence as shown at the bottom of Figure 6.9.

The first three times will be inserted into time list $T_m(s_1)$ as they indicate three matches of s_1. At time 4, s_2 is matched. Before moving to the state N_2, time condition T_1 is evaluated based on the match time of s_2 ($t_{s_2} = 4$) and the entries in $T_m(s_1)$.

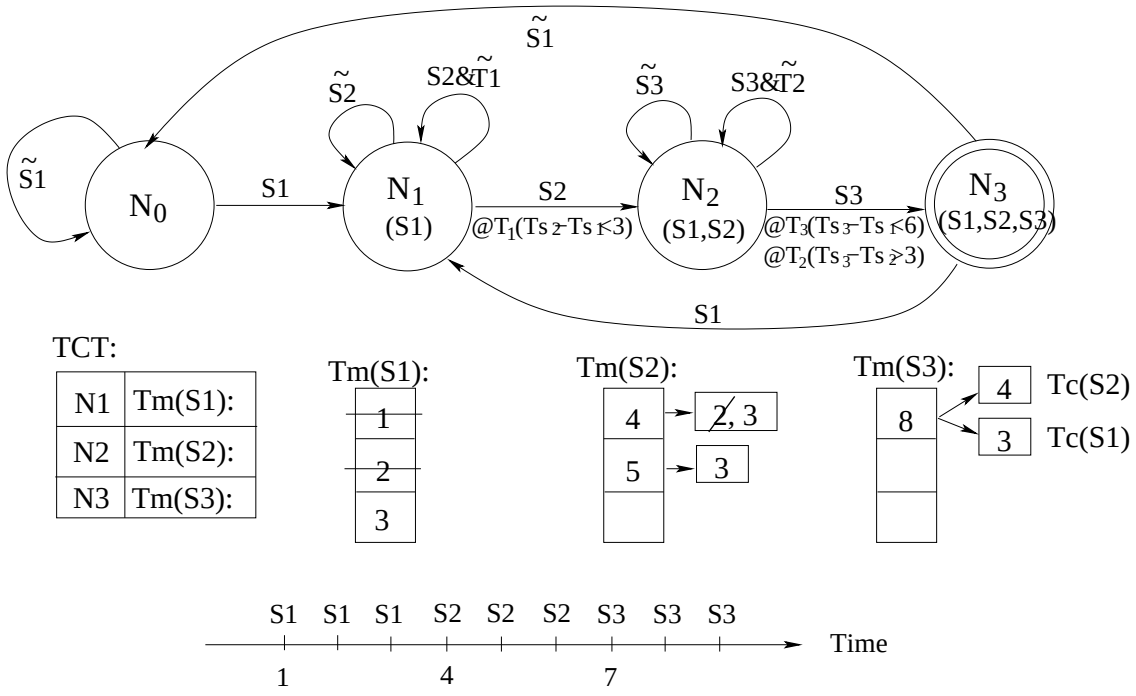


Figure 6.9: Data structure for subscription $cs = s_1; s_2@(t_{s_2} - t_{s_1} < 3); s_3@(t_{s_3} - t_{s_1} < 6)@(t_{s_3} - t_{s_2} > 3)$

Only time 2 and 3 in the list satisfies T_1, so time 1 will be deleted from $T_m(s_1)$. Time 4 is inserted into $T_m(s_2)$ and at the same time time 2 and 3 are inserted into $T_c(s_1)$. Similarly, at time 5, s_2 is matched again, time 5 is inserted into $T_m(s_2)$ and time 3 is inserted into $T_c(s_1)$. At time 6, s_2 is matched again. However, since it doesn't satisfy the time condition T_1 based on the time entries in $T_m(s_1)$, it is discarded. For the same reason, at time 7, s_3 is matched. But it is discarded since the time condition T_3 is not satisfied for 7.

At time 8, subscription s_3 is matched again, we evaluate the time conditions T_2 and T_3 associated with this subscription. T_2 is satisfied for the match time 4 of s_2 and T_3 is satisfied for the match time 3 of s_1. Thus, time 2 will be deleted from the $T_m(s_1)$ and $T_c(s_1)$ lists. Similarly time 5 will be deleted from $T_m(s_2)$. At this point, the time lists and compatible lists will be updated as shown in the figure. According to these entries in the lists, we know that the composite subscription is matched by the pattern

$(s_1(@3); s_2(@4); s_3(@8))$.

The time lists and compatible lists will be updated along with the matches of subscriptions (or transitions in the state machines). We describe the detailed process in Algorithm *Match*.

Algorithm $Match(FSM, e)$

($*$ Given a finite state machine FSM and an event e update the current state and time lists $*$)

1. $N_n = \delta(N_c, e, t)$ //Find the next state N_n based on current state N_c, event input e, current time t and transition function δ
2. **if** $\exists$ time condition $@T$ refered by state N_n
3. insert time t into $T_m(s_n)$
4. **for** each time condition $@T$ dependant on state N_n
5. $N_i = Get\_Ref\_State(@T)$
6. List $l_m = N_i.get\_time\_list()$
7. Set matched_times = $eval(@T, l_m, t)$
8. **if** !matched_times.isEmpty()
9. entry = $T_m(s_n).insert(t)$
10. entry.set(N_i, matched_times)
11. **if** $N_n \neq N_c$
12. prune(N_c), $N_c = N_n$
13. **if** N_c is matched state
14. find the final matched time if no time list is empty
15. return N_c.getMatchedPattern()

Algorithm $Prune(N_c)$

($*$ Recursively update the time lists when leaving a state $*$)

1. List $l_c = N_c.get\_time\_list()$
2. **for** each time condition $@T$ associated with the incoming transition to N_c
3. $N_i = Get\_Ref\_State(@T)$
4. List $l_i = N_i.get\_time\_list()$
5. $U = \bigcup_k l_c[k].get(N_i)$
6. l_i.Update(U)
7. Prune(N_i)

8. **for** each time condition @T associated with the outgoing transition from N_c
9. $N_i = Get\_Dependent\_State(@T)$
10. List $l_i = N_i.get\_time\_list()$
11. **for** each $l_i[k].T_c(s_c)$
12. $l_i[k].T_c(s_c)$.update(l_i)
13. Prune(N_i)

Upon an incoming event e, we find the primitive matched subscriptions and the next state N_n based on the current state N_c. If there is no time condition associated with this transition, we just move to the next state. Otherwise, we insert the time into the time list of the next state if it is referred by a time condition (N_n is the referencing state of a time condition). If there are time conditions dependent on N_n (N_n is the dependent state), we update its compatible sets. When leaving the current state, we recursively prune all time lists and compatible sets to find out the final matches that satisfy all time conditions. If N_n is a final matched state, we get the associated composite subscriptions and report the result.

Time conditions across finite state machines

In general, a composite subscription contains both temporal operators and Boolean operators, in which case, it consists of multiple state machines and Boolean machines. Notice that a primitive subscription can also be considered as a state machine. Thus, all time conditions appear among state machines. According to the locations where the dependent state and referencing state of a time condition appear, we classify the time condition evaluation procedure for a general composite subscription into five cases as listed in Table 6.4. The graphically representation of the example composite subscription in Table 6.4 is shown in Figure 6.7.

