A Fast Filtering Engine for Publish/Subscribe System

by

Mohammad Rubaiyat Ferdous Jewel

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

Your file Votre référence
ISBN: 0-612-91440-2
Our file Notre référence
ISBN: 0-612-91440-2

Abstract

A Fast Filtering Engine for Publish/Subscribe System

Mohammad Rubaiyat Ferdous Jewel

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

The current trend in the distributed systems is moving away from tightly-coupled systems towards systems of loosely-coupled, dynamically bound components. Due to the demands of modern applications, the popularity of *Content-based* publish/subscribe systems is rapidly increasing. However, one of the principal challenges faced by the publish/subscribe research community is to design an efficient matching algorithm for a flexible subscription language. To date, matching algorithms developed for publish/subscribe systems sacrifice subscription language expressiveness for matching performance. For example, almost all state-of-the-art matching algorithms can only support conjunctive subscriptions and many allow only equality predicates. In this thesis, we develop and evaluate a new matching algorithm that can handle any free form expression over a number of predicates. We also propose two extensions to our basic algorithm that can give additional performance benefits at the cost of a slightly restrictive subscription language. Our performance results demonstrate that the algorithm can achieve substantial event processing rates for very large number of subscriptions processed in the system.

Acknowledgements

This thesis is the result of two years of work whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

First, I would like to pay my heartiest thanks and gratitude to H.-Arno Jacobsen, my supervisor, for his expert supervision and constant source of encouragement without which this work would not be materialized in the present form.

I express sincere gratitude to Renee Miller who as my second reader provided many constructive comments on the preliminary version of this thesis. I learnt a lot from all her comments and corrections.

Special thanks goes to Badih Schoueri and Ziad Odeh for their cooperation during my work. Needless to say that I am grateful to all of my colleagues at the Department of Computer Science at the University of Toronto and at the Department of Computer Science and Engineering at the Bangladesh University of Engineering and Technology, for their support and friendship.

I am very grateful for the love and support of my parents and my family. Last but not least, I would like to recognize the unrelenting support my wife, Farah Farzana whose patient love enabled me to complete this work. I would also like to thank her for her constructive criticism and suggestions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Within the last decade, due to the advances in communications and networking, computing systems research has spread its areas vastly in many new and potential directions. Distributed systems not only kept itself within the boundary of academic research, but also became one of the principal focuses in the computing industry. Many standard applications are now redesigned into a distributed platform, leaving their old monolithic approach behind. Business processes and their deployment are getting automated and interconnected day by day. Integration of independent applications is becoming one of the prime concerns in the industry.

The importance of flexible, well-structured, but especially scalable communication mechanisms has been drastically increasing during the last decade. Applications tend to become very dynamic, i.e., components are not always up and are not locality-bound. These constraints demand more flexible communication models, reflecting the nature of tomorrows applications. The publish/subscribe interaction style has proven its ability to fill this gap. Based on the concept of information bus [28], publish/subscribe promotes the decoupling of parties in time as well as space. Consumers subscribe to the information bus by specifiying the nature of the information they are interested in, and producers publish information on that bus.

One of the main challenges faced today by the publish/subscribe research community is to efficiently solve the publish/subscribe matching problem for a given subscription language and publication data model. Much research effort has been devoted to solving this problem [1, 4, 15, 14, 25]. In this thesis, we develop and evaluate a new matching algorithm that can handle any free form expression over a significant number of predicates. Our performance results show that the algorithm can achieve substantial event processing rates for very large number of subscriptions processed in the system.

This chapter gives a broad overview of our research topic starting with the current trends in Middleware Systems in Section 1.1. Section 1.2 provides the introduction to Publish/Subscribe systems. We define our research objectives in Section 1.3. The organization of the thesis is presented in Section 1.4.

1.1 Middleware Systems

Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems [3]. Informally known as a *glue technology*, middleware is defined as a layer of software above the operating system but below the application program, providing common application programming interfaces (API) to connect two applications in a distributed system.

The wide spectrum of disseminated pieces of information available to the users through the Internet or within a local area network makes distributed systems inherently heterogeneous. Middleware hides such complexity and heterogeneity. Within the course of time, middleware has grown in different directions, and we currently see a few major categories in middleware [3]. These vary in terms of the programming abstractions and the kinds of heterogeneity they provide beyond network and hardware.

Distributed *Tuple Space*, e.g. Linda [18], was an early approach to manage heterogeneity across different programming languages. The basic idea behind tuple space is

that one can have many active programs distributed over physically dispersed machines, unaware of each other's existence, and yet still be able to communicate. They communicate to each other by releasing data (a tuple) into the tuple space. Programs read, write, and take tuples that are of interest to them from the tuple space. In order to model a publish/subscribe using tuple space, the read operation will model a subscription, and any write operation will model a publication. Linda provides a library of functions to be called to utilize the middleware, hence relieving the programmers from the burden of learning a new programming language [3]. Jini[1], a recently developed Java[2] framework, is built on top of JavaSpaces[3], which closely resembles tuple space. Jini allows fast, easy incorporation of legacy, current, and future network components. *Remote Procedure Call* (RPC) [27] provides an interface that allows one process to execute a procedure in another system using an ordinary procedure interface. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces–function calls are the programmer's interface when using RPC [31]. However, RPC is based on the twin assumptions that, both client and server are available simultaneously and, that suitable actions can be taken to ensure the completion of the RPC should either the client or server fail. In contrast, the tuple space model is an asynchronous paradigm which enforces no assumptions about timeliness, ordering or synchronisation. emphMessage-Oriented Middleware (MOM)[32] provides the abstraction of a global message queue within a distributed system, where producers push their messages in the queue when they need to, and consumers receive messages from the queue. This allows the interaction in an asynchronous way, and decouples the producers and the consumers. MOM is actually a modification to the existing general tuple-space. Many MOM products offer queues with persistence, replication, or real-time performance. MOM of-

[1] www.sun.com/jini/
[2] java.sun.com
[3] java.sun.com/products/javaspaces/

fers the same kind of spatial and temporal decoupling that Linda does [3]. Recently, *Java Message Service* (JMS)[4] has become a messaging standard that allows application components to create, send, receive and read messages.

The distributed object concept introduced another approach to address the issue of distributed computing. A process can transparently use an object, which actually resides in another system, in a way as if the object is in the same address space of the process itself [3]. Different solutions of this concept at industry level are currently ready to use in order to achieve large scale solutions. The Common Object Request Broker Architecture (CORBA) is a standard for distributed object computing and is the broadest distributed object middleware available in terms of scope. CORBA offers heterogeneity across programming language and vendor implementations [3].

1.2 Publish/Subscribe Systems

The current trend in distributed systems is moving away from tightly-coupled systems towards systems of loosely-coupled, dynamically bound components. This trend fits the event-based application paradigm. The event-based approach is well-suited for integrating autonomous, heterogeneous components into complex systems by means of detecting and exchanging events. Event-based systems usually use a notification service which is responsible for delivering events to interested consumers. In a notification service based on the publish/subscribe paradigm, events flow from event producers to one or more consumers.

Publish/subscribe (pub/sub) is a messaging paradigm where information flows from the information producers or *publishers* to the information consumers or *subscribers* according to the specific selection criteria expressed by the individual subscribers. The subscribers express their interests by means of subscriptions, while publishers simply

[4]java.sun.com/products/jms/

publish information. The pub/sub paradigm decouples the communication between the subscribers, who subscribe their interests with the pub/sub broker and the producers. The broker filters the published information against the stored subscriptions. One of the main advantages of the publish-subscribe model is that it decouples publishers and subscribers in several dimensions. Eugster *et. al.* [13] introduce three dimensions of decoupling: space decoupling that captures the fact that interacting parties do not need to known each other, time decoupling that captures the fact that parties do not need to be actively participating in the interaction at the same time, and flow decoupling that captures the asynchrony of the model. This communication paradigm lends itself well in modeling and implementing selective information dissemination systems [34], enterprise application integration tasks, etc. Many popular middleware platforms have been exposing publish/subscribe-style application programming interfaces. This includes computing standards, such as the CORBA Event Service, the CORBA Notification Service, the OMG Data Dissemination Service, the Java Messaging Service, or the older DCE Event Management Specification, and middleware platform products, such as Sonic MQ and MQ Series, to name just two.

One of the principal challenges faced by the pub/sub broker is to efficiently solve the pub/sub matching problem for the given subscription language and publication data model. The basic idea behind the matching problem is to efficiently identify the subscriptions among all others stored at the broker that match a given event[5]. Much research effort has been devoted to solving this problem [19, 1, 15, 22, 12, 6]. Common to most of these solutions is the focus on *conjunctive subscription languages*(see Section 1.2.2). In this thesis we propose an efficient matching algorithm that is designed to be extremely flexible and support very expressive subscription languages.

[5]The terms *event* and *publication* are often used synonymously in the publish/subscribe literature.

1.2.1 Topic Based Publish/Subscribe

The conventional pub/sub system was primarily modeled as *Topic Based* pub/sub system. This is also known as *Channel Based* pub/sub system. The classic publish/subscribe interaction model is based on the notion of topics or subjects, which basically resemble groups [30]. Subscribing to a topic T can be viewed as becoming member of a group T. The topic abstraction however differs from the group abstraction by its more dynamic nature. While groups are usually disjoint sets of members topics typically overlap, i.e., a process participates (in the role of publisher and/or subscriber) in more than just one topic. Topics however allow only a limited expressiveness. The advantage of this paradigm is its simplicity, especially in the matching algorithm used.

The simplest example of a topic-based pub/sub is a newsgroup. A person, Alice, who is interested in getting news on the topic `sports.soccer`, can subscribe to this topic. So, all the news postings related to `sports.soccer` are forwarded to Alice whenever they arrive. In this example, Alice is a subscriber in this system. On the other side, there might be many news brokers who send out (a.k.a. publish) news articles frequently on different topics. All articles are categorized based on the currently existing channels or topics in the system. After getting an article from any broker, the pub/sub system will check if the topic of the article is `sports.soccer`. If so, the system will notify Alice about this newly published article. This provides the very basic filtering that just selects the appropriate article based on topic. The drawback of such a system is that publishers cannot provide more attributes on their events, and subscribers cannot use flexible subscription pattern. In our example, there is no way that will allow Alice to add additional conditions on her subscription. For example, using a topic-based model, it would not be possible for her to get news on the same topic only if Brazil scored more than three goals.

1.2.2 Content Based Publish/Subscribe

Recent research efforts have been targeted towards content-based or property-based publish/subscribe schemes. This more flexible variant allows consumers to delineate their individual interests by expressing properties of messages they wish to receive. In a content-based pub/sub, a subscription is defined as a set of predicates. Each predicate is essentially a triple (*pvar, op, pvalue*) containing one attribute variable *pvar*, one relational operator *op* and a value *pvalue*. To date, all subscription language models assume that the predicates in a single subscription are bound together by conjunctive operators (i.e., Boolean AND) [1, 15]. For example, a typical subscription in a selective information dissemination system for someone looking for a car, would be as follows

s = *(make, =, Honda), (model, =, Civic), (year, =, 2003), (price,* $\leq$*, 20,000.00)*

This indicates that the person wants a car made by Honda (model – Civic), brought to the market in the year 2003, and has a price less than or equal to $20,000.00. This implies that the user wants all of the predicates to be satisfied by an event. Hereafter, we refer to this subscription as *Conjunctive Subscription*, where an event has to satisfy every predicate of a subscription to satisfy that subscription itself.

An event in our system is a list of components where each component is a pair (*evar, evalue*) made up of an attribute variable *evar* and a value *evalue*. For example, a car seller might publish an event as,

e = *(make, Ford),(model, Taurus), (year, 2002),(price, 17000),*

which means the seller has a car made by Ford (model–Taurus), brought to the market in the year 2002, and a price of $17,000.00. A component (*evar, evalue*) of an event e matches a predicate (p = *pvar, op, pvalue*) if they both have the same attribute variable, i.e., *evar=pvar*, and the relation (*evalue op pvalue*) holds. As an example, we can say that an event (*length, 112*) matches the predicate (*length, >,100*) because they have the same attribute variable *length*, and the relation $112 > 100$ holds. It is clear that the event e, in our example, does not match the subscription s shown in the previous

paragraph. Matching an event against a subscription gives us a Boolean result of `true` or `false`. In the Section 4.2.1, we redefine the semantics of predicate match to leverage some more powerful subscription language models.

For many situations this interpretation of subscriptions is too restrictive, and more expressive subscription languages would help to define much more powerful information filters.

1.3 Objectives

The principal objective of our approach is to process any kind of subscription that is an arithmetic expression with any operator and any form of bracket structure. This representation will give the subscriber the freedom of choosing any kind of expression over the predicates. So we will allow a subscription of the form, *s = (make, =, Honda) AND ((model, =, Civic) OR (model, =, Accord)) AND ((year, =, 2003) OR (year, =, 2002)) AND (price, ≤, 20000).* We can represent this as a binary tree, as shown in Figure 1.1. This much more powerful subscription expresses the fact that the user is firm about her choice on the make and price of the car, but is ready to accept cars from two different years and two different models.

Had this subscription been expressed in the conventional conjunctive form, the subscriber would have to submit and manage four different subscriptions. More generally speaking, the number of subscriptions required in conjunctive form to express these disjuncts increases exponentially with the number of predicates. If we consider a subscription,

$s = (\ p_1\ OR\ p_2\ )\ AND\ (p_3\ OR\ p_4\ )\ AND\ (\ p_5\ OR\ p_6\ )$

where p_i is a predicate, it needs the following 8 different subscriptions in a conjunctive subscription language to express the same interest.

$$s_1 = p_1\ AND\ p_3\ AND\ p_5$$

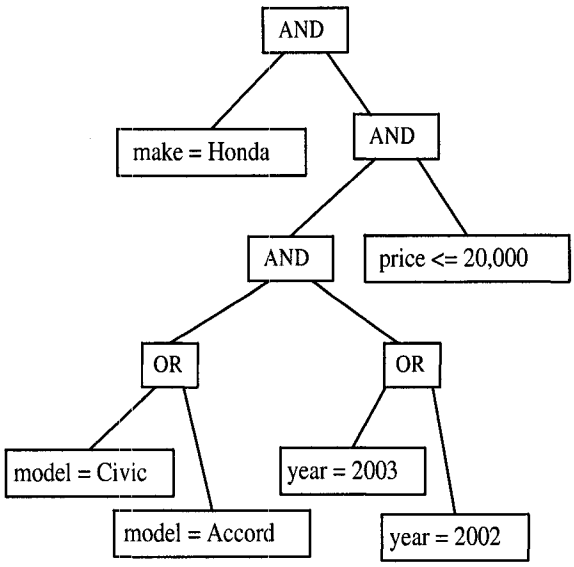


Figure 1.1: A complex subscription example

$$s_2 = p_1\ AND\ p_3\ AND\ p_6$$
$$s_3 = p_1\ AND\ p_4\ AND\ p_5$$
$$s_4 = p_1\ AND\ p_4\ AND\ p_6$$
$$s_5 = p_2\ AND\ p_3\ AND\ p_5$$
$$s_6 = p_2\ AND\ p_3\ AND\ p_6$$
$$s_7 = p_2\ AND\ p_4\ AND\ p_5$$
$$s_8 = p_2\ AND\ p_4\ AND\ p_6$$

Also, using the predicates and different Boolean operators without using brackets would not give us the exact expression we want. Obviously, two subscriptions

$s_1 = p_1\ AND\ (p_2\ OR\ p_3)$ and

$s_2 = (p_1\ AND\ p_2)\ OR\ p_3$

do not bear the same meaning. Our expression-based subscription language model can provide a powerful and flexible form of subscription without any restrictions. The set of relational operators we allow for predicates is the common choice $=, \leq, <, \geq, >, \neq$.

Furthermore, we do not restrict our subscription language to using only Boolean

operators. In Chapter 4, we show that we can handle any operator or function between the matched results of two predicates. We propose new predicate matching semantics, rather than using the conventional match result of `true` or `false`. Combined with this novel predicate matching semantics, our flexible subscription language can accommodate various powerful publish/subscribe models.

The contributions of our work are as follows:

1. We develop an original matching algorithm using a new proposed tree structure, referred to as an Aggregate-Tree (A-Tree), that is efficient and flexible. It can process any subscription expression, not just a conjunctive subscription language, which is the central focus of most algorithms developed to date.

2. We demonstrate the flexibility of A-Tree by showing how it can process various subscription language extensions not considered in other pub/sub approaches. These are a constraint-based subscription language, a partial match semantic, priority-based subscriptions, and composite subscriptions.