In Table 6.4, SM_x represents state machine x, $BM(SM_x)$ represents the Boolean machine where a reference to state machine x appears. In case 1, the time condition is defined between two subscriptions in the same state machine. In cases 2 and 3, the

Table 6.4: Time condition $@(s_i, s_j)$ for example subscription $cs = s_1; ((s_2; s_3) \wedge (s_4, s_5)); s_6; (s_7 \vee s_8)$.

Ref. State Location ($SM(s_i)$)	Dep. State Location ($SM(s_j)$)	Example
$SM_x (= S_X)$	SM_x	$@(s_2, s_3)$
master	$SM_y (= S_Y)$	$@(s_1, s_2)$
SM_x	master	$@(s_4, s_6)$
$SM_x \neq SM_y \neq$ master, $BM(SM_x) = BM(SM_y) (= S_Z)$		$@(s_3, s_4)$
$SM_x \neq SM_y \neq$ master, $BM(SM_x) (= S_Z) \neq BM(SM_y) (= S_U)$		$@(s_3, s_7)$

time condition is defined across the child state machine and the master state machine. In cases 4 and 5, the time condition is across two different child state machines that are referred to by the same Boolean machine (case 4) or by different Boolean machines (case 5).

To evaluate time conditions across different state machines, we build a global time condition table TCT (as shown in Figure 6.9) containing the mapping between referencing/dependent states and the corresponding time lists. When the referencing state is reached (e.g., s_i is matched), a new entry will be inserted into its time list in TCT. When the dependent state is going to be reached (e.g., s_j is matched), the associated time condition will be evaluated and the time list and its compatible list is updated accordingly. If s_j is not referred by any latter time condition, the time list of s_j will be cleaned up. This procedure will be applied for cases 1,2,3 and 5 as listed in Table 6.4.

For case 4, when the time condition is between two states located in different state machines, but the two state machines belongs to the same Boolean machine, there is no order constraint of these two primitive subscriptions. In this case, the time condition is treated as an absolute time difference, of the form $|t_{s_i} - t_{s_j}| < T$. For this type of time

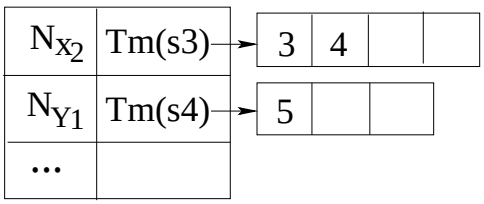


Figure 6.10: Global Time Condition Table Example

condition, we can't fix either state to be the dependent state at which the time condition will be evaluated. Instead, we have to evaluate the condition $-T \leq t_{s_i} - t_{s_j} < T$ after both s_i and s_j are matched. To save the evaluation time, we insert new entries into the time lists upon the match of s_i and s_j, but we don't evaluate the condition, create the compatible list or do pruning until we transit into the first common descendent node of the two nodes in the Boolean tree referring to the state machines where s_i and s_j appears.

Take the composite subscription shown in Figure 6.7 as an example. The composite subscription is defined as $s_1; ((s_2; s_3) \wedge (s_4; s_5)); s_6$ and the time condition is defined as $@(|t_{s_3} - t_{s_4}| < T_1)$. The state machines and Boolean machines are represented in Figure 6.7. To enable the time condition evaluation, we build a global time condition table as shown in Figure 6.10. There are two entries: one records the matching time for s_3 which is updated upon entering node N_{X_2}; the other records the matching time for s_4 which is updated upon entering node N_{Y_1}. s_1 appears in the state machine SM_X and s_2 appears in the state machine SM_Y. The time condition won't be evaluated until the first common descendent of SM_X and SM_Y, SM_Z in this case, is reached, which means that both s_3 and s_4 are matched and we can start evaluating the absolute time condition.

Notice that absolute time condition is not possible for case 5 even though the time condition is also across two non-master state machines. Since SM_X and SM_Y do not belong to one Boolean machine, there must be a sequence operator between SM_X and SM_Y, and thus, an absolute time condition is not possible.

6.3.2 Prediction Algorithm

Each finite state machine records incremental matches of a pattern. Each successive state in a finite machine represents a further partial match status towards the final full match. If the machine is in state N_c currently, then the probability that it moves to state N_n upon the occurrence of next event depends only on the present state. In other words, the description of the present state fully captures all the information that could influence the future evolution of the process. Therefore, a finite state machine in our model can be considered as a Markov chain.

We propose a mechanism to leverage the properties of Markov chains to predict the probability of a future match of a composite subscription based on the current state and event history. First we explain how to assign the transition probabilities for a Markov chain (i.e., a state machine) based on the event history. Then we introduce a conditional probability for a single state machine to reach the matched state given the current state. Finally, we define a probability of future match for a general composite subscription represented by multiple state machines and Boolean machines.

Markov Chain Model Training

In order to calculate the probability of reaching future states, we need to determine the long-run transition probability for the finite state machine, i.e., the Markov chain.

With this objective, we set a counter for each transition in a state machine. Along with the incoming event stream, these counters are updated when the state machine takes a transition from the present state to another state. If the occurrence of the events follows a certain probability distribution, the behavior of the Markov Chain would exhibit a certain probability distribution.

Definition: Given a finite state machine, which is represented as a digraph $G = (V, E)$. $e_{ij} \in E$ is an edge from node v_i to node v_j, where $v_i \in V, v_j \in V$ are the states. The

transition probability from state i to state j is defined as

$$p_{ij} = \frac{N_{e_{ij}}}{\sum_k N_{e_{ki}}}$$

where $N_{e_{ij}}$ is the number of times that the transition from state i to state j has been taken and $\sum_k N_{e_{ki}}$ is the total number of times that all incoming transitions have been taken, which is also the number of times we have arrived at state i.

Given the transition probabilities between the states, the state machine can be considered as a complete Markov chain model. Since in our model, the state space is finite, the transition probability distribution can be represented by a matrix P, called the *transition matrix*, with the (i, j)th element of P equal to

$$p_{ij} = \Pr(X_{n+1} = j \mid X_n = i).$$

Prediction for Simple Composite Subscription

When we call the prediction engine to evaluate the probability of a match, we assume the Markov chain is a time-homogeneous Markov chain, so that the transition matrix P always remains the same at each step. The following probabilities can be computed:

1. Given the current state of a state machine, what is the probability of reaching the final matched state over the next n events. This probability is referred as *n-step probability*. We have P, which tells us what happens over one step, then we can work out what happens over n steps by $P^{(n)} = P^n$. Let α_i be the probability of reaching the final matching state (state M, which represents a match of the whole pattern) from the current state (state i) over the next n events is the (i, M)th element of $P^{(n)}$:

$$\alpha_i^{(n)} = P_{iM}^{(n)}.$$

2. Given the current state of a state machine, what is the probability of reaching the final matched state *within* the next n events. *Within* n steps means the match

can occur over any k steps with the condition $k \leq n$. Let β_i be the probability of reaching the final matched state (state M) from state i within the next n events is:

$$\beta_i^{(n)} = \sum_{k \leq n} \alpha_i^{(k)} = \sum_{k \leq n} P_{iM}^{(k)}.$$

Prediction for General Composite Subscription

When a composite subscriptions consists of both temporal operators and Boolean operators, it is represented by multiple state machines and Boolean machines. We define a global state to represent the status of such composite subscriptions.

Definition:A *global state* G is a group of sub-states. If we use M to represent the master state machine, $C_1, \cdots, C_k$ to represent the set of child state machines, the global state can be represented as follows:

$$G_i = \{M.N_{i_M}, C_1.N_{i_{c1}}, C_2.N_{i_{c2}}, \cdots, C_k.N_{i_{ck}}\}.$$

Definition: The prediction problem for a general subscription with multiple state and Boolean machines is defined as follows: given the current global state G_i, find the probability of reaching fully matched state G_m in n steps: $Pr^{(n)}(G_m|G_i)$, where $G_i = \{M.N_{i_M}, C_1.N_{i_{c1}}, C_2.N_{i_{c2}}, \cdots, C_k.N_{i_{ck}}\}$ and $G_m = \{M.N_{m_M}, C_1.N_{m_{c1}}, \cdots, C_k.N_{m_{ck}}\}$.

When G_i or G_m consist of active states of child state machines, individual conditional probabilities can be computed for each state machine separately, then these probabilities will be combined according to their relationship in the Boolean machine. Multiple state machines can be considered as independent events. The "and" Boolean operator require the occurrence of two events, and the "or" operators requires the occurrence of at least one of two events. Based on probability theory, we give the definition of the conditional probability for a simple composite subscription which contains only two child state machines that are combined by one Boolean operator. In this case, there are two states in the master state machine.

Given a composite subscription $s_G = s_{C_1} \wedge s_{C_2}$ where s_{C_1} and s_{C_2} contain only sequence operators. C_1 and C_2 are two child state machines. Suppose the current global state is

$G_i = \{C_1.N_{i1}, C_2.N_{i2}\}$, and the matching state would be $G_m = \{C_1.N_{F1}, C_2.N_{F2}\}$. Then the probability of reaching the final matched state in n steps would be

$$Pr^n_{\wedge}(G_m|G_i) = Pr^n_{C_1}(N_{F1}|N_{i1}) * Pr^n_{C_2}(N_{F2}|N_{i2}).$$

If $s_G = s_{C_1} \vee s_{C_2}$, the probability would be as follows:

$$Pr^n_{\vee}(G_m|G_i) = Pr^n_{C_1}(N_{F1}|N_{i1}) + Pr^n_{C_2}(N_{F2}|N_{i2}).$$

For a complex composite subscription which contains multiple child state machines and Boolean machines, we decompose the transition process into three steps when computing the conditional probability. The decomposition is performed on the master state machine M. The first step is to move from the current state $M.N_{i_M}$ to the next state $M.N_{i_M+1}$. The second step is to move from $M.N_{i_M+1}$ to the state just before the global state, $M.N_{j_M-1}$. The third step is to move from $M.N_{j_M-1}$ to the global state $M.N_{j_M}$.

$$\begin{aligned} & Pr^n(G_j|G_i) \\ = \; & \Sigma_{p+q+r=n} Pr^p(G_i \rightarrow M.N_{i_M+1}) \cdot Pr^q(M.N_{i_M+1} \rightarrow M.N_{j_M-1}) \cdot Pr^r(M.N_{j_M-1} \rightarrow G_j) \end{aligned}$$

The first and last function will be expanded according to the combination functions if G_i or G_j contains active child state machines. The second function can be computed based only on the master state machine since it doesn't involve any active child state machine. Taking Figure 6.7 as an example, suppose currently s_2 and s_4 are matched. The probability that cs is fully matched in n steps would be

$$\begin{aligned} & Pr^n(G_m|G_i) \\ = \; & Pr^n(M.N_4|X.N_{X_1}, Y.N_{Y_1}) \\ = \; & \Sigma_{p+q=n} Pr^p(M.N_2|X.N_{X_1}, Y.N_{Y_1}) \cdot Pr^q(M.N_4|M.N_2) \\ = \; & \Sigma_{p+q=n} (Pr^p(X.N_{X_2}|X.N_{X_1}) \cdot Pr^p(Y.N_{Y_2}|Y.N_{Y_1})) \cdot Pr^q(M.N_4|M.N_2) \end{aligned}$$

6.4 Experiments

In this section we experimentally evaluate our approach. All the algorithms are implemented in Java, and the experiments are run on a 3GHz Linux machine with 4GB of

RAM. We are using two workloads for experimentation: a synthetic load that lets us independently examine various aspects of our approach by running controlled experiments; and a second real-world data set to demonstrate the behavior of our approach under realistic conditions. Our workloads comprise composite subscriptions and events.

Table 6.5: Ptopss Workload parameters

Parameters	**Description**	**Default values**
$Length_{cs}$	Length of each composite subscription	10
$Num_{non\_contiguous\_op}$	Number of non-contiguous operators	2
Num_{cs}	Number of composite subscriptions	1,000
$Num_{full\_matches}$	Number of fully matches per subscription	20
$Num_{partial\_matche}$	Number of partial matches per subscription	20
$Size_{event\_pool}$	Size of pool where workload is generated	50

The parameters that characterize the workload generation are summarized in Table 6.5. First, we generate $N_{cs} = 1000$ composite subscriptions, each with length $Length_{cs} = 10$. The length represents the number of primitive subscriptions in the composite subscription. To generate primitive subscriptions, we randomly draw an event from a pool of predetermined events, and generate the subscription to either match or not match the event. By default the event pool contains 50 events.

The event pool is also used to generate the event stream. Based on the probability of drawing events from the pool, we generate two composite subscription workloads: (1) a *uniform_cs* workload, where each event is selected from the event pool based on a uniform distribution (i.e., each event has the same probability to be selected.); and (2) a *gaussian_cs* workload, where the probability for each event to be selected follows a Gaussian distribution. Each sequence of primitive subscriptions has $N_{non\_contiguous\_op} = 2$ non-contiguous operators that are randomly distributed among a total of $Length_{cs} - 1$

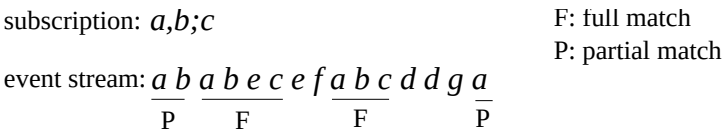


Figure 6.11: Sample event stream

operators.

The generation of event streams is controlled by varying the sequences of events that lead to full matches versus partial matches. The parameters are, $Num_{full\_matches} = 20$, the number of full matches and, $Num_{partial\_match} = 20$, the number of partial matches. Sequences of events giving rise to full versus partial matches are generated in a random order. They are also interspersed with a number of irrelevant events that do not match primitive subscriptions. Figure 6.11 shows an example of an event stream.

We generate two kinds of event stream workloads: (1) *uniform_pub* and (2) *gaussian_pub*, according to the probability distribution that determines the length of each fully or partially matching event sequence. In the *uniform_pub* workload, the length of each event sequence is uniformly distributed between 1 and $Lenth_{cs}$. In the *gaussian_pub* workload the length is drawn from a Gaussian distribution.

All measurements are performed after the system loaded all subscriptions. For evaluating the matching algorithm, we look at the effect of the number of composite subscriptions, their length, the number of non-contiguous operators and the size of the event pool. For the prediction algorithm, we look at the ratio of the number of full matches to partial matches, the number of prediction steps, and the prediction threshold. For each experiment, we vary one parameter and fix others to their default values as specified in Table 6.5.