3. We provide a detailed scalability analysis of A-Tree and compare it to three other popular matching algorithms. Our results in Chapter 5 indicate that A-Tree is quite scalable, and at a very high number of subscriptions it performs almost as good as the popular counting algorithm, but not as good as the Gryphon algorithm. However, both counting and Gryphon can only process conjunctive subscription languages and are highly specialized for that purpose. So, the flexibility of A-Tree has a performance price but shows linear scalability for millions of subscriptions. Furthermore, in terms of memory overhead, the A-Tree algorithm performs as well as the Gryphon algorithm for a high number of subscriptions. However, for a lower number of subscriptions, Gryphon requires much less memory.

1.4 Outline

The remainder of this thesis is organized as follows. We summarize the related work and current trends in subscription matching algorithms, both in the research community and industry, in Chapter 2. In Chapter 3, we discuss our proposed algorithm and related data structures, show two optimized variants of our general algorithm, and briefly describe some properties of our tree structure. Next, in Chapter 4 we introduce our novel *Priority-based Subscription*, *Constraint-based Subscription* and *Partial Match-based Subscription* schemes. Additionally, we show the flexibility of our tree structure and propose different ways to further improve the performance of our algorithm. Chapter 5 contains the details of our experimental framework, and performance results evaluating our algorithms comparing with two other well-known algorithms. Finally, in Chapter 6 we conclude with brief discussions and also provide some directions for future research.

Chapter 2

Background

Publish-subscribe systems have received considerable attention in commercial products and standards, as well as academic research systems. Various matching algorithms for such systems have recently been the focus of intensive research. We can broadly classify the related work into four different categories, namely,

1. publish/subscribe matching algorithms,

2. stream-based processing,

3. continuous query processing, and

4. trigger management.

By far the most closely related approaches are the publish/subscribe matching algorithms developed to date. We will discuss four of these matching algorithms in Section 2.1. In Section 2.2 we present the ongoing research in matching algorithms based on XML based subscriptions, which is also somewhat related to our topic. We also discuss stream-based query processing in Section 2.3, and trigger processing in the database area in Section 2.4. Finally, we present the idea of a novel state-persistent subscription model in Section 2.5.

2.1 Publish/Subscribe Matching Algorithms

Much research has gone into developing efficient matching algorithms for content-based publish/subscribe systems [19, 1, 15, 22, 12, 6]. All these algorithms focus on processing a conjunctive subscription language over relational predicates.

Gough and Smith [19] use an approach based on finite state automata. A set of subscriptions is translated into a *non-deterministic finite state automaton* (NFSA), which is transformed into a *deterministic finite state automaton* (DFSA). The algorithm assumes a primary ordering of all n attribute variables available. Each subscription is modeled in a canonical form[1] consisting of n terms. The i-th term of each subscription is a predicate consisting of the i-th attribute variable. Each subscription is a path of n transitions, where the i-th transition depends on the matching value of i-th predicate. As a subscription, in reality, does not contain a predicate for each single attribute, the missing predicates are set as *don't care* transitions. A set of m subscriptions are aggregated together to form a compound NFSA.

Events are processed by advancing through the states of the machine, possibly ending at some terminating states, which comprise matching subscription identifiers. The algorithm has not been evaluated with different predicate and subscription ranges extensively. Its main weakness is the possible exponential step of translation between an NFSA and a DFSA. The structure can degenerate to a graph exponential in size. Subscription insertions are assumed to be processed in the NFSA, which must then be transformed into a DFSA again. So, insertions are very expensive. Besides, each insertion involves creating the compound NFSA structure again for all of the subscriptions together, and then forming the DFSA. So it is a weak structure in the aspect of maintenance. There is also no provision to delete a subscription. Deletion, like insertion, requires rearranging the new set of subscriptions.

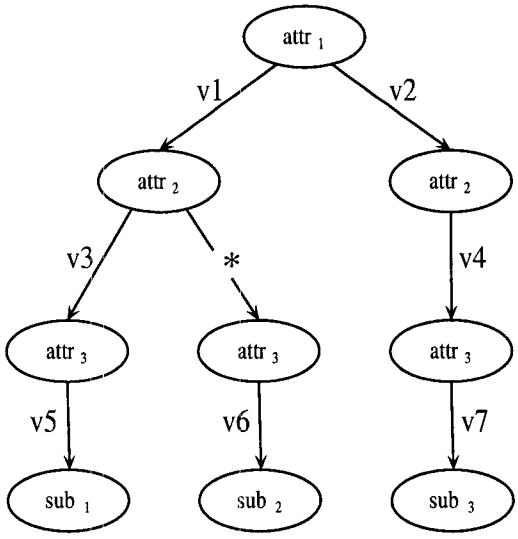
[1] A canonical form is a standardized form that minimizes the number of terms in an expression by suitably converting it.

The expressiveness of the subscription language of this algorithm is limited to conjunctive predicates. It also cannot handle range predicates. The result of the matching of a single predicate is only a boolean value, `true` or `false`.

The Gryphon [1] algorithm is another tree-based matching algorithm that represents subscriptions as a tree. The algorithm pre-processes the set of subscriptions into a data structure, called a *matching tree*, that allows fast matching. Each subscription is assumed to be a conjunction of predicates. The algorithm assumes a predicate-variable ordering and allocates nodes for predicate-tests. Tree edges are labeled with test results. Each non-leaf node contains a test, and edges coming out of a node represent the results. A leaf node l contains at least one subscription s. So the subscription s is represented by the path through the nodes of the tree starting from the root of the tree to l. If a subscription does not involve a certain predicate variable, a *don't–care* edge is added as a test result. Figure 2.1 shows an example match tree of Gryphon with three subscriptions having three equality predicate tests.

After construction of the matching tree, the algorithm can find the subscriptions that are matched by an event by traversing the tree starting from the root node. At each node α, it computes the test result using the attribute values taken from the current event, and follows the selected edges coming out of α that are consistent with the test result. Thus using a top-down traversal, the algorithm identifies the leaf nodes reachable by the current event, and hence finds the matched subscriptions.

The algorithm supports a conjunctive subscription language only. For a large number of subscriptions, the algorithm is very sensitive to the chosen ordering of variables. From the tree structure used by Gryphon, we see that it tries to make efficient reuse of common pefixes. So if two subscriptions have the same first m predicates, they will share the same prefix path for those m predicates in the matching tree. After that, they will follow different branches. So this common prefix is evaluated only once if required during matching an event. This reduces the number of tests required for a high number of



Matching tree structure

$$sub_1 := (attr_1 = V_1) \wedge (attr_2 = V_3) \wedge (attr_3 = V_5)$$

$$sub_2 := (attr_1 = V_1) \wedge (attr_2 = *) \wedge (attr_3 = V_6)$$

$$sub_3 := (attr_1 = V_2) \wedge (attr_2 = V_4) \wedge (attr_3 = V_7)$$

Figure 2.1: Gryphon tree example with three subscriptions

subscriptions. But it cannot take any advantage of common predicate chain existing in the middle or at the end of two subscriptions. A bad variable ordering can lead the Gryphon structure to blow up exponentially in size. For large subscription populations, insertions can take a very long time. The authors also did not address the issue of deletion of a subscription from the tree.

A specialized version of the Gryphon algorithm supports equality predicates only, analytically proving sub-linear speed-up. So, the Gryphon algorithm trades off performance for subscription language expressiveness, achieving significant performance for a very restrictive subscription language.

Fabret *et al.* [15] developed a matching algorithm for conjunctive subscription languages that clusters subscriptions according to access predicates. A predicate p can be an access predicate for a subscription s only if any event satisfying s satisfies p. Conversely, if an event does not satisfy p, it will not satisfy s. So if an access predicate is false, all the subscriptions residing in the associated cluster do not have to be evaluated, thus pruning the search space. As subscriptions are grouped based on their size and common conjunction of predicates, evaluating one predicate by an event gradually leads to a group of fewer of subscriptions. The algorithm also uses multi-attribute hash indices to evaluate several subscription attributes by a single comparison.

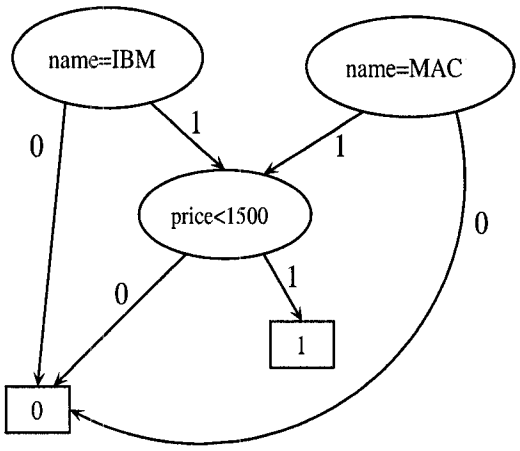
A cluster containing n subscriptions consists of n one-dimensional arrays. These arrays are called *predicate arrays* and each of them contains references to bit-vector entries. A cluster also has a one-dimensional array called a *subscription line* that contains subscription identifiers. An entry $[i, j]$ in the predicate array contains a bit-vector reference to the i^{th} predicate for a subscription s, the position of which is j in the subscription line. The subscription s will match an event if and only if all bit-vector entries referenced at column j of the predicate array are equal to 1. The matching algorithm goes through the whole array to check for the above specified condition.

Though it is a main memory algorithm, the authors are concerned about the processor

cache misses for the algorithm. The performance is improved by prefetching a block of memory into the cache before the algorithm actually uses it. This maximizes temporal and spatial locality, and avoids expensive cache misses.

The authors proposed two different approaches, static and dynamic, to build the clustering structure for a set of subscriptions. The latter considers the subscription population and statistics on incoming data items to incrementally reshape clusters. However, determining the access predicates is a non-trivial step and is not discussed and analyzed by the authors. Furthermore, the access predicate's pruning property only works for conjunctive subscription languages.

An algorithm that can process a more expressive subscription language, also involving disjunctive operators is the BDD-based scheme [4]. Ordered binary decision diagrams are canonical representations for Boolean formulae. Ordered BDDs can take advantage of overlapping sub-expressions in formulae by representing them with the same BDD-segments (Figure 2.2). This only works if the variable ordering is chosen advantageously, which depends on the *insertion-arrival-order* of subscriptions. It has been seen that BDDs do not scale well with an increase in the number of unique variables in the system [4]. To represent subscriptions with a BDD, each distinct predicate in the system must be represented with a unique variable. While there is a large degree of overlap among the predicates in a given subscription population, the number of distinct predicates may still be in the hundreds of thousands for any reasonable selective information dissemination application. Moreover, the evaluation of events with BDDs involves the traversal of the whole BDD. For publish/subscribe the canonical representation characteristic is not important and is not required. The A-Tree structure we propose is not canonical.



Matching tree structure

$sub_1 := (price < 1500) \wedge (name = IBM)$

$sub_2 := (price < 1500) \wedge (name = MAC)$

Variable ordering chosen: $(name = IBM) \prec (name = MAC) \prec (price < 1500)$

Figure 2.2: BDD example with three subscriptions

2.2 XML-based Matching Algorithms

Most data in the web is unstructured, containing text, graphics, multimedia, etc. Recently, XML is gaining popularity as a standard to exchange this kind of semi-structured data. Content based publish/subscribe also evolved in this paradigm. Several query languages have been proposed for XML, e.g. XML-QL [11], Quilt [5], XPath [9], XQuery [5], etc. A specific class of matching algorithms have evolved with the XML query language. The matching algorithms of this class operate on tree-structured data. Over the past five years, many such algorithms have been developed, such as YFilter [12] and Xtrie [6]. These algorithms assume a different subscription language model. The YFilter models a publish/subscribe system where the subscriptions are modeled by XPath expressions. An event is published in the form of an XML document. The XML document contains its Document Type Definition (DTD) in it, which is matched to all the subscriptions. The YFilter converts each XPath expression into a *Finite State Machine* having an ac-

cepting state. While matching the XML document, if an accepting state is reached, the corresponding subscription is satisfied. The performance of this matching depends on the ordering of the components in XML, as it makes use of common prefixes in the subscriptions, aiming to get the advantages achieved in other contexts by the popular Trie structure [16, 10]. This model does not apply for the subscription language and publication data model we use.

2.3 Stream-based Matching Algorithms

Work on data stream processing [7] and continuous query processing [8, 26] constitute further related approaches. Continuous queries are persistent queries that allow users to receive new results when they become available. NiagaraCQ [8] allows a large number of users to be able to register continuous queries in some high-level query language, in this case XML-QL. It models XML documents with predefined DTDs as the publications or events. Unlike previous group optimization efforts that mainly focused on finding an optimal plan for a small number of similar queries, the authors proposed a new incremental group optimization approach in which queries are grouped according to their signatures. Instead of regrouping all the queries in the system with the arrival of each new query, all the existing groups are considered as possible optimization choices. This scheme can be implemented using a general query engine and is very scalable.

NiagaraCQ supports two categories of queries. *Change-based* continuous queries are fired as soon as new relevant data becomes available. This refers to the semantics of a typical subscription. *Timer-based* continuous queries are executed only at time intervals specified by the user. When the time period is passed, the query is deleted from the system. This adds an additional temporal dimension to a typical subscription. Adding such a time frame would not be a big challenge in any publish/subscribe system, if it has the provision to delete a subscription.

To make NiagaraCQ scalable and efficient, the authors used incremental evaluation of continuous queries by considering only the changed portion of each updated XML file, not the entire file. The authors assume that only a small portion of each incoming XML file gets updated frequently if it is compared to the previous one, and hence it can save a significant amount of computation. In our observation, the key difference between data stream processing and publish/subscribe is that data streams assume a computational model based on a stream of data over which queries are evaluated, whereas publish/subscribe, in its most basic interpretation, does not operate over streams, but over individual sets of data tuples representing publications. Publish/subscribe does not aggregate data and does not keep state over a window of data in a stream. Publish/subscribe data tuples are processed in a highly asynchronous manner by the publish/subscribe broker and data tuples arrive at the broker from the most diverse and highly distributed publishers. In terms of matching algorithms, there are a few similarities, as has been shown by Fabret *et al.* [15]. They use a predicate counting-based algorithm to evaluate matching subscriptions over streams. In the publish/subscribe literature such algorithms have been used by Yan and García-Molina [33], for example.

2.4 Trigger Processing

Database triggers have also been intensively studied. Hanson *et. al* presented algorithms for achieving scalability in trigger processing [22]. The actual performance gains are not quantified. Conventional database systems are designed for fast and efficient evaluation of queries against stored data. In contrast, the queries are stored, and an event is evaluated against these stored queries in a Publish/Subscribe system. Trigger processing in database management systems can be the basis for implementing a publish/subscribe system. Database triggers can be used to model subscriptions. To manage a very high volume of subscriptions the trigger management system has to be scalable. We are not

aware of any study actually doing this and hence the scalability of such an approach is not known. TriggerMan [22] proposes optimization techniques for trigger executions.

2.5 State-persistent Subscription Model

An interesting model known as the *state persistent model* was proposed recently by Leung and Jacobsen [21]. Here, subscriptions and publications are still expressed using the conventional topic-based model, but a new component – persistence, is added to a subscription. So matching is redefined as a function with this extra parameter of persistence. When an event matches a subscription S for the first time, the state of S is set to `matched`. For future events, even if they match S according to the subscription expression of S, the subscription is not considered as satisfied and notifications are not sent to the subscriber involved. For example, a user might want to be informed when the price of IBM share goes above $30.00. The predicate for this can be expressed as $p = IBM > 30$. When the price goes above $30.00 for the first time, the user is notified. Afterward, if another event comes in with price higher than $30.00, the user is not notified as her subscription is already in the matched state. If the price goes down and goes above $30.00 again, the user will be notified. So it reduces the redundant notification traffic. A notification to a subscriber is sent out only if the state of the subscriber is changed by an event.

Chapter 3

The A-Tree Matching Algorithm

This chapter describes the *Aggregate Tree* (A-Tree) matching algorithm in detail. We first discuss the intuition behind our design in Section 3.1, and its pros and cons. We then describe the data structures the algorithm is based on followed by the actual algorithm, including the matching, deletion, and insertion operations. In order to highlight the flexibility of the algorithm, we also describe a set of unique properties of the algorithm. Finally, we briefly describe our implementation.

3.1 Intuition Behind Algorithm Design

Our proposed idea of an *Aggregate Tree* (A-Tree) is inspired from the classical *Arithmetic Expression Tree* (AET) concept [2]. An AET is a convenient tree structure to represent any mathematical expression, such as algebraic expression, Boolean expression, etc. All the leaves of such a tree represent the operands of an expression and the non-leaf nodes represent operators. This allows the representation of any complex expression comprising different operators and brackets. For example, if we consider the following arithmetic expression

$$(a * b) + (c/d)$$

it can be represented by the tree shown in Figure 3.1.

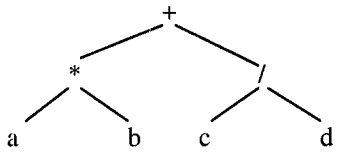


Figure 3.1: An example of arithmetic expression tree

The idea of A-Tree is also based on a similar concept. We shall demonstrate this in Section 3.2.