6.4.1 Matching Performance

Number of subscriptions: Figure 6.12 shows the number of states with increasing number of subscriptions. We see that the number of states grows linearly as the number

of subscriptions increase. Furthermore, the rate of increase for small event pool size is less than that of the larger event pool size. This is because there are more merged common states with a smaller event pool size. Similarly, for the *gaussian_cs* workload, a small number of events are selected to generate the subscription, and hence there are even more common states than the *uniform_cs* workload.

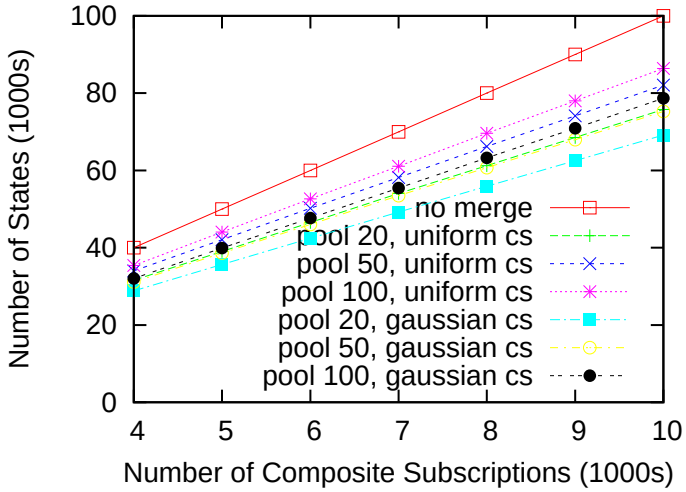


Figure 6.12: #States vs. #subscriptions

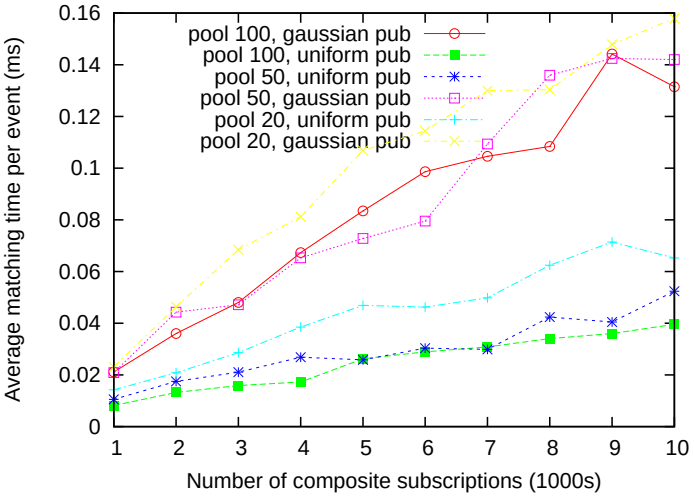


Figure 6.13: Matching time vs. #subscriptions

Figure 6.13 shows the average time to process one event given a fixed set of subscriptions. As the subscription size size increases, so does the time. Unlike the number of

states, given a fixed number of subscriptions, the matching time is larger for a smaller-sized event pool or *gaussian_cs* subscriptions where the workload share more common states. This is because with a larger number of common states, one event may trigger more transitions, thus requiring processing time.

Number of non-contiguous operators: Figure 6.14 shows that as the number of non-contiguous operators increases, so does the matching time. This is because more subscription instances remain partially matched, waiting for events to trigger their transitions.

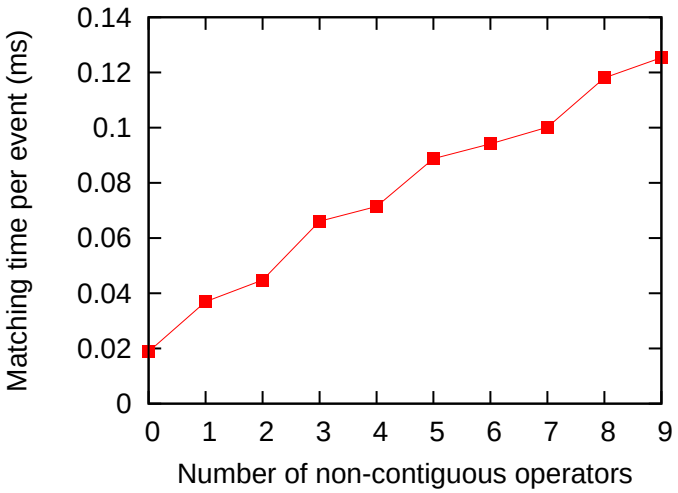


Figure 6.14: Matching time vs. #operators

Composite subscription length: Figure 6.15 shows the effect of increasing the length of composite subscriptions on merging. We plot the commonality, which is represented by the ratio between the number of shared states with merging and the total number of states without merging. We see that the commonality decreases with increasing subscription length. In our FSM model, each state in the FSM is defined by the prefix of the subscription. For a longer subscription, there are much more combinations to select events to generate a subscription compared to a shorter subscription. This explains why the commonality degree decreases with the increase of the subscription length. This experiment also shows that the merging process favors shorter subscriptions.

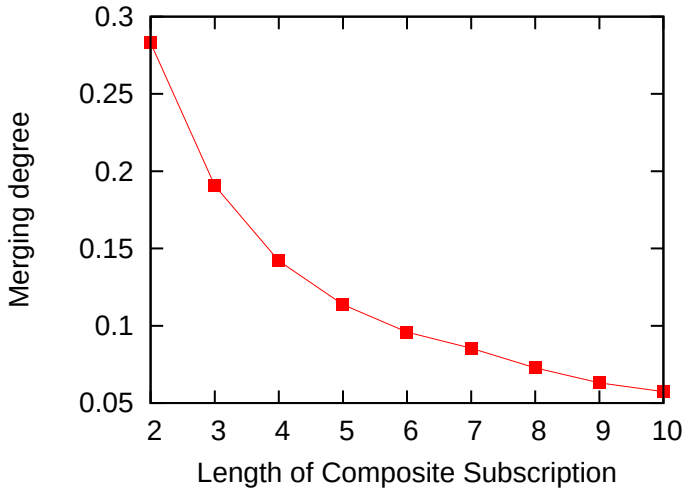


Figure 6.15: Merging degree vs. sub length

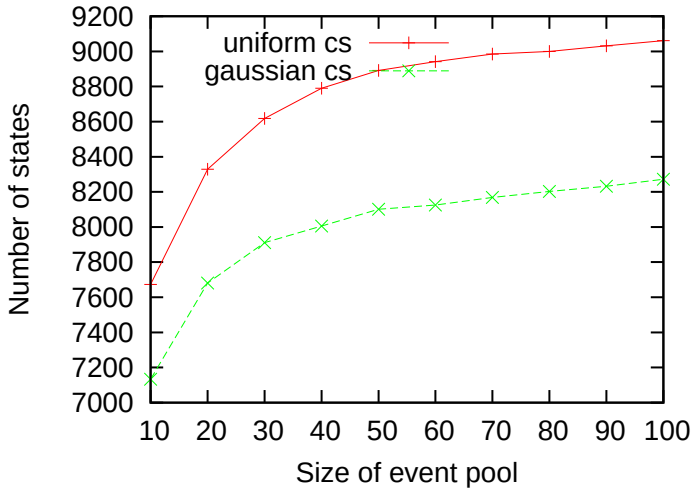


Figure 6.16: #States vs. pool_size

Size of event pool: The remaining figures show the effect of the event pool size on merging and matching. Figure 6.16 shows the number of states increasing with the pool size for the *uniform_cs* and *gaussian_cs* workloads. As the size of the event pool increases, the number of states for both workloads increases. This is because the number of shared common states among subscriptions decreases with increasing event pool size. However, for the *gaussian_cs* workload, a small number of events in the pool is selected to generate the subscription, therefore, the *gaussian_cs* workload has less states than the

uniform_cs workload.