3.2 Tree Structure

In our A-Tree representation, subscriptions are represented as trees, just like arithmetic expressions are represented in an AET. Leaves of such a subscription tree are the subscription's predicates. An intermediate node in the tree contains the Boolean operator, with two child nodes, either leaf or non-leaf. In the next section, we show that we can support a very general subscription language with any kind of operator, not just Boolean operators. Here, we limit ourselves to discussing the Boolean operator case for simplicity. The root of the subscription tree keeps track of the subscription identifiers of the represented subscriptions. While inserting more subscriptions, we try to reuse the existing expressions from the A-Tree. So instead of creating a completely new set of nodes for a subscription, we share the sub-trees between two subscriptions as much as possible. This finally gives us an *aggregation* of a large number of binary trees, having shared sub-trees. The A-Tree is thus a forest of binary trees, since it will have multiple roots (see Figure 3.2 for an example).

3.2.1 Example with Multiple Subscriptions

To provide an example, we assume a web portal with apartment rental service where subscribers are people looking for different types of apartments to rent. In this context, we define a few simple predicate variables to define the subscriptions. The user might want to rent either an apartment or a house. The monthly rent can be any positive real number. For simplicity, we assume that the users are interested in renting living places having 1, 2 or 3 bedrooms in either of the two cities – Toronto or York. The predicate variables are given below along with their domain.

$$\begin{aligned} type &\in APT, HOUSE \\ rent &\in any\, positive\, real\, value \\ city &\in TOR, YORK \\ bedroom &\in 1, 2, 3 \end{aligned}$$

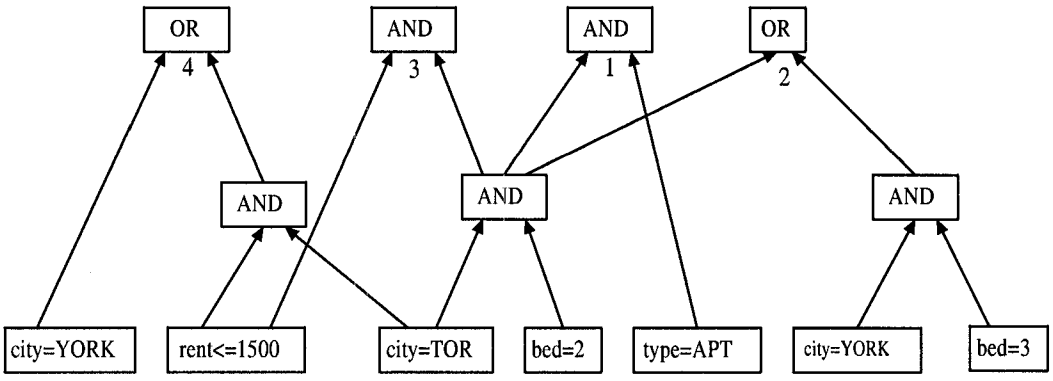


Figure 3.2: Taking advantage of common subexpressions

An event in the current context is comprised of a living place's information along with its type, rent, city and number of bedrooms. Now, we assume that we have four different subscriptions. First, S_1 represents the interest of a person who is looking for an apartment with 2 bedrooms in Toronto. Using boolean-algebra notation we define S_1 as,

$$S_1 \equiv (type = APT) \wedge (city = TOR) \wedge (bedroom = 2).$$

The second subscriber wants a two bedroom living place in Toronto or a 3 bedroom one in York. He does not care about the type or the rent of the place. This subscription can be defined as,

$$S_2 \equiv ((city = TOR) \wedge (bedroom = 2)) \vee ((city = YORK) \wedge (bedroom = 3)).$$

The third subscriber wants a place in Toronto with 2 bedrooms the rent of which is $1500.00 or below. We can write this as,

$$S_3 \equiv (city = TOR) \wedge (bedroom = 2) \wedge (rent \leq 1500).$$

The fourth subscriber is looking for any kind of living place in York without caring about its rent, or any living place in Toronto with rent equal or less than $1500.00. This subscription can be described as,

$$S_4 \equiv ((city = TOR) \wedge (rent \leq 1500)) \vee (city = YORK))$$

Figure 3.2 shows a tree structure that contains all of the above subscriptions. Some nodes in the figure are annotated with subscription numbers. Since each node can point to multiple parents, we are able to reuse common subexpressions. Let us assume that a new event is published in the system that describes the availability of a 2 bedroom apartment in Toronto with a rent of $1600.00. This event satisfies the subscriptions S_1 and S_2. While matching the event, the expression $(city = TOR) \wedge (bedroom = 2)$ is evaluated once, its value is stored at its node, and that value is used directly by the expressions defined in nodes for S_1 and S_2. We do not have to evaluate $(city = TOR) \wedge (bedroom = 2)$ twice, which saves computation time.

Though the example assumes some predefined predicates for the sake of simplicity, our tree structure is flexible and permits any predicate. While all the approaches require that all the attributes share the same semantics, the Gryphon approach also requires that every subscription use each attribute. Hence, all subscribers must be aware of all attributes. Our approach does not require a subscription to use all of the attributes. An

event also does not have to use all the attributes. However, there has been some research on how current publish/subscribe systems can be extended with semantic capabilities [29]. We allow the use of a predicate variable more than once, which allows range predicates in a subscription (see Section 3.2.3).

3.2.2 Evaluation of a Subscription

Our matching algorithm proceeds in two phases. The first phase, known as the *predicate matching algorithm*, determines all matched predicates in the system using all the components of the current event. The second phase, based on the results of the first phase, determines all the satisfied subscriptions. In this work, we focus on the second phase – the subscription matching algorithm. The first phase is performed as in other two-phased algorithms by managing lists of predicates per operator [15], by keeping tables of predicates, interval skip-lists [23], or interval trees [24]. The predicate matching algorithm implementation we use keeps linked lista of predicates per operator for predicates over infinite value domains and tables for predicates of small and finite value domains.

After having the list of matched predicates, we have to evaluate each subscription expression and find the matched ones. This constitutes the subscription matching. We can keep the list of all expressions and evaluate all of them one-by-one using a brute force approach. Obviously, it is extremely costly in terms of execution time and memory. The important fact to note here is that the more complex the subscription language is, the costlier the matching algorithm. For this reason, in the past, researchers had considered allowing only conjunctive predicates in the subscription to get a fast matching algorithm. In this work, we try to find a structure that gives us a matching algorithm with reasonable speed while allowing a highly expressive subscription language.

To determine whether a given subscription is matched by an event or not, we can choose one of two methods. Using a tree structure to represent a subscription has the primary advantage that it is possible to evaluate the whole expression by a post-order tree

traversal. Recursive descent parsing, using a simple *depth-first search* (DFS) traversal or using a stack, can evaluate any such expression. This is a top-down approach where the operation starts from the root of the tree. For any node, it traverses the left subtree and the right subtree and then it computes the value of the node. For any leaf, this value is defined by matching the corresponding predicate with the event. For any non-leaf, this value is computed according to its operator. The execution of such a process requires a stack structure and it involves accessing the data structure of a non-leaf node at least twice.

There can also be a bottom up approach to evaluate such a tree. Once we have evaluated all the leaf nodes, i.e., all predicates, the `true` or `false` value of a node is propagated up to its parent nodes. The function value for each non-leaf node will be evaluated when both its children have updated their values to it. The process continues for all the nodes, starting from the lowest level nodes in the tree. At the end of the traversal, the root nodes will have computed values in them – `true` of `false`. This approach is similar to a *Breadth-First Search* (BFS), which starts from the lowest level of nodes rather than from the root. BFS needs large queues that consume much memory. For executing a DFS, we need a stack which is less amount of memory compared to its bottom-up counterpart.

3.2.3 Advantages of a Tree Structure

Trees have been used as a natural representation for expressions for a long time. We foresee a few substantial advantages of using a tree-based representation in our context. First, a tree-based representation of a large set of expressions allows an effective reuse of subexpressions. In our context, there can be millions of subscribers submitting their interests in terms of Boolean expressions of predicates similar to arithmetic expressions of variables. It is apparent that instead of creating two separate subtrees for the same subexpression existing in two subscriptions, we can create one subtree for the first one, and

reuse that subtree for the second one. Thus, an effective subexpression reusing scheme can substantially reduce the run-time and memory cost. We show this in Section 5.8.

Second, the bottom-up traversal of the tree-based structure eliminates much of the unnecessary traversals. As we have described before, leaves in the tree contain predicates. In a real publish/subscribe system, the total number of predicates in the system can be quite large, but an incoming event will satisfy only a small fraction of the whole predicate population leaving a large fraction unmatched. So at a higher level of abstraction, we can imagine the tree structure with large number of leaves, but only a fraction of them satisfied by an event. The remaining leaves remain unmatched. In this situation, if we execute a top-down parsing, most of the time we will start from the root, traverse a number of non-leaf nodes and end up at a leaf node that is in unmatched state. So the traversal path to an unmatched leaf is a wasted stream of computations. Instead, we can start the traversal starting from the matched leaves and propagate the values upward to their parents and so on. This intuitively cuts down the cost of many extra node traversals. This may lead to many unmatched non-leaves due to the function specified in them (e.g. Boolean `AND`), but the total wasted computation will not be as large as in the top-down approach. So BFS, in our context, essentially has the advantage of visiting a small subset of nodes rather than visiting almost all of the nodes in the tree.

The tree structure defined in Gryphon has to use a fixed set of attributes as it uses each attribute variable at each level in the tree. Our proposed tree structure, on the other hand, stores predicates in individual leaves. Thus it is more flexible allowing any subscription to use any combination of attribute variables. An event also does not have to be confined within any fixed set of attributes.

The tree representation allows a subscription to have more than one predicate using the same attribute variable. For example, a subscription can contain two predicates, $p_1 = (price, \leq, 20,000.00)$, $p_2 = (price, \geq, 18,000.00)$. When we put a Boolean `AND` between these two, we get a ranged predicate. So our tree structure naturally permits

ranged predicates. More than two predicates also can be used in disjunction in our structure. For example, a part of a long complex subscription can be–

$$(model, =, Accord) \wedge ((model, =, Civic) \vee (model, =, CRV))$$

The non-leaf nodes describe any function on two results. It can be any Boolean or arithmetic operator. But we do not confine this to predefined operators. A non-leaf node can specify any user defined complex function or equation made up with two results from the two child-nodes. A unary operator, e.g., Boolean `NOT`, can be easily implemented using a non-leaf node, keeping one of its children empty, or resolving the higher level `NOT`s using De-Morgan's rule and then setting a negation operation on the predicates. It is easy to add a negation of a predicate because the relational operators we use in the subscription language can be reversed according to their logical meaning. For example, negation of $>$ relation means $\leq$. This solution, however, involves some amount of computation at the time of insertion of a new subscription. Hence, we only consider the first approach to implement unary operators.

Tree representation enables us to pipeline event filtering (see Section 4.4.1). Parallel processing can be used to achieve better throughput.

3.3 Aggregate Tree

This section discusses the Aggregate Tree structure in detail. The supplementary data stuctures that are required for the matching are also discussed in the following subsections. After introducing the basic tree structure, we discuss the hashtable structure used to locate a particular leaf node in the A-Tree using its predicate identifier. We also describe how we use a FIFO queue to help us in the traversal of a multi-stage graph at one stage of the algorithm.

3.3.1 Tree Structure

The *Aggregate Tree* (A-Tree) is basically a forest of binary trees. Each of the non-leaf nodes can have exactly two children, either of them can be a non-leaf or a leaf node. Unlike a conventional binary tree, a single node can have multiple parents (e.g. Figure 3.5). Each non-leaf node keeps track of the value associated with it, its list of parents, and two children, and some statistics needed for subscription insertion and deletion. An internal node N essentially represents a complex Boolean expression of predicates, E. So a node also contains a list of subscriptions, where all of them represent the same expression E. The node N can have multiple parents if all its parents contain the expression E within their own expression. This achieves the reuse of sub-expressions within the whole A-Tree structure. The list of nodes can be implemented using a list of statically allocated nodes, or by dynamically allocating nodes on demand. We experiment with both implementations.

Each leaf of this tree contains a unique predicate identifier, related to a predicate that is used in the first phase of the filtering engine. It has a field called `value` that can have either a value of zero or one depending on the first phase of the matching. If a new event satisfies the predicate in the first phase, this field will be set to one, otherwise it will be zero. Before processing any event, we will reset the value of each leaf nodes to zero initially so that we only set the value of the matched leaf nodes to one. After processing the leaf nodes, each of the matched leaves will be set to zero again. Hence, after filtering an event, it is not required to reset all the leaf nodes.

A non-leaf node will contain an operator, or a user-specified function. The two children provide the two operands required for the operation represented by a non-leaf node. A non-leaf node will have spaces to keep the values of its two operands and the result of the operation, as shown in Figure 3.3. The two fields, `operand1` and `operand2` will contain the two operand values required for the function of a node. Though the operand values can be either `true` or `false` for Boolean result, or some other numerical value for

different types of functions, we keep a separate value called `undefined`, which will let us know that an operand value has not yet been assigned. The major fields of a node are discussed below.

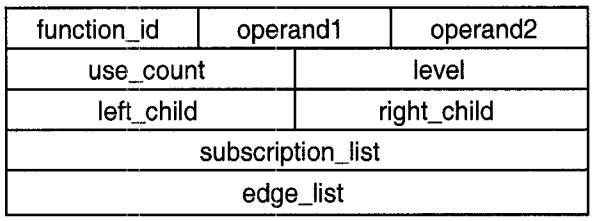


Figure 3.3: Major elements of a non-leaf node

use_count This is an integer field that indicates the number of subscriptions using the expression denoted by this node. For a new node this field is set to 1. But if a new subscription is added to the system afterwards and it happens to reuse any existing node α, we increase the `use_count` field of α by 1. Thus all nodes in the tree will have this field equal to or greater than 1. Again, in case of deleting a subscription, we simply decrease the `use_count` fields of the nodes that were being used by that subscription. While executing this process, the `use_count` field of some of the nodes may become 0. This indicates that those nodes are no longer being used by any subscriptions in the system. Those nodes can be safely deleted from the tree. So the main purpose of this field is to provide a very fast subscription deletion mechanism.

function_id This is an integer field that keeps the identification of the operation needed to be performed at this node. For example, if we allow Boolean operators, this can be a flag indicating `AND`, `OR`, `XOR`, `NAND`, etc. This field is valid only for non-leaf nodes. The main advantage and strength of our tree structure is that we can accommodate any operation and hence our event filtering algorithm can afford a very strong and versatile subscription language. This field actually can refer to a

user-specified function. When evaluating this node, we use its two operand values to compute the result of the function.

level This is an integer field to keep track of a node's level in the tree. It is the same level numbering concept for a binary tree but the concept is reversed for our tree. For any node α, this field keeps track of the distance between α and the farthest leaf among all the leaves within the subtree of α. It is set at the time of the creation of a node. An edge increases this value by 1. The leaf nodes will have this number set to 1. For any non-leaf node R, the `level` value is set according to the following formula.

$$level(R) = 1 + max\{level(P_i), \forall P_i | P_i \text{ is a direct child of } R\} \quad (3.1)$$

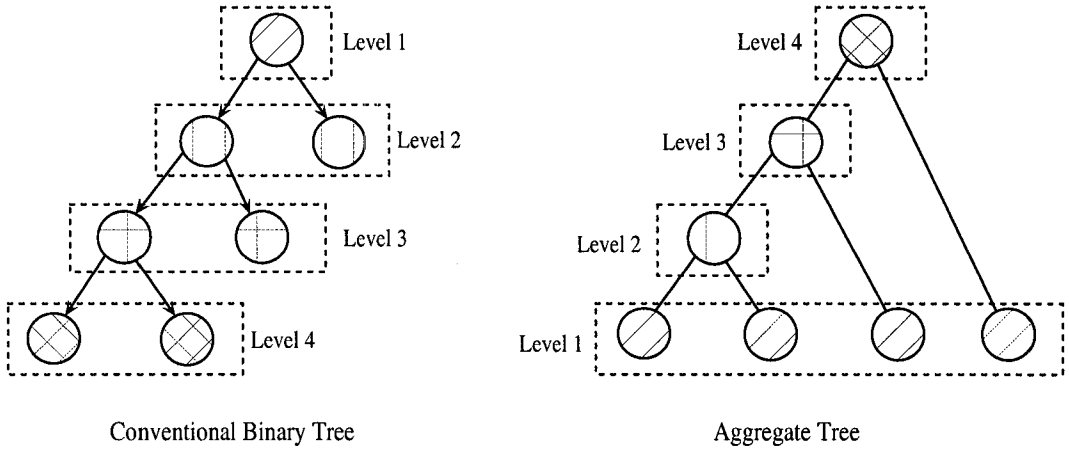


Figure 3.4: An example to illustrate the difference in level numbering scheme between a conventional binary tree and the proposed Aggregate Tree

Figure 3.4 shows two example trees where both of them are structurally the same but the level numbering scheme is different in our representation. For any non-leaf node *N1* in our tree, $level(N1) = 5$ can be interpreted as follows – within the set of descendant nodes of *N1*, there is at least one leaf node *N2* such that there are

4 edges in the path from *N2* to *N1*. In other words, from any other leaf node in the set of descendant nodes, it takes a maximum of 4 edges to reach *N1*. The significance of this attribute is that to get the two operand values ready in any non-leaf node *N1*, we have to wait $level(N_1)$ number of steps, where each step is the task of updating a node's value to its parent. A set of nodes *resides at* level L, if all of them have their `level` set to L.