Figure 6.17 shows that the number of transitions decreases with the increasing event pool size for different workloads. As we mentioned before, the number of shared common states among subscriptions decreases with increasing event pool size. Hence there are fewer instances during the matching process and so fewer transitions as well. The publication workload generated by the Gaussian distribution contains shorter partial matches than the workload generated by the uniform distribution, which results in fewer transitions. Thus, the number of transitions in the Gaussian workload is smaller than that in the uniform workload.

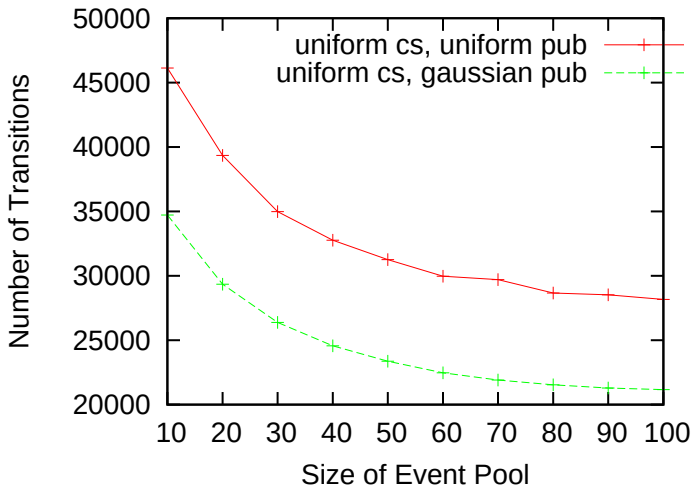


Figure 6.17: #Transitions vs. pool size

6.4.2 Prediction Performance

We evaluate our prediction algorithm on three important metrics: false positives, true positives and precision, which is defined as the ratio between true positives and all predictions. We look at the effect of the number of lookaheads, the threshold, and the ratio between the number of full matches and partial matches, and at the effect of the workload distribution. The prediction results are described as follows.

First, we compare the precision results when using the same workload to train our model, but test the prediction algorithm on different workloads. The workloads are different in terms of the increased number of partial matches (i.e., decreased ratio of the number of full matches and partial matches) in the event stream. In Figure 6.18, we can see that the precision decreases as the number of lookaheads increases. Also, the precision increases with the increase of prediction threshold but it stabilizes for larger thresholds. The precision decreases when the test file contains more partial matches.

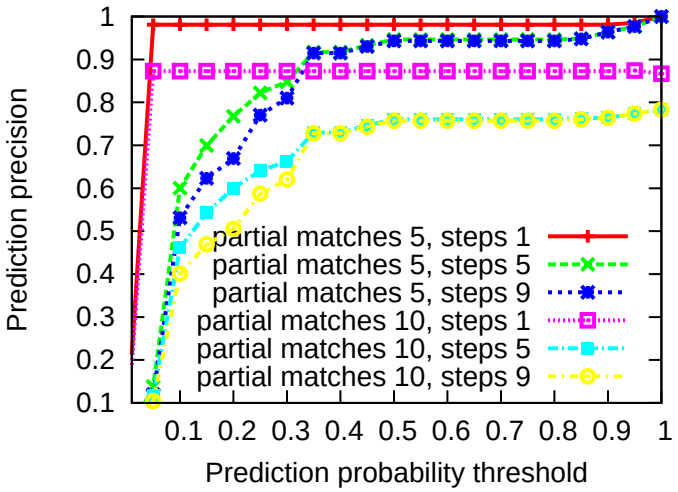
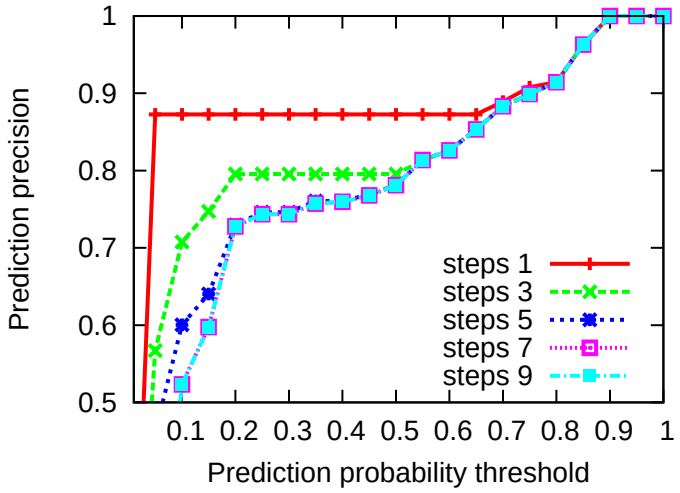


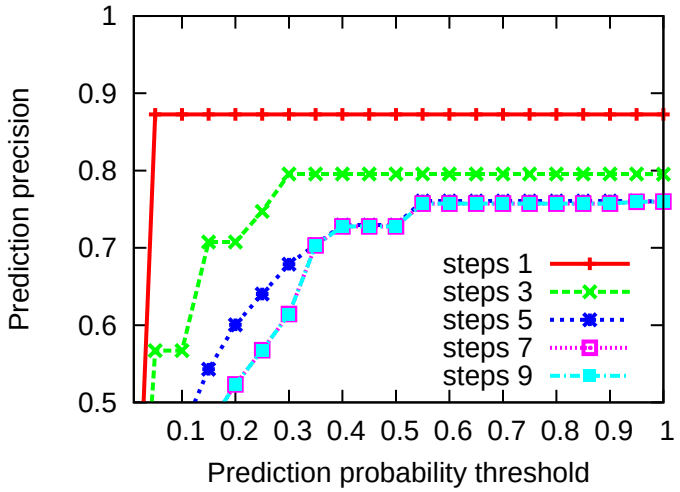
Figure 6.18: Precision vs. #partial matches

Second, we compare the precision results when using different workloads for training, but test on the same workload. In this experiment, the workloads are different in terms of the event distribution; all the other parameters are the same. Figure 6.19(a) shows the precision when testing on the same workload as training. Figure 6.19(b) shows the precision when training with a Gaussian event stream and testing on a uniform event stream. Comparing these two figures, we see that the precision decreases when the training workload and testing workload are not consistent. An additional discovery is that the precision converges for different number of lookaheads in Figure 6.19(a). We can conclude from this result that if we train and test on the workload with the same distribution, an increase in the number of lookahead steps does not decrease the quality

of prediction for larger thresholds.