According to the definition in Section 3.1, if a non-leaf node, R, has two leaf nodes as its two children, $level(R)$ will be 2. For example, R may refer to an expression like $A \wedge B$. It means that for the two operand values, R is dependent on two nodes residing in levels lower than 2. In general, if a non-leaf node R has its `level` field equal to n, it indicates that for the value of its two operands, R is dependent on two other nodes in the tree where each of them reside in a level lower than n. Therefore, all the nodes at level n can be processed without waiting further only if all the nodes of level $n-1$ have already been visited (see Theorem 4). While running the subscription matching part, this field helps us in executing a level ordered BFS using a number of *First-In-First-Out* (FIFO) queues (see Section 3.3.3).

edge_list This list keeps track of the nodes that are the immediate parent(s) to this node. It is alternately referred to as *parent_list*. It can be implemented in either an array or a linked-list structure, which keeps track of the pointers from this node to other nodes of the tree. It is worth noting here that a pointer from one node P to any other node Q means that as soon as we compute the result of the operation of P, that result will be propagated to any one of the two operand fields of Q. So P can point to any number of nodes, as a subexpression can be shared by many other expressions or subscriptions in the system. Hence, it might be wise to use a dynamic data-structure to keep track of the edge list for the purpose. On the other hand, it imposes some run-time cost for filtering, insertion and deletion of a subscription. According to the property of the field, `level`, in our tree structure,

if a node P points to a node Q, $level(P) < level(Q)$ (Equation 3.1).

subscription_list As described earlier, any node in our tree structure denotes a predicate or a boolean expression that contains predicates. Hence, any node in the tree might refer to a subscription. Furthermore, if two or even more subscribers in the system submit the same expression, we will end up having a single node referring to those two different subscriptions. So a node has to keep track of its subscription list. As it might vary in size depending on the predicate usage pattern of the subscribers, the list should be implemented as a dynamic list. This list might be empty for some nodes in the tree because the boolean expressions denoted by those nodes may not represent whole subscriptions but may be parts of the different subscriptions in the system.

3.3.2 Hash Tables

The predicate matching stage keeps track of all the predicates using unique predicate identifiers. In the second stage, for any predicate we need to locate the corresponding leaf node in the A-Tree using its predicate identifier. A hash-table, named *Predicate-to-Tree* (PT) is used for this purpose to get the fast access to such a relational table.

To keep track of the existing subscriptions in the system, we keep another hash-table, called a *Subscription-to-Tree* (ST) table, where each entry corresponds to a unique subscription S and keeps the reference to a single non-leaf node *root(S)* that represents the expression for S. As the edges are bidirectional in our tree, we can trace the set of all descendant nodes from the root. This is required when we need to delete S from the A-Tree.

3.3.3 Queues

For the main A-Tree algorithm, we use a number of *First-In-First-Out* (FIFO) queues. Each of them is designated to a specific level. For example, if Q_1 is the queue corresponding to level 1, it essentially means that Q_1 can only contain nodes residing at level 1 of the tree. So, generally, any queue Q_n can contain the nodes residing at the level n of the tree. Due to this property of the queues, we eventually have to keep a maximum M number of such queues where, M is the maximum value allowed for the field `level`, or the maximum number of operators allowed in a subscription. Having the above level marking scheme in the A-Tree, we ultimately transform our tree structure into a multi-stage graph, where our level numbers reflect the stage numbers. The leaves constitute the first stage of the graph, and the nodes at level M constitute the last one. The queues we described above simply enable us to do the traversal of a multi-stage graph including some of our own processing on each node.

3.4 General Filtering Algorithm

In this section we describe our filtering algorithm in detail. We first discuss the basic version of the algorithm. Using the same tree structure, we then describe two other variants of our original algorithm that achieve better run-time performances compared to the original one. However, those two optimizations have some limitations in other aspects.

When a new event comes into the system, the nodes of the tree are processed in a way similar to *Breadth First Search* (BFS). Before starting to process an event, the values of all the leaf nodes are set to 0 and those of the non-leaf nodes are set to an `undefined` state. The first phase of the filtering engine will match a set of predicates P_{true} for the current event. For each predicate in P_{true}, we identify the corresponding leaf node in our tree using the Predicate-to-Tree table described in section 3.3.2. The values of those leaf

nodes are set to 1. All these leaf nodes are in level 1. So they all are pushed into the queue Q_1.

After the initial steps, the queue processing algorithm proceeds in a conventional BFS fashion. At each step, the algorithm processes the queue corresponding to the lowest level. After having finished processing one queue, it starts processing the queue at the next higher level. The algorithm starts from Q_1. After completing a visit to all the nodes of Q_i, Q_2 is processed and the algorithm continues. For any queue Q_i, visiting each node α in Q_i involves three basic tasks, which are described in the following sub-sections. For simplicity, the steps that we describe here assume that we use only Boolean operators. But as mentioned before, our algorithm can support any kind of operator, and is not restricted to Boolean operators only.

3.4.1 Computation of the Result

The result of a node (α) is computed according to the operator defined in the node. To achieve this task, the two operand values for the current node must be available right at the point of visiting α. This is ensured by the level-order bottom-up traversal of the tree (Theorem 4). After computing the result, we propagate the result to all the parent nodes of α. However, in a variant of our original algorithm, known as the *Zero Suppression* scheme, (see Section 3.5.1) the computed value of the node α is only propagated upwards if the computed result is `true`. In this scheme, while computing the result of α, if one of the operands is undefined, it is assumed to be `false` and the result is computed accordingly.

3.4.2 Propagation of the Result

After computing the result of a node, α, it is propagated to all of its parent nodes. For each node β, which is a parent node of α, we first have to check the first operand field (operand1) of β. If it is undefined, it implies that the other descendant of β has not yet

updated its operand field, and hence the second operand of β is also undefined. It also implies that β is not in any queue. So the value of the first operand of β is set to the result of α, and β is pushed to the queue Q_j where j is the level of β.

If we find the first operand of β already set to some valid result, we know that β is already residing in Q_j. In that case, we set its second operand's value to the result of α and do not push β into the queue again. It might be worth mentioning that we push a reference to β into the queue, so only an update of the second operand field in this case suffices, no updates to the queue are necessary.

3.4.3 Notification of Satisfied Subscribers

After the necessary computation, if the result for a node α is `true`, using the subscription list of α all the subscribers are notified because they are all matched by the current event.

The pseudo-code for the general matching algorithm is shown in Algorithm 1.

3.5 Optimized Filtering Algorithm Variations

This section describes two variants of our vanilla A-Tree algorithm. The first variant, known as Zero Suppression only works for Boolean expressions. But it is a useful optimization which gives us significant benefits in terms of reduced execution time. The second variant, known as True Value Path Filter, eliminates the need for any FIFO queue, giving benefits in terms of memory requirements. However, this algorithm is also restricted to Boolean expressions, and can support a limited number of operators.

3.5.1 Zero Suppression Filter

The *Zero Suppression* optimization scheme, as discussed in Section 3.4.1, works only when we use a pub/sub system where the result of any expression is Boolean. So the result of any subexpression within a expression will be `true` or `false`. When the result of

Algorithm 1 General A-Tree Filtering Algorithm

Q[1..M]: queue that can hold nodes of the tree according to their level

```
current_level ← 1
while current_level < M do
  while Q[current_level] ≠ empty do
    α ← pop(Q[current_level])
    if α.operand2 = undefined then
      α.operand2 ← 0
    end if
    α.result ← execute_function(α.function_id, α.operand1, α.operand2)
    for all β|β ∈ {α.edge_list} do
      if β.operand1 = undefined then
        β.operand1 ← α.result
        j ← β.level
        push β into Q_j
      else
        β.operand2 ← α.result
      end if
    end for
  end while
end while
```

computation on any node is `false`, it is not propagated to any of its parents. Conversely, while computing the result of any node, if one of its operands is undefined, it is assumed to be `false` and the result is computed accordingly. Corollary 4 shows that this optimization is guaranteed to find all matched subscriptions. This optimization seems very promising because if we get a result of `false` for a substantial number of non-leaf nodes in the A-Tree while matching an event, the scheme saves a huge number of operations, such as value update to parents, pushing nodes to queues, etc. We believe this provides us a good optimization to reduce the run-time cost. The results of our experiments also demonstrate improved run-time performance.

The primary drawback of this optimization, as we have discussed before, is that we are confined to Boolean operators only. This is because of our assumption that any undefined value at a node indicates a result of `false`, and this can only happen if we are using Boolean operators. So it is not possible to optimize our proposed Constraint-based Subscription Language (Section 4.2.1) or Partial Match-based Subscription Language (Section 4.2.2) using this scheme.

An Example of Zero Suppression Filter

We will describe the execution steps of the zero suppression filter using Figure 3.5. We allow only Boolean operators and Boolean values as the matched results. We assume that we have the following different subscriptions using the predicates A, B, C, D and E. The subscriptions are

$$
\begin{aligned}
S_1 &= (A \wedge B) \oplus (C \wedge D) \\
S_2 &= B \vee (C \wedge D) \\
S_3 &= A \wedge B \wedge C \wedge E \\
S_4 &= A \wedge B \wedge C \\
S_5 &= C \wedge D
\end{aligned}
$$

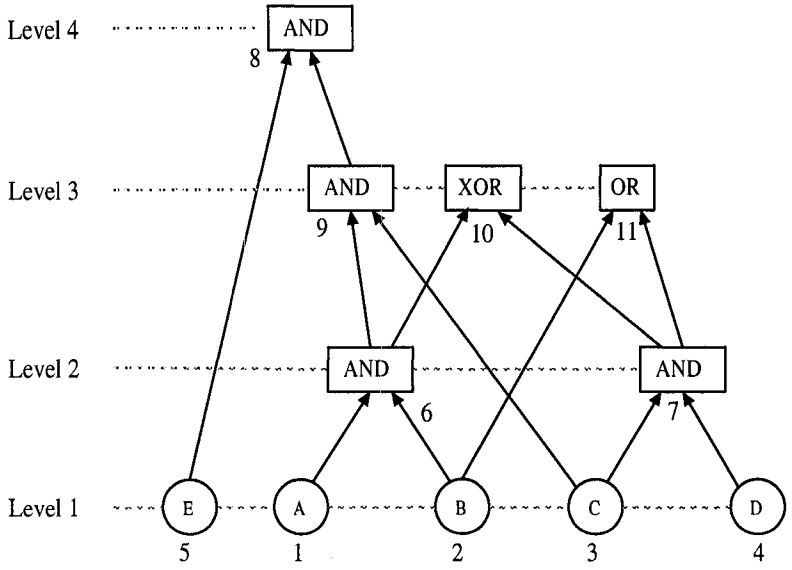


Figure 3.5: Matching algorithm example

$$S_6 = A \wedge B \wedge C$$

$$S_7 = C \wedge D$$

Table 3.1 shows the node to subscription mapping for all the subscriptions we currently have in the system.

Node number	Subscription
10	S_1
11	S_2
8	S_3
9	S_4, S_6
7	S_5, S_7

Table 3.1: Node to subscription mapping table

Since the A-tree has four levels, we will use four queues. Before starting the matching algorithm, all of those queues will be empty. Now, we assume that a new event has come

into the system that satisfies the predicates A, B and C. So the algorithm will execute and try to find the matched subscriptions as described in the following steps. For the description, we assume that each node has two fields *operand1* and *operand2* to keep the two operand values for its operation.

- Nodes 1, 2 and 3 will be pushed into Q_1, which is the queue designated for nodes of level 1. All these nodes have their result set as 1 or *true* because each of them directly represents a predicate that has already been satisfied by the incoming event. Nodes residing in Q_1 are processed one by one.

- First, node 1 is taken out from Q_1 and its only parent is node 6. So its result value of 1 (equivalent to logical value *true*) is assigned to the field *operand1* of node 6. As node 6 is not currently in any queue, it is pushed to Q_2, as level of node 6 is 2. Now, node 1 is removed from Q_1.

- Node 2 is popped out from Q_1, its value is assigned to *operand2* of node 6 and *operand1* of node 11. Node 11 is pushed into the queue Q_3. It is worth noting here that since we already pushed the reference to node 6 to the queue, Q_2, we can only update its second operand and the value in the queue will get updated automatically.

- Node 3 is popped out from Q_1, its value is assigned to *operand1* of node 9 and node 7. Node 9 is pushed into Q_3 and node 7 into Q_2 according to their level value.

- Now Q_1 is empty, which indicates that we are done with all nodes at level 1. Next, we process the nodes from Q_2. So Node 6 is popped out from Q_1. It has both its operands as 1 and the result of the operation specified at this node (boolean `AND`) is 1. This value is assigned to *operand2* of node 9 and *operand1* of node 10. Node 10 is pushed into Q_3.

- Node 7 is popped out from Q_2. It has *operand1* set to 1 but *operand2* is undefined as node 4 did not pass any value to this node. Now, according to the Zero Suppression scheme (Section 3.5.1), *operand2* is assumed to be 0. So the result of the operation `AND` defined at node 7 is 0. According to zero suppression scheme, this value will not be passed to the parent nodes 10 and 11.

- At this point, Q_2 is empty. So we start with Q_3. First, node 11 is popped from Q_3. It has *operand1* set to 1 but *operand2* is undefined, hence 0. The function defined at this node is `OR`, so the result is 1. Node 11 contains the subscription S_2, so S_2 is satisfied by the incoming event.

- Node 9 is popped from Q_3. It has *operand1* and *operand2* set to 1. Function defined at this node is `AND`, so the result is 1. This value is assigned to *operand1* of node 8, it is pushed to Q_4. As node 9 contains the subscriptions S_4 and S_6, they are satisfied.

- Node 10 is popped from Q_3. It has *operand1* set to 1 but *operand2* undefined, hence 0. The function defined at this node is `XOR`, so the result is 1. As node 10 represents the subscription S_1, it is satisfied. Q_3 is now empty, so we start processing the last queue, Q_4.

- Node 8 is popped out from Q_4. It has *operand1* set to 1, but *operand2* is implicitly 0. So the result of the `AND` operation is 0.

After the completion of the above steps, we have found that the incoming event has satisfied the subscriptions S_1, S_2, S_4 and S_6.

3.5.2 True Value Path Filter

The second variant of our original A-Tree algorithm does not use any queue. This reduces some amount of the total required memory, compared to the previous two algorithms.

This variation assumes Boolean operators `AND` and `OR` as functions in the non-leaf nodes and Boolean values as matched values.

We assume that each event in pub/sub has a unique event identifier (e.g., a unique integer value). An event e *touches* a node α if, while matching the event, any operand of α is set by any of its children. Intuitively, it can be seen that e can touch α 0, 1 or 2 times within the total execution period of the matching algorithm, depending on the results of the two children of α.

For this algorithm variant, we introduce a new field, `current_event_id` in each node α to keep the identifier of the most recent event that touched α. For any matched predicate, we find its corresponding leaf node, say θ. The result of θ is definitely `true`. We propagate this `true` value to all of its parents. The important point here is that for any node we compute the result and propagate only the `true` value to all of its parents, in exactly the same way as the Zero Suppression scheme. Before updating the `true` value in any node α, which is a parent of the node θ, we first compare the `current_event_id` field of α (say v_1) with the event identifier of the current event (let v_2). If we find that $v1$ is older than $v2$, we overwrite $v1$ with the current event identifier, $v2$ and assign `true` to the first operand of α. We also set the result of α to `true` without waiting for the other operand's value if its function is Boolean `OR`. We recursively propagate the result to all of its parents in the same way. If the function is Boolean `AND`, we do not propagate the value. We wait until the other child of θ updates the second operand value to this node.

If we find $v1$ and $v2$ to be equal, it definitely indicates the first operand of α is already set to `true` by one of its children. So we assign `true` to the second operand of α, and compute the result. If the function of α is `AND`, the result will be `true`, and it is recursively propagated to all of its parents. If the function, on the other hand, is `OR`, we do not perform any computation for this node and do not propagate the value any further. Since α was touched by the current event already, we can safely assume that the value was already propagated upward at that time.