(a) Precision for uniform workload



(b) Precision for gaussian workload

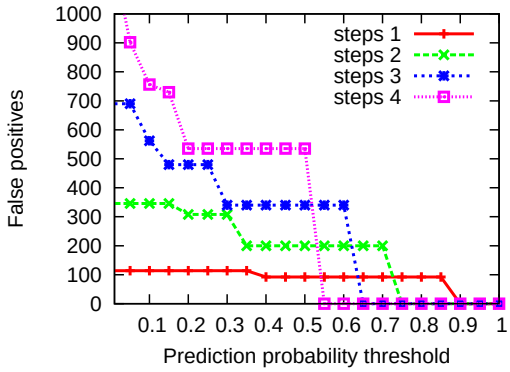
Figure 6.19: Comparing prediction results for different workload

In order to validate the practicality of our approach, we evaluated with a real-world data set, where the daily temperatures are monitored and an alert is announced when the temperature is high for several consecutive days, perhaps indicating a heat wave. We download a set of real weather data consisting of average daily temperatures for 157 U.S

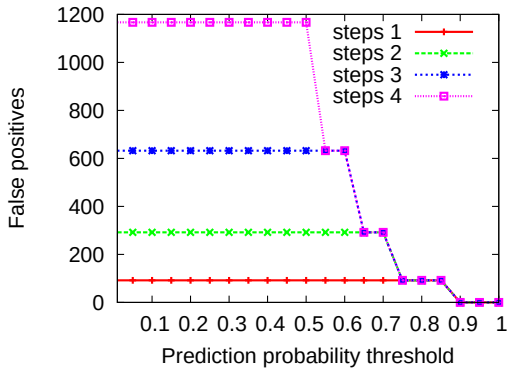
and 167 international cities [1]. We define the subscription as $s = (T > 30), cs = s, s, s, s, s,$ where it represents the case the temperature is higher than 30 degree for five consecutive days. The generated model is built based only on the composite subscription, without any knowledge of the event data. In the generated model, there are only 5 states and each state represents one more hot day than the previous state. For comparison, we divide the temperature into 4 categories separated by $30, 20, 10$ degrees and manually build a *full* Markov model consisting of all temperature combinations from one day to 5 days. There are 1365 states in total. We ran our predicting algorithm for these two models and the results are shown in 6.20.

Comparing figures 6.20(a), 6.20(b), 6.20(c) and 6.20(d), we can see that generated model makes more predictions than the full model. Both the number of false positives and true positives are bigger in the generated model than the full model. However with a larger number of lookaheads, both false positives and true positives drops to zero faster in the full model than in the generated model when increasing the prediction threshold. This is because the full model consists of much more states and information, thus we can differentiate the cases how the partial match status is reached and be more conservative to make predictions. Figures 6.20(e) and 6.20(f) shows the prediction precisions. When increasing the lookaheads, the generated model performs much better than the full model. Furthermore, the number of states in our model is much less than that in the full model, thus the prediction algorithm runs much faster on the generated model than on the full model, the cost to maintain the transition probability is also much smaller in the generated model.

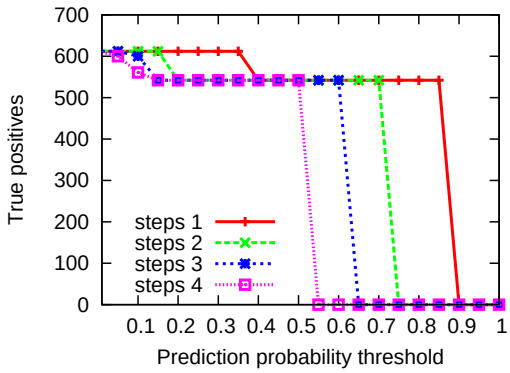
[1] The data is downloaded from *http://www.engr.udayton.edu/weather/source.htm*. Source data are from the Global Summary of the Day (GSOD) database archived by the National Climatic Data Center. The average daily temperatures are computed from 24 hourly temperature readings in the Global Summary of the Day (GSOD) data.



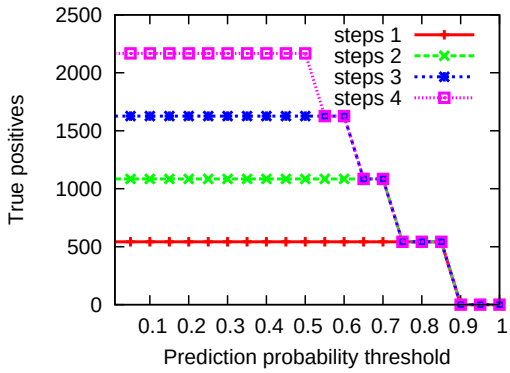
(a) False positives (full model)



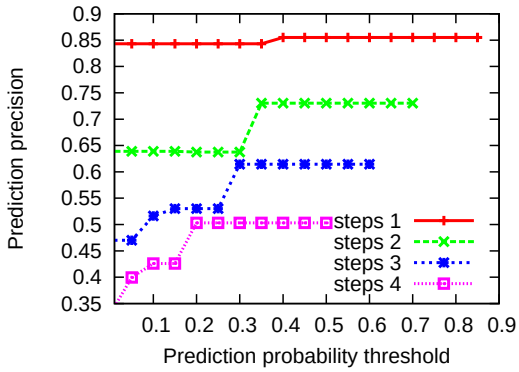
(b) False positives (generated)



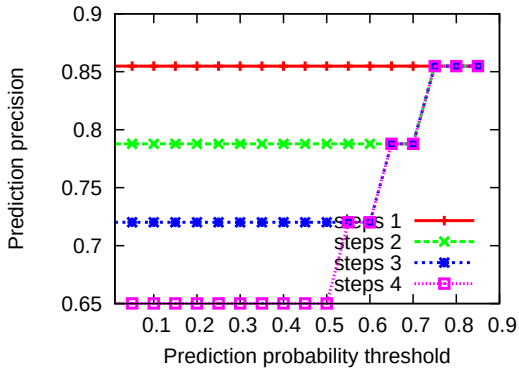
(c) True positives (full model)



(d) True positives (generated)



(e) Prediction precision (full model)



(f) Prediction precision (generated)

Figure 6.20: Evaluation on real data, compared with a model with full knowledge

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we propose an approximate publish/subscribe model to express uncertainties in both subscriptions and publications when exact information is not available. Based on the approximate model, we define an approximate matching mechanism. We developed two algorithms to optimize routing performance – approximate covering-based routing and approximate merging-based routing based on the similarity of subscriptions. The system we describe is able to support high matching rates for very complex subscriptions through covering and merging optimizations. Our experiments show reduction of at least 25% in matching time, 30% in routing table size, while resulting only less than 5% false positives. The approximate routing algorithms are expressive in the sense that the possibility and necessity measure associated with the similarity provides flexibility to tune the trade off between routing table size and false positives.

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 thesis, are a very good fit for such applications. The

system we presented in this thesis 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 architecture with one server and two client nodes, the throughput is twice of that of a centralized architecture; while the throughput increases to 400% for an architecture with one server and six client nodes. Based on our experimental results, we calculate the index size be $150K$ when the imprecise indexing algorithm is used to manage millions of subscriptions. Since the filtering algorithm can process $150K$ subscriptions in a single node, we can support 1 million subscriptions on a computer cluster with 7 client nodes.

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.

In this thesis, we propose a probabilistic publish/subscribe model to match and predict future matches of composite subscription pattern with temporal and Boolean operators. In the model, a finite state machine is built for the temporal subexpressions of each pattern. The finite state machine is a Markov chain, with transition probabilities obtained from given event histories, and is used to compute pattern match probabilites and notify subscribers of possible future matches.

To optimize our matching and prediction algorithms, those finite state machines that share certain common subexpressions are merged into a single machine. This optimization helps save memory and improve the matching efficiency while maintaining the cor-

rectness of the match probability computation.