It is important to note here that there can be some nodes left in the tree touched once by the current event, but the result was not computed on the first access. All of these nodes have Boolean `AND` as their function, hence the computed results for them are implicitly `false`. Their `current_event_id` are set to the current event's identifier, but it will be eventually be set to a new value by the next event.

From the above description, we see that the upward propagation of `true` value stops when two specific conditions occur. The basic idea, as we have discussed, is to propagate the Boolean value of `true` to as many nodes as possible starting from a leaf-node. We run a recursive DFS starting at every matched leaf node. We continue along one path of `true` propagation until we cannot do that anymore, because we reach a node with a value of either `false` or unknown. We define a *true value path* (TVP) as a sequence of nodes $n_1, n_2, \ldots, n_k$ where

- n_1 is a leaf node,
- n_{i+1} is a parent node of n_i, where $1 \leq i \leq (k-1)$,
- the computed value of the node n_i is `true`, where $1 \leq i \leq (k-1)$,
- the computed value of the node n_k is `false` or `unknown`.

The correctness of the True Value Path algorithm has been proven in Corollary 5. This algorithm exploits the advantage of the Zero Suppression scheme, but additionally it reduces the overhead of queues by traversing the true value path starting from a leaf node. The general filter algorithm and the zero suppression filter use a large queue with an enormous number of queue operations. On the other hand, the true value path filter has the overhead of running the recursive depth first traversal. We have compared these three algorithms to uncover the potential of each of them in Chapter 5.

The drawback of the TVP optimization is similar to that of the zero suppression optimization. The TVP algorithm supports only Boolean expressions, like the zero suppression algorithm. However, an additional drawback is that TVP can allow only the

Boolean operators `AND` and `OR`. However, it is easy to implement any complex Boolean function using these two operators. The other drawbacks of the zero suppression filter also prevail in this algorithm. Algorithm 3 shows the pseudo-code of the TVP filtering algorithm.

Algorithm 2 procedure True_Value_DFS(θ)

```
θ : [input parameter] current node with result true
e : current event
e_id : identifier of current event
{update true value to all parents}
for all α|α ∈ {θ.parent_list} do
  if α.current_event_id = e_id then
    if α.function_id = AND then
      α.result ← true
      call True_Value_DFS(α)
    end if{don't do anything for OR}
  else
    α.current_event_id ← e_id {set the new event's id to α}
    α.operand1 ← true
    if α.function_id = OR then
      α.result ← true
      call True_Value_DFS(α)
    end if{don't do anything for AND}
  end if
end for
```

Algorithm 3 True Value Path Filter

matched_list ← set of non-leaf nodes matched by the current event

for all $\theta | \theta \in \{matched\_list\}$ **do**

call True_Value_DFS(θ)

end for

3.5.3 An Example of True Value Path Filter

We demonstrate the execution steps of the TVP algorithm with a simple example. Figure 3.6 shows the A-Tree. For simplicity, we only show the execution steps in traversing the tree and computing the results of each of the nodes, and ignore the subscriptions and their expressions. We assume that for a new incoming event, leaf nodes 1, 2, 3 and 5 are satisfied, i.e., they contain a value of `true`. We start with each leaf separately.

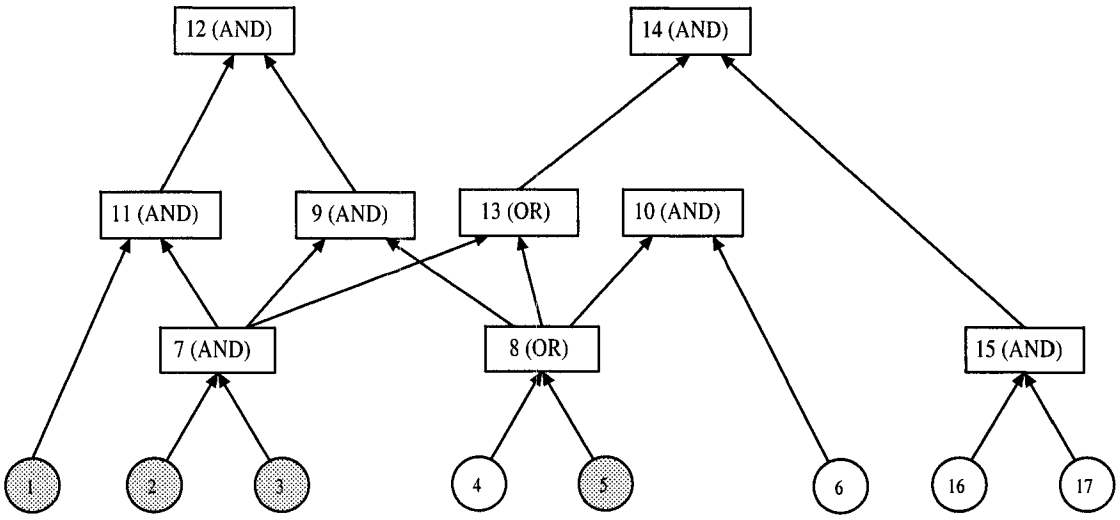


Figure 3.6: An example of TVP filter algorithm

- Starting from node 1, we propagate `true` to node 11. The event touches 11 for the first time. Node 11 has an `AND` operator, so the traversal stops here. We start the execution with node 2.

- In the same way as before, from node 2, we only update the `true` value to node 7,

and stop there.

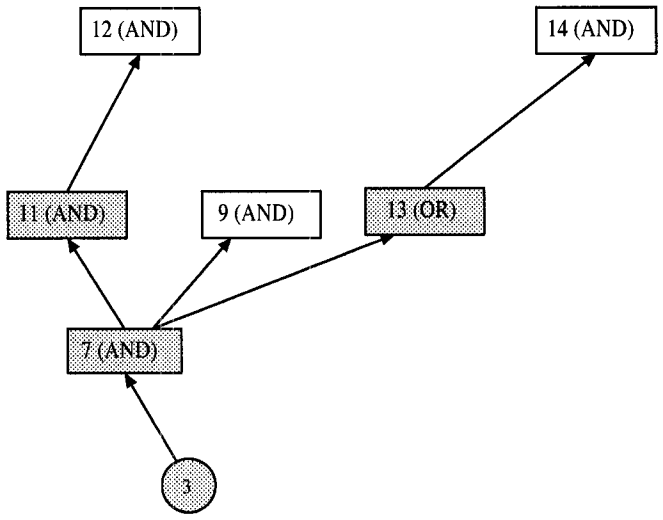


Figure 3.7: All true value paths starting from the node 3

- Starting from node 3, we update the `true` value to the second operand of 7, as the first one already exists. We compute the result of 7, which is `true`, propagate it to 11, taking the first path to its parent. Subsequent calls to DFS will update values to the other parents of 7, namely nodes 9 and 13. In the same manner, we propagate the value of `true` from node 11 to node 12, but the path stops there as node 12 does not have the other operand's value as yet. Now, the same true value path (TVP) traversing is performed for the nodes 9, and 13. For node 9, the path ends right at 9, since the value of the other operand for this node is still unknown, and the operation at this node is an `AND`. For node 13, as it contains an `OR` operation, we compute the result – `true` right away, and propagate that to node 14. Figure 3.7 shows the TVPs found starting from the leaf node 3. Nodes, set to `true` by this traversal are highlighted with a different shade.

- Starting from 5, we update the `true` value, according to the DFS order, to the nodes 8, 9, 12, 13, and 10, respectively. For node 13, this `true` value is the second

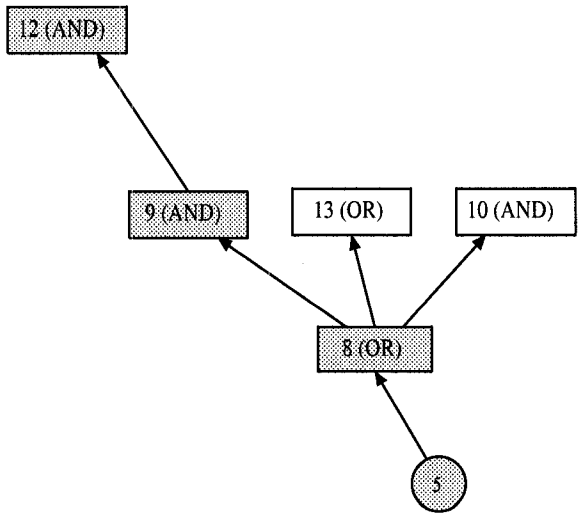


Figure 3.8: All true value paths starting from the node 5

operand, but as it has an `OR` as its operator, so we do not propagate `true` from 13 to 14, since it was already done earlier. This ensures the traversal of a TVP only once. Figure 3.8 shows the TVPs starting from the leaf node 5.

It is worth noting here that, in the example above, nodes 10 and 14 have one of their operands set to `true`. As their second operands were not assigned a value of `true`, they never reach a matched state. Eventually, in future, a new event might overwrite their operand1 fields, deleting the stale value.

3.6 Maintenance Algorithms

Besides filtering, deletion and insertion of subscriptions into the data structures are two crucial operations of the algorithm. Especially, insertion is important, since it controls the degree of overlap achieved among differed subscriptions and thus directly influences the matching performance. The two optimizations for the general algorithm described before do not require any changes in the maintenance algorithms.

3.6.1 Deletion of a subscription

Deleting a subscription is a straightforward and fast operation. The entry of the subscription, S, to be deleted, is looked up in the ST table. This table points to the root node N_S used by S. In our node structure, we have an integer field `use_count` (Section 3.3.1). This indicates the total number of subscriptions using this node in the whole tree. Now, all the nodes existing in the subtree rooted at N_S are used by S. We trace all of them by a simple *DFS* starting from N_S. For each of those nodes, its `use_count` field is decremented by 1. If the `use_count` becomes zero for any node, it can be safely deleted, as it is not shared by any other expression in the tree.

Algorithm 4 Deletion Algorithm

sid : id of the subscription that is to be deleted

table_entry ← get_entry(sid,ST table)

$\alpha \leftarrow nodereferencefrom$ table_entry {remove the references of this subscription from all nodes}

call Delete_Node (α) {remove the entry with sid from ST table}

delete the entry (sid, θ) from ST table

3.6.2 Insertion of Subscriptions

Insertion of a new subscription is the most crucial part of the filtering algorithm. A better insertion algorithm will always result in a better tree structure and more efficient filtering in terms of memory and run-time cost savings will be achieved.

The key point behind an optimal insertion of a subscription is the number of existing nodes we can reuse from the A-Tree structure for this purpose. A subscription is an expression tree over atomic predicates, where the leaves of the tree denote the predicates and the non-leaf nodes denote the binary operations over two operands. Now, the inser-

Algorithm 5 procedure Delete_Node(θ)

θ : current node to be deleted, as parameter

α : left child of θ

β : right child of θ

decrease $\theta.use\_count$ by one

call Delete_Node (α)

call Delete_Node (β)

if $\theta.use\_count = 0$ **then**

  delete θ from the tree structure

end if

tion process starts with the leaves of this tree. Within the subscription tree, we try to find all syntactically valid sub-expressions, and determine an optimal representation of these sub-expressions in the given A-Tree structure. In this context, we define optimality in terms of number of nodes reused from the existing A-Tree structure. So the most optimal representation is the one that reuses the maximum number of nodes from the existing structure. Optimality, however, is not strictly confined to the number of nodes reused. We can easily define optimality with some other metric such as access frequency of predicates, etc. The insertion of a subscription does not change the structure of the existing A-Tree, rather we rearrange the new subscription and find the best possible representation for it. The insertion order of subscriptions has obvious effects in the structure of the A-Tree generated using this approach. Hence, the quality of the A-Tree depends on the insertion order of the subscriptions.

An expression over Boolean operators can be reorganized if all its non-leaf nodes contain the same operator. By reorganizing a sub-tree, we try to determine the best representation, given the current A-Tree structure. For example, if we consider a new subscription or a sub-expression in a new subscription to be $(A \wedge B \wedge C \wedge D)$, we can arrange the whole expression in 5 different ways.

$$(((A \wedge B) \wedge C) \wedge D)$$
$$((A \wedge (B \wedge C)) \wedge D)$$
$$((A \wedge B) \wedge (C \wedge D))$$
$$(A \wedge (B \wedge (C \wedge D)))$$
$$(A \wedge ((B \wedge C) \wedge D))$$

So if we try to insert this expression into the A-Tree, the best representation should be the ordering that reuses the maximum number of existing nodes from the A-Tree.

The main cost of our insertion algorithm depends on the number of ways we can rearrange an expression. This is exactly the classical problem of finding the number of possible binary trees that can be formed from N distinct leaf nodes. The number is known as the *Catalan number* [20] and its value was mathematically proven to be $\binom{2(N-1)}{(N-1)}/N$, which is exponential. A single subscription in a typical pub/sub system will not consist of more than 12 - 15 predicates. If we consider a moderate rate of subscription insertion and we want a high insertion rate, this overhead might pose a challenge.

To achieve a high rate of insertion, we can cast the problem of finding an optimal arrangement for a sub-tree as a classical dynamic programming problem. Here, all the sub-expressions in the subtree will be associated with a cost value. For the sake of simplicity, we can assume this cost to be 1 if that sub-expression exists in our tree structure as a node, or 0 if it does not. A more optimized but complex cost model can be used considering additional statistical information gathered from the tree. In the algorithm, we have overlapping sub-problems to solve. To describe this formally, we assume that we want to rearrange an expression with n predicates (a full subscription may consists of several such expressions.) We first define a few terms used in the discussion.

- $T_{i,j}$ is the sub-tree containing the predicates from i to j as its leaves, where $1 \leq i < j \leq n$.

- $root(T_{i,j})$ is the root node of the tree $T_{i,j}$.

- $TCost(T_{i,j})$ is the total cost to build the tree $T_{i,j}$.
- $cost(n_1, n_2, op)$ is the cost to create a new node having the operator op, and two children n_1 and n_2 in the A-Tree. This function can be defined based on the context (e.g., it could account for a more complex cost model). For simplicity, we can assume that this term is 0 when there already exists such a node; otherwise it is 1, indicating that we need to create one new node for the tree, $T_{i,j}$.

There can be many different possible representations for $T_{i,j}$. We can find the sub-tree structure with minimum cost by solving the following recurrent relation,

$$TCost(T_{i,j}) = \begin{cases} min_{i<k<j}\{C_L + C_R + C'\} & \text{if } i < j \\ 0 & \text{if } i = j \end{cases}$$

$$\begin{aligned} C_L &= TCost(T_{i,k}) \\ C_R &= TCost(T_{k+1,j}) \\ C' &= cost(root(T_{1,k}), root(T_{k+1,n}), op) \end{aligned}$$

Here, n, is the number of predicates in the subscription. So, in a bottom-up approach, we try building all the optimal solutions with the number of leaves starting from 2 to n, where n is the number of leaves (i.e., predicates) for the subtree corresponding to this subscription. So we compute $TCost(T_{i,j})$ for all possible values of i and j. Finding the optimal arrangement of the sub-tree that gives us the minimum cost has the run-time cost of $O(n^3)$. This indicates that our algorithm can handle a fairly high rate of insertions. While computing the optimal tree structure for a subscription, we keep a table of nodes existing in the A-Tree that can be reused when we insert the subscription.

To insert a subscription S, we first locate the *rearrangeable* sub-trees within its representation, and run our rearrangement algorithm on those sub-trees. After getting the optimal representation for all of them, we make the optimal arrangement S_T. We insert the subscription starting from the leaf nodes that represent the predicates of the subscription. If some of the predicates are new to the system, we add new leaves for them.

Algorithm 6 procedure Rearrange_Subtree(S_T)

S_T: subtree to be arranged, as parameter

Map: table to keep track of parent child relations between three nodes

$m \leftarrow$ number of leaves in S_T

for $p = 1$ to $m - 1$ **do**

  $i \leftarrow 0$

  while $i + p < m$ **do**

    $j \leftarrow i + p$;

    $gc \leftarrow \infty$;

    $T_{opt} \leftarrow nil$;

    for $k = i$ to $j - 1$ **do**

      $T_1 \leftarrow$ subtree having the predicates [i,k]

      $T_2 \leftarrow$ subtree having the predicates [k+1,j]

      $T_3 \leftarrow$ subtree combining T_1 as left, T_2 as right subtree

      $tc \leftarrow$ cost for building T_3

      if $tc < gc$ **then**

        $gc \leftarrow tc$

        $T_{opt} \leftarrow T_3$

      end if

    end for

    save the relations of the root of T_{opt} to Map

  end while

end for

set S_T to optimal structure using Map

Algorithm 7 procedure Update_Predicate_Table(S)

S: subscription to be inserted, as parameter

{insert the new predicates to A-Tree}

$L_1 \leftarrow$ list of predicates in S

for all predicate P_i in L_1 **do**

  if no entry for P_i in PT table **then**

    insert a new leaf node α in the A-Tree

    insert (P_i, α) entry in PT table

  end if

end for

Then we proceed to the next level. If any two leaf nodes in S_T have a common parent node, we look into the A-Tree structure if we have the common parent node. This is done by matching the two `parent_lists` of the two nodes. If we do not have such a node, we add a new node to the A-Tree. Otherwise, we just increase the `use_count` field of the parent node by 1. This process proceeds upward for all the nodes in S_T. After this is done for all the nodes in S_T, the reference to the root node in A-Tree for S is inserted in the hash-table, ST, along with the subscription identifier of S. The rearrange and the insertion algorithms are formally presented in Algorithm 6, Algorithm 7 and Algorithm 8, respectively.