7.2 Future Work

Uncertainties exist in many aspects in information dissemination systems. Investigate uncertainties management in information filtering systems due to the lack of information, imprecision and semantic ambiguity intrinsic to data is becoming more and more essential. Based on my previous work, future enhancements to manage uncertainties fall in the area of representation, filtering and processing of imprecise data.

In this thesis we designed an approximate matching algorithm to process information with uncertainties. The question it leaves for us is how to improve the algorithm efficiency. Can we pick out the matched predicates and subscriptions more quickly based on the properties of membership functions used to describe the vagueness? Are the users satisfactory degrees useful to trim out unnecessary data? When applying algorithm to select the function parameters for approximate publish/subscribe model, clustering algorithms such as K-means are good alternatives for merging especially when the distribution of subscriptions follow certain patterns. With the imprecision nature of our model, how to apply approximate clustering algorithms to decide the parameters for our membership is also an interesting direction.

For graph-based metadata filtering, *join* operation is used in both indexing and filtering algorithms and this operation consumes large portion of the time complexity. Improve the join operation is the key to improve the algorithm efficiency for indexing and query processing. One avenue of future research is how to allow a join operator combine multiple tuples into a new tuple based on some constraints.

Being based on RDF, the G-ToPSS system can be easily extended to use additional semantic information expressed in languages built on top of RDF, such as RDFS and OWL. RDFS taxonomy can be used to increase the expressiveness of the query language.

In the future, we will work on extending the covering and merging indexing algorithms not only from the syntax relationship among subscriptions, but also based on the semantics (taxonomy). With the additional semantics information about the query and RSS documents, the workload partition would achieve more accurate matching results and higher throughput.

For probabilistic publish/subscribe system, currently the Markov probability distribution is build based on composite subscription pattern, in other words, on top of the finite state machine where the states are directly known. However, if the subscription patterns change frequently with insertion and deletion, the current model incurs large cost to maintain. Therefore, an alternative approach is to construct a Hidden Markov Model based on the event stream itself, where the transition states are not directly visible. Currently the prediction is performed for crisp data which means that both the event and subscription patterns information are certain. To incorporate the A-ToPSS and G-ToPSS model, how to conduct prediction for uncertain data and for semantic information leave problems for long term future work.

Bibliography

[1] *Web Services Notification (WS-Notification), Version 1.0.* 2004.

[2] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD/PODS Conference (SIGMOD 2008)*, 2008.

[3] 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*, pages 53–61, 1999.

[4] M. Altherr and M. Erzberger. ibus- a sfotware bus middleware for the java platform. In *International Workshop on Reliable Middleware Systems*, pages 43–53, 1999.

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

[6] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *the International Conference on Data Engineering*, Cancun, Mexico, 2008.

[7] Ghazaleh Ashayer, Hubert Leung, and H.-Arno Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *DEBS'02 Workshop at ICDCS'02 (DEBS'02)*, Vienna, Austria, 2002.

[8] S. Bahu and J Widom. Continuous queries over data streams. *ACM Special Interest Group on Management of Data (SIGMOD) Record*, 3:109–120, 2001.

[9] 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 *International Conference on Distributed Computing Systems*, 1999.

[10] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS)'99*, 1999.

[11] D.J. Barrett, L.A. Clarke, P.L. Tarr, and A.E. Wise. A framework for event-based software integration. *ACM Transaction on Software Engineering and Methodology*, 5(4):378–421, 1996.

[12] O. Benjelloun, A. Das Sarma, C. Hayworth, and J. Widom. An introduction to uldbs and the trio system. *IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases*, 29(1):5–16, 2006.

[13] P. Bose, M. Galibourg, and G. Hamon. Fuzzy querying with SQL: Extensions and implementation aspects. *Fuzzy Sets and Systems*, 28, 1988.

[14] Ioana Burcea, Hans-Arno Jacobsen, Eyal de Lara, Vinod Muthusamy, and Milenko Petrovic. Disconnected operation in publish/subscribe middleware. In *Mobile Data Management*, pages 39–, 2004.

[15] Ioana Burcea and H.-Arno Jacobson. L-ToPSS - push-oriented location based services. In *Proceedings of the 2003 Workshop on Technologies for E-Services*, Lecture Notes in Computer Science. Springer, 2003.

[16] 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, August 2001.

[17] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with xpath expressions. In *International Conference on Data Engineering (ICDE)*, 2001.

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

[19] J. Chen, D.J. Dewitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. *The 19th ACM SIGMOD International conference on Management of Data*, pages 9–17, 2000.

[20] Paolo Ciaccia, Danilo Montesi, Wilma Penzo, and Alberto Trombetta. Fuzzy query languages for multimedia data.

[21] 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.

[22] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, 2000.

[23] A. Compailla, S. Chaki, S. Jha, and H. Veith. Efficient filtering in publish-subscribe system using binary decision diagrams. In *23rd International Conference on Software Engineering(ICSE)*, 2001.

[24] Coral8. *http://www.coral8.com*.

[25] T. Corporation. Everything you need to know about middleware: Mission-critical interprocess communication (white paper). 1999. http://www.talarian.com.

[26] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso 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.

[27] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16(4):523–544, 2007.

[28] Yanlei Diao, Peter Fischer, Michael Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of International Conference on Data Engineering*, 2002.

[29] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internat-scale xml dissemination service. In *Proceedings of the International Conference on Very Large Data Bases (VLDB 2004)*, 2004.

[30] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[31] Didier Dubois and Henri Prade. *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. Plenum Press, New York, 1988.

[32] P. T. Eugster, R. Guerraoui, and J. Sventek. Type-based publish/subscribe. *Technical Report, Distributed Programming Laboratory, Ecole Polytechnique Federale de Lausanne*, 2000.

[33] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[34] Francoise Fabret, H.-Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *The 20th ACM SIGMOD International Conference on Management of Data*, 2001.

[35] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. ACM SIGMOND/SIGACT conf. on Princ. of Database Syst. (PODS)*, Montreal, Canada, 1996.

[36] R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of ACM SIGMOND/SIGACT conference on Principle of Database Systems. (PODS)*, Seattle, WA, USA, 1998.

[37] R. Fagin, Lotem A., and Naor M. Optimal aggregation algorithms for middleware. In *Proc. Twentieth ACM Symposium on Principles of Database Systems*, pages 102–113, 2001.

[38] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *International Conference on Feature Interactions in Telecommunications and Software Systems(ICFI'05)*, Leisester, UK, 2005.

[39] Norbert Fuhr and Thomas Rolleke. A probabilistic relational algebra for integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15(1), 1997.

[40] G. Gottlob, C. Koch, and K.U. Schulz. Conjunctive queries over trees. *Journal of ACM*, 53(2):238–272, 2006.

[41] Ashish Kumar Gupta and Dan Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2003. ACM Press.

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

[43] M. Happner, R. Burridge, and R. Sharma. Java message service. October 1998.

[44] M. Happner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java message service. Sun Microsystems Inc., 2002.

[45] John Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.

[46] Shuang Hou and H.-Arno Jacobsen. Predicate-based filtering of xpath expressions. In *Proceedings of the 20th International Conference on Data Engineering*, 2006.

[47] Shuang Hou and Hans-Arno Jacobsen. Predicate-based filtering of xpath expressions. In *the 22nd International Conference on Data Engineering*, Atlanta, Georgia, USA, April 2006.