3.7 Properties of A-Tree Matching

In this section, we state a few properties of the A-Tree matching algorithm that have explicitly and implicitly been used in the above presentation. Formal proofs of these properties are presented in the Appendix.

Algorithm 8 procedure Insert_Subscription(S)

```
S: subscription to be inserted, as parameter
call Update_Predicate_Table(S)
L_2 ← list of subtrees in S that are rearrangeable
for all subtree T_i in L_2 do
  call Rearrange_Subtree(T_i)
end for
S_T ← rearranged subtree for S
m ← number of leaves in S_T
for i = 1 to m − 1 do
  for all two sibling nodes α and β in level i do
    α_T ← A-Tree node corresponding to α, β_T ← A-Tree node corresponding to β
    op ← boolean operator in S binding α and β
    if no node θ exists in A-Tree binding α and β with op then
      create a new node θ in A-Tree, insert θ in the parent_list of α and β
      set α and β as the two children of θ, set use_count of θ to 1
    else
      increase use_count of θ by 1
    end if
  end for
end for
sid ← subscription id of S, θ ← root node of S in A-Tree
insert the (sid, θ) entry in ST table
```

3.7.1 Tree Consistency and Deletion/Insertion Safety

Due to our new level identification scheme, we can state a few important properties about the A-Tree structure that make the A-Tree a consistent structure. By the term consistent, we mean that every internal node of the A-Tree will have exactly two children. This consistency is preserved after insertions and deletions of subscriptions from the tree. Therefore, A-Tree will remain a forest of binary trees after any number of insertions and/or deletions.

The following two theorems state level preserving properties. M is the maximum level (see Section 3.3.1) and L denotes the level number.

Theorem 1. *If there exists at least one node at level L ($1 < L \leq M$) in the A-tree, there must be at least one node at level $L - 1$.* □

Corollary 1. *No level L in the A-tree can be empty, given $1 \leq L \leq M$.* □

If M is the highest level in the A-Tree, any level between 1 to M is non-empty. This ensures that no matter how many insertions and deletions we do, we will always have nodes in each level of the tree. We prove these properties formally in Appendix A.

If we were to delete a node in the tree, it may still point to child nodes, which would then be dangling and thus result in an inconsistent tree structure, but according to Theorem 2, the deletion safety property, this cannot happen.

Theorem 2. *If α and β are two nodes in the tree such that β is a child of α, $use\_count(\alpha) \leq use\_count(\beta)$.* □

The field `use_count` of any node can never be greater than that of its parent nodes. So it helps us to decide when it is safe to delete a node completely from the A-Tree. When we delete a subscription's reference from a node, we decrease this field. So if for any node, `use_count` gets set to 0, we can safely delete it.

Finally, the consistency of the A-Tree structure is captured with the following Theorem.

Theorem 3. *Any non-leaf node in the tree will always have exactly two children.* □

In summary, this means that any non-leaf node in the A-Tree will always have exactly two children, no matter how many subscriptions we insert or delete in the system.

If a node θ is the parent of two nodes α and β in our A-Tree structure, these properties ensure the following important points in our algorithm.

1. Due to our level ordered queue operations, by the time we take θ out of a queue to compute the result of it, we already have accessed the nodes α and β and computed their results in some lower level queues. So two operands for θ are always ready.

2. The insertion and deletion algorithms keep all the properties of A-Tree consistent.

3.8 Implementation

The A-Tree algorithm is implemented in C for performance reasons. The tree data structure is an array of nodes. We experiment with two different approaches to implementing the tree. In the first approach, which is fully static, we declared a static array large enough to hold the maximum number of nodes needed for the system depending on the expected maximum number of subscriptions. In the second approach, nodes are allocated on demand. However, rather than allocating nodes one by one, node allocation is done in batches, to amortize memory allocation cost. Node deletion is managed in both approaches with a free-list of nodes. When a new node is to be allocated, it is first drawn from this free-list.

The `parent_list` of any node is a list of references to all its parent nodes. It is a sorted list. This list plays an important role while finding an expression in the A-Tree. To do that, we need to determine whether two nodes have a common parent, with the

operator of that parent node matching the operator of interest. So, if we are to find $A \wedge B$, and we know that the nodes corresponding to the expressions A and B in the A-Tree are α and β, respectively, we need to find a node θ such that it contains the operator $\wedge$ and it is the parent of α and β. So we need to consult the union of the parent list of α and that of β, and check each node of the resulting set to see if it contains the operator $\wedge$. As the `parent_list` is a sorted list, we can do this union in $O(n)$ operations. An alternative would be to keep a hash-table of all different expressions represented in the A-Tree and then lookup the node reference in that table. That hash-table would consume an extremely large amount of memory, as the table would have a significant number of entries. The size of the table would also reduce the speed of searching in the hash table. Considering the fact that expression lookup is only needed in case of subscription insertion, and we assume to have a lower rate of insertions compared to the rate of incoming events, we decided to trade off the extra run-time benefits for a reduced amount of memory required.

The queues are *First-In-First-Out* (FIFO) queues, implemented as static arrays. Each queue manages the reference to nodes only, so that queue operations are not costly. These queues are not required for the true value path filter.

Chapter 4

Flexibility in A-Tree Matching

As mentioned earlier, the A-Tree structure is extremely flexible in many aspects. In this chapter, we discuss some powerful subscription languages we can accomodate easily using this structure. We did not implement these completely, but it will not be difficult to get these implemented for evaluation with some minor modifications to our existing data structures. We also discuss potential enhancements to our existing algorithms.

4.1 Priority-based Subscriptions

Priority-based subscriber notification is an important functionality for many applications of publish/subscribe systems, where the relative importance of various matching subscriptions has to be distinguished to realize different quality of service levels for subscribers. This can be naturally supported with the A-Tree matching algorithm by introducing different priority levels for subscriptions. The semantics of a subscription priority is that if the priority of a subscription, S_1, is higher than that of subscription S_2, S_1 should be notified no later than S_2, assuming that we are considering an event that matches both S_1 and S_2. To make our A-Tree structure capable of handling priorities on subscriptions, we can add a priority field in the node structure. As a node θ is used by a set of subscriptions Sub_θ, the value of this field will be the maximum priority of all subscriptions

in Sub_θ. This ensures that all nodes in the A-Tree used by a subscription S will have a priority equal to or greater than the priority of S. Intuitively, a subscription with the highest priority value P_{max} will form a subtree within the A-Tree, where all of its nodes are marked with priority P_{max}.

Figure 4.1 gives an example. We assume that at the very beginning we insert a subscription, $S1$, with priority value 2 that uses predicates $r1$, $r2$, $r3$, and $r4$. So all these four leaf nodes will be set to priority 2. Based on $S1$'s subscription expression, we created new internal nodes a, b, and c with priority value 2. The root node a keeps track of $S1$. Now a new subscription $S2$, with priority value 3 that is higher than $S1$'s priority, is inserted. $S2$ uses the predicates $r3$, $r4$, and $r5$. So we change the priority of $r3$ and $r4$ to 3. We create a new leaf $r5$ with priority 3. We then find that to insert $S2$, we have to reuse the node c, and create a new node d. So we set the priority of c to 3, create d with priority 3, and mark $S2$ referring to the node d.

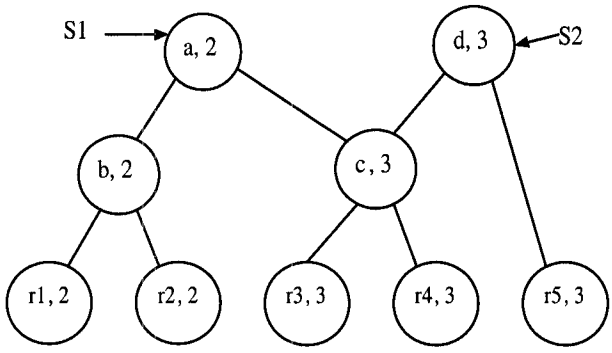


Figure 4.1: Priority Subscription example

To delete a subscription, we follow our usual deletion algorithm. But after deleting the reference of a subscription from a node, we have to adjust the priority of that node. Note that the priority of a node is actually the highest priority value of all of its parents. We delete the references of a subscription starting from the root node. So for any node, adjusting the priority is simply getting the highest priority value from its list of parent nodes. For example, in Figure 4.1, if we want to delete $S2$, we can delete d and $r5$

directly. For the node c, we delete its reference to the parent d, and then find the highest priority value from its list of parents. Here, c has one parent left – a. So we set the priority of c to equal to that of a, which is 2. For $r3$ and $r4$, we do the same adjustment.

For priority-based matching, we associate a priority label with each queue and introduce multiple queues for a specific level, in contrast to the one-queue-for-one-level scheme we used earlier. So now a queue $Q_{i,j}$ will contain only the nodes from the level i who have a priority value of j. While filtering an event, we will push a node to its appropriate level and the corresponding priority marked queue. While processing the queues, we will first process all the queues with the highest priority mark. After this, we will start processing the queues with the second highest priority mark, and so on. We can do this without any conflicts, due to the properties of our A-Tree structure. So, after finishing this process for all queues of priority P_k, we have evaluated the subtree with all the nodes having the priority value equal to or greater than k. This process will also forward some results for the nodes with priority levels less than k.

The number of different priority levels is restricted by the number of queues we initially define. We cannot have subscriptions with arbitrary priority. Our system, depending on the implementation and system resources, will always be able to handle a maximum of m levels of priority. However, even if the priority value has a large range, we can divide the range into m number of ranges and assign the priority values 1 to m to each group.

Another important fact is that if a subscription, S_i, has a higher priority than S_j, it will be guaranteed that S_i will be notified earlier than S_j only if the expression for S_i does not occur as a subset of the expression for S_j. Consider, the case of $S_1 \leftarrow A \wedge B \wedge C \wedge D$ and $S_2 \leftarrow B \wedge C$. If an event satisfies S_1, it will also satisfy S_2. So before we get to know that S_1 is satisfied by the event, somewhere in the process we will get to know that S_2 is satisfied. So we cannot guarantee the priority will be reflected during notification in this case.

4.2 Generalized Subscription Languages

We define two new types of subscription languages that provide powerful ways of expressing user interests. For both of these, we need to extend the notion of a *predicate match* slightly. We specify that a predicate is *matched* when an event satisfies it as before, but add a return value to the match. So, when a predicate P = *(pvar, op, pvalue)* is evaluated by an event $E = (evar, evalue)$, we get a pair of values as the result. The Boolean field *match_state* tells us whether the event matched the predicate and the field *return_value* may be a function of the three variables *evalue*, *pvalue*, and *match_state*. Based on the application context, this function can be specified appropriately. For example, we consider a predicate $P = (price, <, 200)$, and two events $e_1 = (price, 170)$ and $e_2 = (price, 170)$. The function for the second field is defined as,

$$\begin{aligned} return\_value &= evalue \mid \text{match\_state=}\texttt{true} \\ &= undefined \mid \text{match\_state=}\texttt{false} \end{aligned}$$

For the first event, the result is $(true, 170)$. For the second one, the result is $(false, undefined)$.

Based on this small extension, we introduce two novel and more powerful types of subscription languages in the following two sections that can be easily implemented based on A-Tree.

4.2.1 Constraint-based Subscription Language

For any subscription, instead of using Boolean operators to bind the predicates, we can define any other operator to gain a more powerful subscription language. For any two predicates, we can define any complex function using the two results of the predicates as operands. For simplicity, we only show the case of arithmetic operators to bind predicates. For any operation between two predicates, we will use the two values of the field *return_value* of the two predicates as the two operands. So if a subscription S has k predicates, $(p_1, p_2, \ldots, p_k)$, the formal expression of the subscription will be,

(p_1 op_1 p_2 op_2 ... op_{k-1} p_k) op_k V. Here, all op_i are binary operators of any type, and op_k is a relational operator, V is a numeric value. This generalized form allows the user to put constraints on the individual predicate variables as well as impose a restriction on the result of the whole expression.

Here we consider an example of a user who is interested in buying three items A, B, and C. The user is concerned about the total price, but also wants to put restriction on the individual prices. We define the event match semantics as outlined above. So the subscription looks like $S : (p1 + p2 + p3) \leq 1000$, where $p1 : (price_A \geq 200)$, $p2 : (price_B \geq 100)$, $p3 : (price_C \geq 600)$. The + operator here is the usual arithmetic addition. The user restricts each price, but wants to get notified if the total price is less than or equal to $1,000.00. It is easy to see that there are many combinations adding up to the total sum. But the user explicitly emphasizes that $p1$, $p2$ and $p3$ must hold. So an event $e_1 = (price_A, 300), (price_B, 170), (price_C, 500)$ will not match this subscription, even though the total value of three components remains below 1000.00, as e_3 does not match p_3. In the same way, an event $e_2 = (price_A, 300), (price_B, 170), (price_C, 630)$ will not match S due to its failure to satisfy the total expression, though it satisfies all of the individual predicates. Event $e_3 = (price_A, 240), (price_B, 120), (price_C, 630)$ will match S, as it meets all the constraints.

In the A-Tree structure, we can easily accommodate this kind of subscription. To do this, we can redefine the matching semantic in our system and include the notion of a return value. In the internal nodes, instead of processing a boolean function, we can process any function based on the return values of the child nodes. In the root node of each subscription, we set the result according to the relational function defined in the subscription. We can achieve this without losing any other properties and advantages of the A-Tree. Overlapping sub-expressions within this type of subscription population can still be reused efficiently following the usual tree construction process.

We can also define powerful operations on the matched result of predicates, for ex-

ample *average*, *max*, *min*, etc. that can be combined with the above specified constraint-based subscription language. We assume that *avg* is the notation for the average function, and define a subscription $S : (avg(price_A, price_B) \leq 1000) \wedge (avg(price_C, price_D) \leq 1500)$. To handle this type of subscription, we have to define a function for *avg* in non-leaf nodes. For two arguments, the non-leaf node will simply add the two operands and divide it by 2 to get the average. However, if *avg* takes more than two arguments, we can replace the function with a summation (*sum*) function, and in the top level node we can define *avg* function such that it divides the total accumulated value by the number of arguments. In Figure 4.2, the top level node represents the expression $avg(A, B, C, D) \leq 1000$.

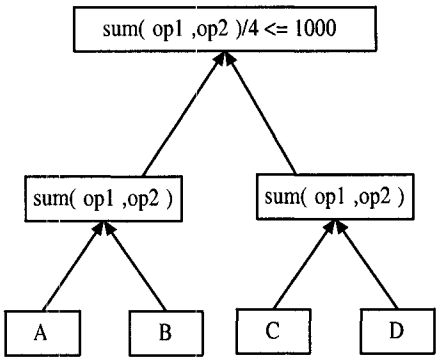


Figure 4.2: Subscription example with avg function

The functions *max* and *min* are even easier to handle. If they are used as a function taking two arguments, we can directly set these functions in non-leaf nodes. But if anyone of them, for example *min*, takes more than two arguments, we can get the equivalent expression by using multiple two argument *min* functions. For example, if a subscription is defined as $S : min(A, B, C) > min(D, E, F, G)$, we can rewrite it as

$$S : min(A, min(B, C)) > min(min(D, E), min(F, G))$$

We can easily represent the subscription S in a binary tree structure (Figure 4.3), and insert it in our A-Tree. Function *max* can be used in exactly the same manner. These simple yet powerful functions will offer a much more powerful publish/subscribe paradigm.

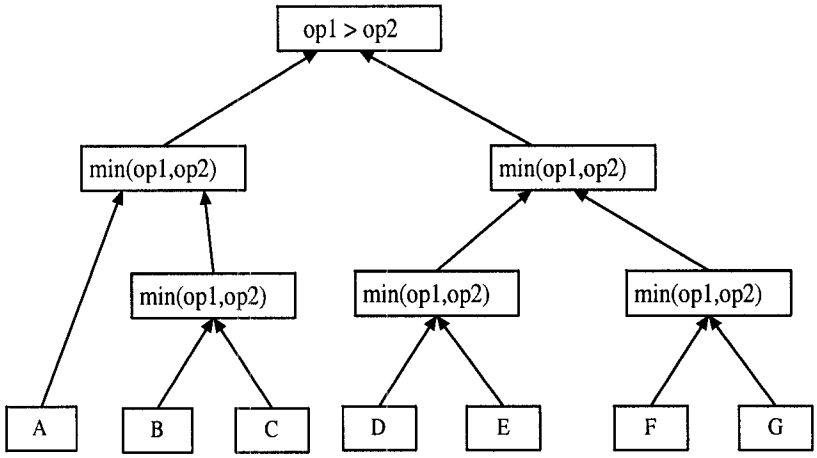


Figure 4.3: Subscription example with min function

As all the return values make up the final result, we cannot apply the Zero Suppression filter or the True Value Path filter that avoid the propagation of `false` in the Boolean case.

4.2.2 Partial Match-based Subscription Language

A subscription *matches partially*, if k of its n predicates match. This scheme adds further flexibility for the expression of subscriber interests. The notion of a partial match can be realized by counting the number of predicates matched for a given subscription and prematurely stopping, if the threshold for a subscription has been reached. Similar to the implementation of a constraint-based subscription language, the partial match paradigm can also be implemented by simply adding return values to a predicate (i.e., 1 for match and 0 for no match) and comparing the result to k.

4.3 Composite Events and Subscriptions

Composite events [17] are important concepts, especially for network management application contexts and have not received much attention in the publish/subscribe literature. A composite event instance is formed by a temporal or causal combination of event instances. For example, a sequence operator. $E_C = (e_1; e_2)_t$ means that if the event e_1 is satisfied before e_2 with the time difference t (i.e., the event e_2 occurs after t time units have passed since the occurence of e_1), the composite event E_C is satisfied. In the context of publish/subscribe, this could be cast into a subscription – a *composite subscription* is also concerned with the temporal state of the matched subscription expressions. For example, a subscription S_C can be defined as $(s_1; s_2)_t$, where s_1 and s_2 are conventional expressions comprising predicates. The semantics of such a subscription is that the subscriber wants to get the notification only if there occurs two events e_1 and e_2 with a time difference of t such that e_1 satisfies s_1 and e_2 satisfies s_2.

Composite subscriptions can use the general sequence operators, for example $\prec$, between two predicates or two expressions in a subscription. These operators are functionally the same as the previously described sequence operators but the latter lack the time difference parameter. For example, a subscription $S_2 = (p_1 \prec p_2)$ is satisfied if an event e_1 matches p_1 and another event e_2 matches p_2 where e_1 arrived before e_2. If we keep the event identifier field `current_event_id` in the node structure, described in Section 3.5.2, we can access the event table to check the arrival time of an event. If an event matches p_1, we keep the event's identifier in the `current_event_id` field of the node α that contains the expression for S_2. If another event matches p_2, and we reach α again, we can check the arrival time of previous event and the current one to decide whether S_2 is satisfied. In the same manner, A-Tree can easily handle $\succ$, $\succeq$, $\preceq$, etc.

The TVP filter can fit this type of event matching perfectly. This optimization is substantially faster than the other two. We keep track of each event's identifier in the A-Tree. We can access the arrival time of the event by the event's identifier. Using

this field, we can easily match composite events with different operators in our A-Tree structure.

We have discussed how we handle the temporal and causal sequence operator with the A-Tree. Composite event and subscription model provides further operators, which are beyond the scope of our current discussion. Our only objective is to highlight the flexibility of the A-Tree matching algorithm.

4.4 Parallel Execution

In this section, we briefly provide a guideline regarding the parallelization of our A-tree algorithm. We have not been implemented a parallel version, but the discussion provides a potential direction for future work.

4.4.1 Pipelining of Events

Functionally, the filtering process works like a multi-stage graph traversal proceeding from the lower stages to the higher ones. This traversal can directly be used for a pipelined processing of events with the A-Tree structure (e.g., realized with multiple threads). All queues are used once for each event, so, after processing all the nodes of queue Q_i, the queue is ready to accept new nodes for the next event. By using some primitive locking mechanism at queues to avoid multiple access, we can concurrently start processing another event using the free queues. If any queue is found to be locked by the previous event, then the concurrent execution has to be kept waiting until that queue is released. For a large number of predicates in each subscription, this approach can provide some extra event processing throughput.

4.4.2 Parallel Execution of Two Phases

As we discussed earlier, ours is a two-phase algorithm. In our current implementation, the second phase, subscription matching part, waits for the first phase to find all the matched predicates. When the first phase passes the complete list of matched predicates, the second phase starts its operation. Thus these two stages work synchronously. This is a potential place to get the throughput increased substantially. The first phase, predicate matching engine, returns the matched predicates one by one. So upon getting the first one, we can start our true value path (TVP) filter, as this algorithm can start executing the DFS function as soon as it gets a matched predicate. So the TPV filter can run in parallel with the execution of the first phase. In future, we plan to deploy this on a multi-processor system to observe the performance results.

Chapter 5

Evaluation of Proposed Algorithm

In this chapter, we discuss our experimental setup and the performance results of our main algorithm and the two variations. For comparison, we also ran a brut-force approach, the counting algorithm [15], and the Gryphon algorithm [1]. All of these are our own implementations. Due to the extreme memory requirements and slow execution time of the brute-force approach, we do not include its results in our comparative discussions.

The remainder of this chapter is organized as follows. Section 5.1 describes our experimental framework in detail. The next section describes our implementation of the workload generator. Section 5.3 describes the metrics we have used for our performance measurements. Section 5.4 gives the experimental results when only conjunctive predicates are used. This section gives a comparative study among all the different algorithms, since all of them support a conjunctive workload. In the next two sections (Section 5.5 and Section 5.6), a mixed workload is considered. Since the other algorithms developed to date do not support such a workload, we only perform a comparative study among the vanilla A-Tree algorithm and its two other variants.

5.1 Experimental Setup

We ran all experiments on a dual-CPU Pentium III workstation with an i686 CPU at 900MHz and 1.5GB RAM. The operating system was Linux (RedHat 7.2). We have created a separate load generator decoupled from our matching engine. All the subscriptions and the events are stored in disk files. Every matching engine uses our own parser to read the subscriptions and events from the input files. We kept the matching engine process separate so that the overhead of the workload does not influence the run-time performance. It also gives us more space in terms of memory while running any of the matching engines.

5.2 Workload Generator

We use a synthetic workload generator to generate the predicates, and combine them to create the subscriptions and the events. This section describes the implementation details of this workload generator.

5.2.1 Workload Parameters

We have used multiple parameters so that we can control the nature of the workload as required. Table 5.1 shows some important parameters that can be used to tune our workload generator. The table contains ranges for each of our parameters, but for any specific experiment, we change only two or three appropriate parameters keeping the others fixed. We define the term *Shared-Nodes-Per-Subscription* (SNPS) that is the average fraction of total nodes of a subscription shared with other subscriptions in the tree. Let us show a simple example to demonstrate what this term actually means. Let us assume that every subscription has 16 predicates. So a tree representation of a subscription will need 16 leaf nodes and 15 non-leaf nodes (a total of 31 nodes). A value of 0.36 (or 36%) for *SNPS* indicates that on an average every subscription

Parameter	Description	Range
N_{subs}	number of subscriptions in the system	100,000 - 4,000,000
L	number of predicates per subscription	4 - 18
L_{max}	maximum number of predicates in the system	20
N_E	number of events to be filtered	100 - 5000
A_E	number of attributes in an event	L - L_{max}
$f_{overlap}$	fraction of total subscription that is not unique	0.9 - 1.0
SNPS	average fraction of internal nodes of a subscription shared with other subscriptions	10% - 80%
$freq_k$	probability of occurrence of the k-th operator	0 - 1
SMPE	average fraction of total subscriptions matched per event	5% - 80%
l_{A_i}	lower limit value for i-th attribute variable	1
u_{A_i}	upper limit value for i-th attribute variable	2,000,000

Table 5.1: Workload parameters

has its $31 \times 0.36 \approx 11$ nodes (both internal nodes and leaves) shared with some other subscriptions in the system. This parameter actually controls the degree of overlap we get in the subscription population. If we set *SNPS* to 1.0, we will get all of the nodes of all subscriptions shared with other subscriptions.

5.2.2 Predicate Generation

Basically in our workload generator, we generate two sets of subscriptions, $Subs_U$ and $Subs_N$. In $Subs_U$, each subscription is unique in terms of its expressions. By a unique subscription, we mean that it does not have any of the predicates or subexpressions within its expression shared with any other subscription. $Subs_N$ has all non-unique subscriptions, so each of the subscriptions in this set has one or more predicates and/or subexpressions shared with some other subscriptions of this set. The sizes of these two

sets are controlled by the parameter $f_{overlap}$.

$$size(Subs_N) = N_{subs} \times f_{overlap}$$
$$size(Subs_U) = N_{subs} \times (1.0 - f_{overlap})$$

We keep a pool of attribute variables, and for each one we define its own range of values. We also define a set of relational operators for predicates. We generate a large pool of predicates using our finite set of predicate variables. Depending on the other parameters, we generate a set of predicates SP_k for each $k-th$ attribute variable. We can either randomly choose a relational operator from the set $(=, \leq, <, \geq, >, \neq)$, or fix the relational operator to $=$ or $>$ for all predicates. This is required since our implementation of Gryphon algorithm can handle only the equality operator in predicates.

For the k-th attribute variable, we choose the value of each predicate in the set SP_k within the range l_{A_k} to u_{A_k}. We can use two policies regarding the selection of a value. In the first case, we can select the value randomly, so all the predicates are evenly spread out within that range. In the second case, we can keep a global table for frequency distribution of values for the variable k, and we can select the value accordingly. This would make the pattern of the predicates in the set SP_k similar to a real-life scenario where it is more likely that a large group of users will be interested in a very small range of values for a specific variable whereas very few users will be interested in the values outside of that range. However, we do not have the second policy implemented yet, so we use the first policy for all our experiments.

5.2.3 Subscription Generation

After the predicate generation phase is over, we generate each subscription depending on our parameters. To control the overlap, we first create two pools of expressions A and B. All of the subexpressions in any of these two sets is unique within that set. So in any set, we do not have two expressions that are the same, or even two expressions that

have some portion of their subexpressions in common. The number of predicates of each expression found in A or B is set according to our global parameters.

Operators that bind the predicates are selected according to their probabilities. We currently allow three Boolean operators – `AND`, `OR` and `XOR`. We use three variations of operators with three different sets of frequency distributions for the operators. Table 5.2 shows the three configurations. The conjunctive operator set (OS_A) is used when we

OS_A	conjunctive workload	$freq_{AND} = 1.0$
		$freq_{OR} = 0.0$
		$freq_{XOR} = 0.0$
OS_B	conjunctive and disjunctive workload	$freq_{AND} = 0.8$
		$freq_{OR} = 0.2$
		$freq_{XOR} = 0.0$
OS_C	mixed workload	$freq_{AND} = 0.6$
		$freq_{OR} = 0.3$
		$freq_{XOR} = 0.1$

Table 5.2: Operator Sets

compare Gryphon and counting algorithm to the three versions of our alogorithm. The two other types are only used with our matching engine, as no other algorithm can handle these types.

To generate an expression for any of the sets A or B, we first generate an empty binary tree with some specific number of leaves. The tree structure is also randomly selected over all possible forms. After that, we fill up the internal nodes with operators selected according to their frequency distribution, and we fill the leaves with the predicates chosen from our set of predicates. We carefully select predicates so that we do not put two conflicting predicates in the same expression. For example, we ensure that logically unsatisfiable expressions like $(price \leq 100) \wedge (price \geq 200)$ never occur.

To generate a subscription, we pick two expressions from the sets A and B, and bind them together using a root-level operator. We pick one expression from A if we want to reuse it in the future. So the size of A controls the number of subscriptions using a common subexpression. The other set B is intentionally kept very large based on the value of N_{subs}, so that if one subexpression is randomly picked from this set, there is a very low probability that it will be picked again in future. So, to generate a unique subscription, we pick two expressions from set B and bind them together. However, generating a non-unique subscription can be done in two different ways. We can pick both expressions from A, or pick one from A and the other from B. Depending on the parameters $f_{overlap}$ and *SNPS*, we decide how frequently we should follow the first or the second way. For example, to generate a high amount of overlap in the subscriptions , we use the first way.

5.2.4 Event Generation

The events are generated according to the parameter *SMPE*, which denotes the average number of subscriptions matched per event. We need to get a specific number of subscriptions matched on average after filtering a large number of events. So a value of 25% for *SMPE* means that on average, for each event we get 25% of all the subscriptions satisfied. We control this by adjusting the value of each attribute in an event. For an attribute variable a_x, we know that we used the range l_{A_x} to u_{A_x} to generate the pool of predicates having the variable a_x. Based on the predicate population in different areas of that range, we set the value of a_x in the event and expect a corresponding change in the number of matches. However SMPE cannot be precisely set in our workload generator before event generation. The final value of SMPE varies in the order of 5% or so. At the end of the experiment we measure this parameter again from the results obtained after filtering all events.

The number of attribute variables in each event A_E is controlled based on the number

of predicates of each subscription. For the conjunctive workload, A_E is set to a random number within the range L to L_{max} (Table 5.1). For the two other types of workload (Section 5.6 and 5.5) we use the range 1 to L_{max}.

5.3 Measured Metrics

This section describes the major metrics we used for our performance measurements.

5.3.1 Insertions per Second

We divide N_{subs} by the total time required to insert N_{subs} subscriptions, and get the number of insertions per second, or in other words the throughput of insertion. This metric indicates the overhead for maintaining the system. For timing measurement, we collect two timestamps using the system call `getrusage` at the point when we have no subscriptions, and at the point when we finish inserting all the subscriptions. We then calculate the difference between these two. While measuring the time, we consider both user level time and system level time consumed by the current process. For any insertion, we did not count the time required to parse a subscription from the input files. We followed the same method to compute other throughput metrics.

5.3.2 Deletions per Second

We measure the total time to delete N_{subs} subscriptions, divide N_{subs} by that time and get the number of deletions per second. This is also a metric related to the maintainability of the system.

5.3.3 Events per Second

The most important metric we look for in a matching engine is the number of events it can process in one second. As we have a set of N_E events ready to be filtered by an

algorithm, we measure the total time to filter all of these events. This time includes the first stage i.e., predicate matching time, for counting and A-Tree based algorithms. We divide N_E by the total time required to match all events, and get the number of events filtered per second.

5.3.4 Memory Consumed

For each of the algorithms, we measure the memory consumed by considering the total size of all the data structures while having all the N_{subs} subscriptions residing in the system. We rounded the computed number of bytes to the nearest megabyte.

5.4 Conjunctive Workload

For conjunctive workload, we have run all the algorithms with number of subscriptions N_{subs} starting from $10,000$ to $4,000,000$. We choose 11 different values of N_{subs} in this range. For each N_{subs}, we selected 4, 6, 8, 10, 12 and 14 as number of predicates (L) in different input sets. Due to space limitation, we show the results with 6, 10 and 14 predicates in Figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 and 5.9. For each pair of values of N_{subs} and L, we generated 1000 events (N_E) and filtered them by all algorithms. Depending on the configuration, for high value combinations of N_{subs} and L, N_E was decreased a little to reduce the total execution time. *SNPS* was kept within the range 35% to 40%.

The most important aspect of this workload is that it only contains predicates with equality checks i.e., only $=$ is used as relational operator in each predicate. This is done as our implementation of Gryphon can handle only equality predicates. For this reason, an incoming event matches small fraction of the total subscription population compared to the system where we allow $>$ or $<$ operators in predicates. So we had to keep *SMPE* within the range 0.1% to 1.0%. However, we think that in real-life scenarios, *SMPE*

would be well above this range.

For the Gryphon algorithm, the predicates have to be ordered. We therefore keep all the predicates ordered in this workload and used a fixed number of attributes for each subscription. For A-Tree-based algorithms, it is not necessary to have ordered predicates or to use all of the predefined set of attribute variables.

For each of the algorithms, we submitted the same set of N_{subs} subscriptions, filtered the same set of events, and measured each of our performance metrics.

5.4.1 Performance Results

Insertion Throughput

The A-Tree algorithm has obvious computational overheads while inserting a subscription due to the cost incurred in finding the optimal representation for it. It needs a search operation to find an expression using the two parent lists of two nodes. The counting and Gryphon algorithms have no such overhead. We show the insertion throughput for 6, 10 and 14 number of predicates in each subscription, respectively in Figure 5.1, 5.2 and 5.3.