[48] H.Whitney. Congruent graphs and the connectivity of graphs. *J. AM*, 23(1):31–42, 1976.

[49] George J. Klir and Tina A. Folger. *Fuzzy Sets, Uncertainty, and Information*. Prentice Hall International Editions, 1992.

[50] OWL Web Ontology Language. *http://www.w3.org/TR/owl-features/*.

[51] Hubert Ka Yau Leung. Subject space: A state-persistent model for publish/subscribe systems. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 2002.

[52] K.S. Leung, M.H. Wong, and W. Lam. A fuzzy expert database system. *Data and Knowledge Engineering*, 4:287–304, 1989.

[53] Charles L.Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[54] G. Li, Sh. 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)*.

[55] Guoli Li, Alex Cheung, Shuang Hou, Songlin Hu, Vinod Muthusamy, Reza Sherafat, Alex Wun, Hans-Arno Jacobsen, and Serge Manoviski. Historic data access in publish/subscribe. In *International Conference on Distributed Event-based Systems*, Toronto, Canada, June 2007.

[56] Haifeng Liu and H.-Arno Jacobsen. A-ToPSS - a publish/subscribe system supporting approximate matching. In *Proceeding of the 28th International Conference on Very Large Data Bases, Demonstration*, Hong Kong, August 2002.

[57] Haifeng Liu and H.-Arno Jacobsen. Approximate matching in publish/subscribe. In *Proc. 5th IEEE International Symposium on Computational Intelligence in Robotics and Automation*, Japan, 2003.

[58] Haifeng Liu and H.-Arno Jacobsen. A-ToPSS - a publish/subscribe system supporting imperfect information processing. In *the 30th International Conference on Very Large Data Bases, Demonstration*, Toronto, Canada, 2004.

[59] Haifeng Liu and H.-Arno Jacobsen. Modeling uncertainties in publish/subscribe system. In *Proceedings of the 20th International Conference on Data Engineering*, Boston, USA, April 2004.

[60] Haifeng Liu and Hans-Arno Jacobsen. Object-oriented publish/subscribe for modeling and processing of uncertain information. *Chapter in Advances in Fuzzy Object-Oriented Databases: Modeling and Application*, pages 301–302, 2005.

[61] Haifeng Liu, Milenko Petrovic, and Hans-Arno Jacobsen. Efficient and scalable filtering of graph-based metadata. *Journal of Web Semantics*, 2006.

[62] L. Liu, C. Pu, and W. Tang. Continuous queries for internet scale event-driven information delivery. *IEEE Transaction on Knowledge and Data Engineering*, 11(4):583–590, 1999.

[63] S. P. Meyn and R.L. Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2005.

[64] G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *Journal of ACM*, 51(1):2–45, 2004.

[65] Richard Monson-Haefel and David A. Chappell. Java message service. In *O'Reilly*, 2001.

[66] Object Management Group. *Event Service Specification, Version 1.1*. 2001.

[67] Object Management Group. Notification service specification, version 1.0.1. August 2002. http://www.research.att.com/ ready.

[68] Object Management Group. *Notification Service Specification, version 1.0.1*. 2002.

[69] Olga Papaemmanouil and Ugur Cetintemel. Semcast: Semantic multicast for content-based data dissemination. In *Proceedings of International Conference on Data Engineering*, 2005.

[70] Joao Pereira, Franoise Fabret, H.-Arno Jacobesen, Franois Llirbat, and Dennis Shasha. WebFilter: A high-throughput XML-based publish and subscribe system. In *VLDB conference*, 2002.

[71] Milenko Petrovic, Ioana Burcea, and H.-Arno Jacobsen. S-ToPSS - a semantic publish/subscribe system. In *Proceedings of the 2nd Annual International Conference*

on Mobile and Ubiquitous Systems (MobiQuitous 2005), San Diego, CA, USA, July 2005.

[72] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. CMS-ToPSS - efficient dissemination of rss documents. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB), Demonstration*, September 2005.

[73] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. G-ToPSS - fast filtering of graph-based metadata. In *the 14th International World Wide Web Conference*, Chiba, Japan, May 2005.

[74] Milenko Petrovic, Vinod Muthusamy, and H.-Arno Jacobsen. Content-based routing in mobile ad hoc networks. In *Proceedings of the IEEE MobiQuitous*, Berlin, Germany, September 2003.

[75] F.E. Petry. Fuzzy databases: Principles and applications, with contribution by patrick bose. *International Series in Intelligent Technologies*, page 240, 1996.

[76] The Resource Description Framework (RDF). *http://www.w3.org/RDF/*.

[77] RSS Readers. *http://www.rss-specifications.com/aggregator-how-to.htm*.

[78] Anton Riabov, Zhen Liu, Joel L.Wolf, Philip S. Yu, and Li Zhang. Clustering algorithms for content-based publicaation-subscription systems. In *Proceedings of International Conference on Distributed Computing Systems*, 2002.

[79] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In *6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, pages 344–360, 1997.

[80] RDF Schema. *http://www.w3.org/TR/2000/CR-rdf-schema-20000327/*.

[81] Sarvjeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne Hambrusch, and Rahul Shah. Orion 2.0: Native support for uncertain data. In *the ACM Special Interest Group on Management of Data (SIGMOD 2008)*, Vancouver, Canada, 2008.

[82] D. Skeen. Vitria's publish-subscribe architecture: Publish-subscribe overview. 1998. http://www.vitria.com.

[83] Philippe Smets. Imperfect information: Imprecision and uncertainty. In *Uncertainty Management in Information Systems*, pages 225–254. 1996.

[84] StreamBase. http://www.streambase.com.

[85] RDF Site Summary. *web.resource.org/rss/1.0/spec*.

[86] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness - transparent information delivery for mobile and invisible computing. In *2001 IEEE International Symposium on Cluster Computing and the Grid*, 2001.

[87] David Tam, Reza Azimi, and H.-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *International Workshop On Databases, Information Systems and Peer-to-Peer Computing*, Berlin, Germany, September 2003.

[88] The READY Project Group AT&T Labs Research. Ready a high performance event notification service. http://www.research.att.com/ ready.

[89] TIBCO Inc. Tibco/rendezvous concepts. October 2000.

[90] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless microsensor network models, 2002.

[91] J. R. Ullmann. An algorithm for subgraph isomorphism. *Amer.J.Math*, 54:150–168, 1932.

[92] Jinling Wang, Beihong Jin, and Jing Li. An Ontology-Based Publish/Subscribe System. In *Middleware*, 2004.

[93] 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.

[94] Ouri Wolfson, Ana Lelescu, and Bo Xu. Approximate retrieval from multimedia databases using relevance feedback.

[95] A. Wolski and T. Bouaziz. Fuzzy triggers: Incorporating imprecise reasoning into active database. In *Proceedings of the 14th International Conference on Data Engineering*, 1998.

[96] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD/PODS Conference (SIGMOD 2006)*, 2006.

[97] Zhengdao Xu and H. Arno Jacobsen. Location constraing processing. In *Proceedings of 30th International Conference on Very Large Data Bases (VLDB), Demonstration*, 2004.

[98] T.W. Yan and H.G. Molina. Index structures for information filtering under the vector space model. In *Proceedings of the International Conference on Data Engineering*, November 1993.