In Figure 5.3, we see that the insertion rate for A-Tree is above 5000 for 100,000 subscriptions. It linearly decreases with the increase in the number of subscriptions. A-Tree performs relatively poorly in terms of insertions. But the numbers show that A-Tree is a manageable structure even for a high number of subscriptions. Figure 5.1 shows that in spite of the complex subscription input processing, the A-Tree still gives us an insertion rate of over 1,000 subscriptions per second with a very high workload. The rate is not much affected with the increase in the number of subscriptions or predicates. The counting algorithm is able to achieve a high rate of insertions because a subscription insertion in this algorithm simply involves the task of setting some entries of a row in a matrix to 1. Although we see a high insertion rate at low number of subscriptions for

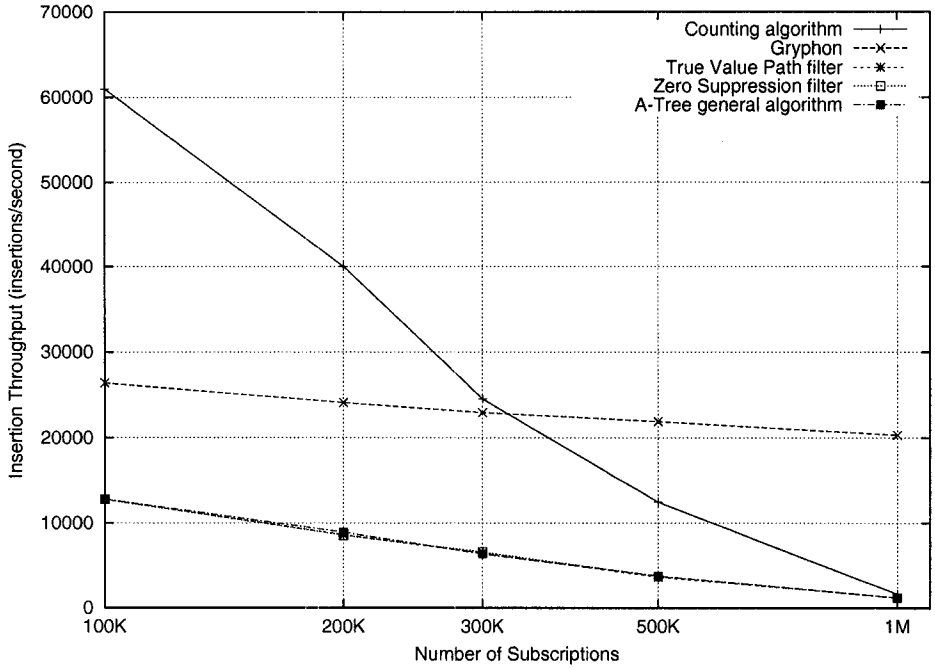


Figure 5.1: Insertion throughput with 6 predicates

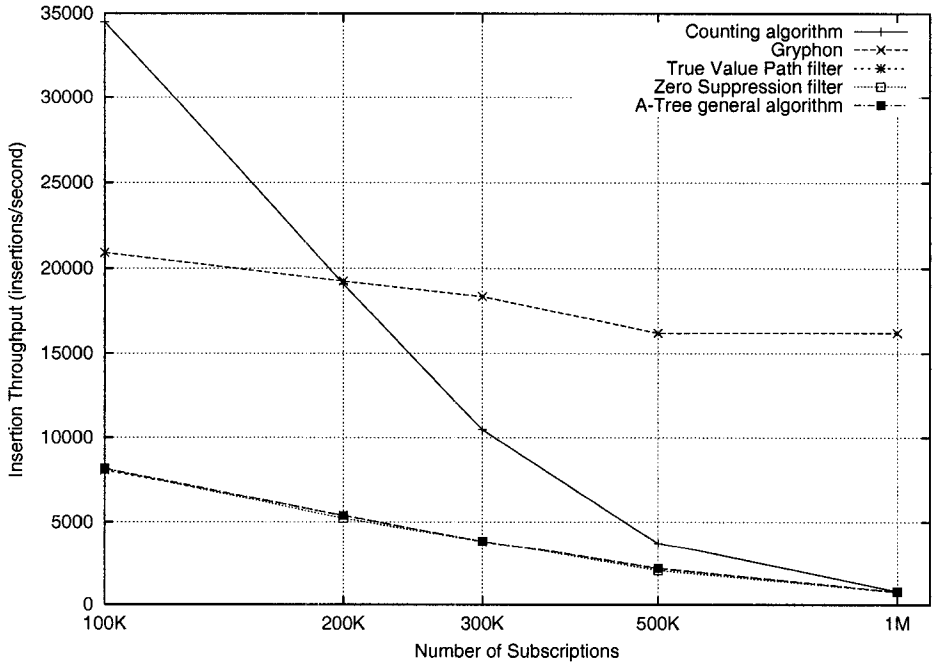


Figure 5.2: Insertion throughput with 10 predicates

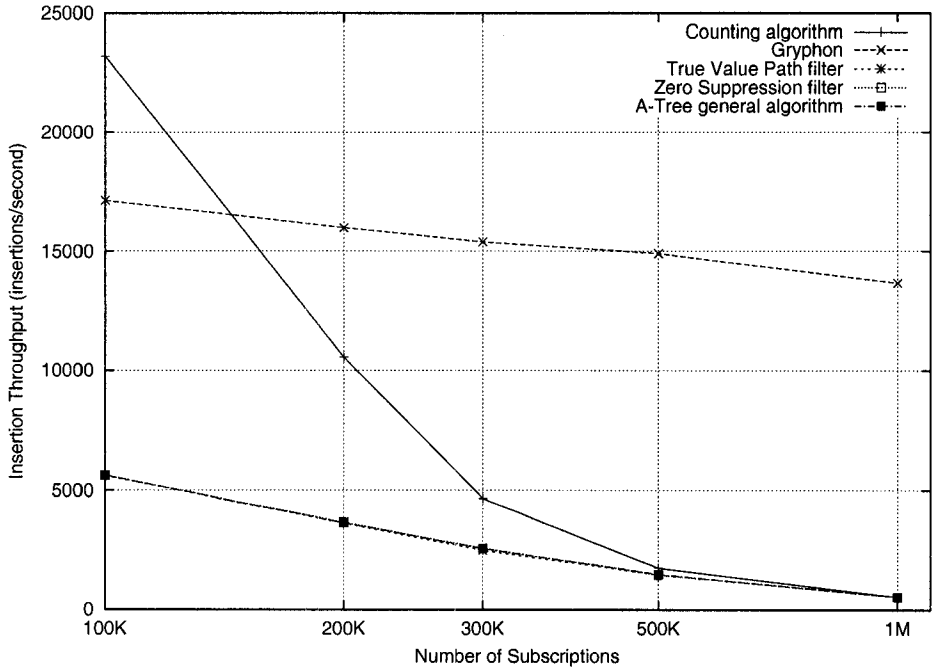


Figure 5.3: Insertion throughput with 14 predicates

counting algorithm, it exponentially goes down as the number of subscriptions increases. Gryphon has a higher rate of insertion compared to all other algorithms, and the rate is affected very little by the increase in the number of subscriptions. The A-Tree algorithm and two variations of it have same insertion throughput because insertion algorithm is same for all of them.

Event Matching Rate

The counting algorithm gives a very high rate of matching compared to the Gryphon and the A-Tree algorithm. The counting algorithm has a very high throughput (more than 350 events per second) with 100,000 subscriptions. The rate goes down quickly but it is still the best upto 400,000 subscriptions. After 500,000 subscriptions, Gryphon starts showing higher event throughput than the counting algorithm. Event throughput results of the A-Tree algorithm and its two variations show that they do not perform as good as Gryphon or the counting algorithm. This is due to the fact that for each internal node, we

have to compute a function value. We kept it modular so that it can accommodate any kind of function. This overhead adds up for all nodes visited while filtering an event. On the other side, the counting and Gryphon do not perform any function on the matched predicates as both of them assume conjunctive subscriptions. Both these algorithms are highly optimized for the conjunctive operator. Counting algorithm shows a very high event matching rate around 100,000 subscription population. But the rate goes down fast as soon as the number of subscription increases. In Figure 5.4, 5.5 and 5.6, it is interesting to observe that with higher number of subscriptions, e.g., with over 1 million subscriptions, all the variants of the A-Tree algorithms perform almost equally well as the counting algorithm. The other important observation here is that the increase in the number of subscriptions has a very little effect on the event matching rate of Gryphon and our algorithms. If we compare the similar curves of counting, we see that it rapidly goes down with the increase in number of subscriptions. Gryphon performs very well with the increase of subscription population. We expect A-Tree to perform as good as the counting algorithm if the number of subscriptions is very high, like 5 or 6 millions. We could not execute the experiments for such high number of subscriptions due to resource constraints.

Another interesting observation to be made in this experiment is that the TVP filter performs exceptionally well in terms of event matching rate in the range of subscriptions 100,000 to 500,000. It also performs slightly better for 1 million or more subscriptions. Hence, we think this optimization is a very good choice in situations where it is applicable.

In Table 5.3 we show the event matching rate for the brute force algorithm.

Memory Consumption

The memory overhead of A-Tree is quite modest. At lower subscription population, it requires comparatively high amount of memory, but it increases very little with the increase in number of subscription. Compared to Gryphon, it always takes more amount of

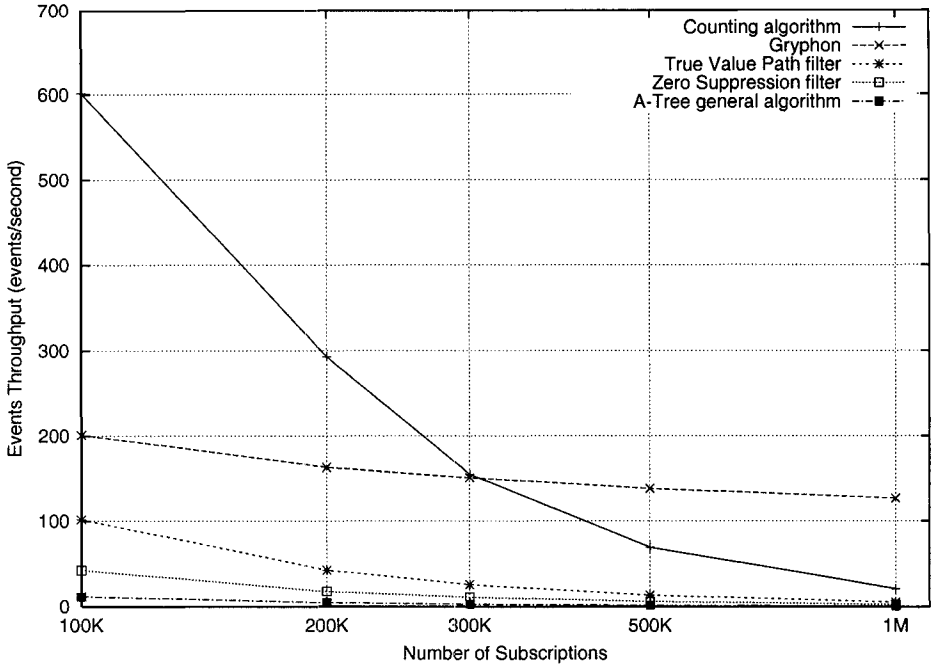


Figure 5.4: Event matching throughput with 6 predicates

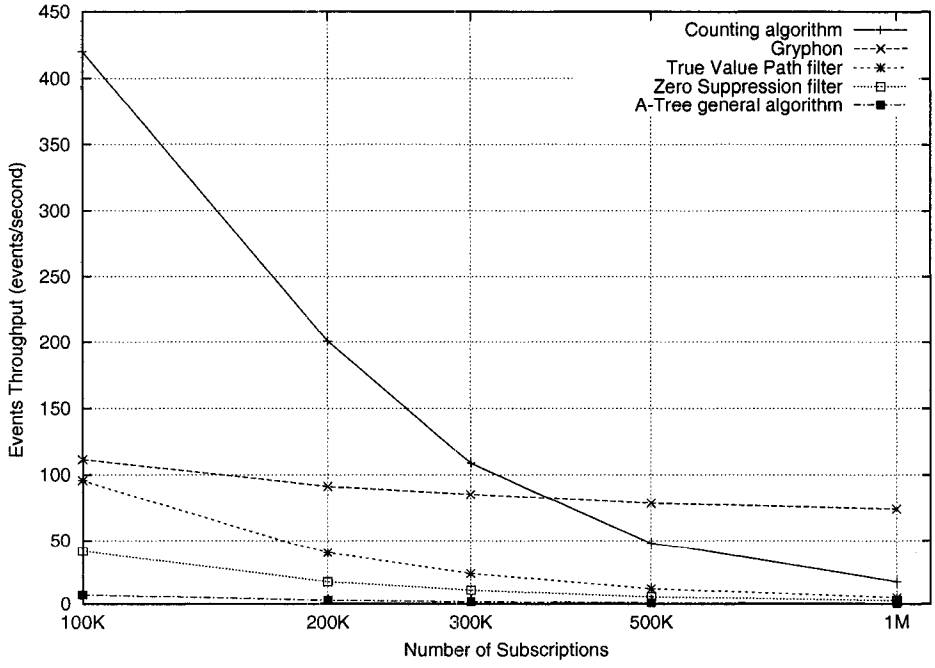


Figure 5.5: Event matching throughput with 10 predicates

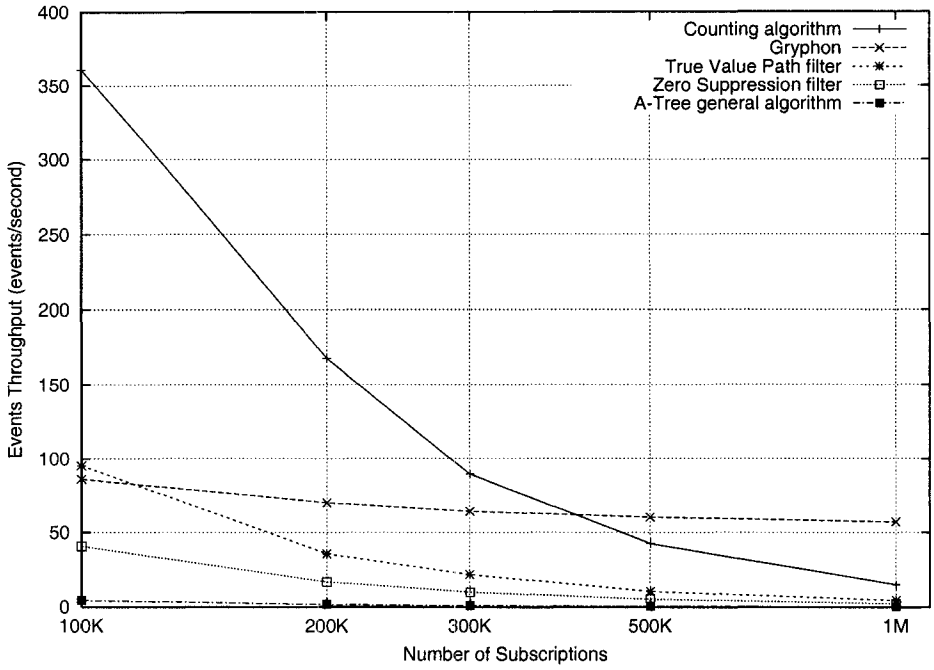


Figure 5.6: Event matching throughput with 14 predicates

Number of subscriptions	**Events per second for $L = 6$**	**Events per second for $L = 10$**	**Events per second for $L = 14$**
100,000	2.5615	1.0886	0.5838
200,000	1.2715	0.5440	0.2922
300,000	0.8475	0.3621	0.1957
500,000	0.5078	0.2164	0.1170

Table 5.3: Event matching rate of brute force algorithm

memory for the same number of subscriptions, but the memory requirement fo Gryphon goes up rapidly. The moderate memory requirement of A-Tree makes it suitable for implementing a very large number of subscriptions using a complex and powerful subscription language. Our tree structure can afford 4 millions of subscriptions with 14 predicates keeping the memory requirements within 1.6 gigabytes. To the best of our knowledge, no other experiment was performed in the past using such a high number of subscriptions with such a high number of predicates in each subscription. If we try to execute Gryphon without predefined set of attribute variables for each subscription, Gryphon takes over 1.8 gigabytes of memory for 300,000 subscriptions with 6 predicates. So we use a predefined set of attribute variables for this experiment to keep the memory usage low for Gryphon. The counting algorithm, on the other hand, takes exponential amount of memory if we use its bit-sliced-static-array version, which is known to be faster than the other versions [15]. However, the list-based version of the counting algorithm requires significantly less memory. We execute this version to compare it with our algorithms. Figure 5.7, 5.8 and 5.9 show the memory requirements of all the algorithms in three graphs for subscriptions with 6, 10 and 14 predicates. The counting algorithm shows the best results in terms of memory requirement. It uses exceptionally low amount of memory for a very high number of subscriptions. The TVP filter saves a substantial amount of memory since it does not use queues (Section 3.5.2). The general A-Tree algorithm, and the zero suppression filter take the same amount of memory. Compared to the counting algorithm, our algorithms take more memory due to the overhead of large static queues, two external tables and the complex node structure. The flexibility of our subscription language comes with this additional cost of memory requirement.

In Table 5.4 we show the memory requirement for the brute force algorithm.

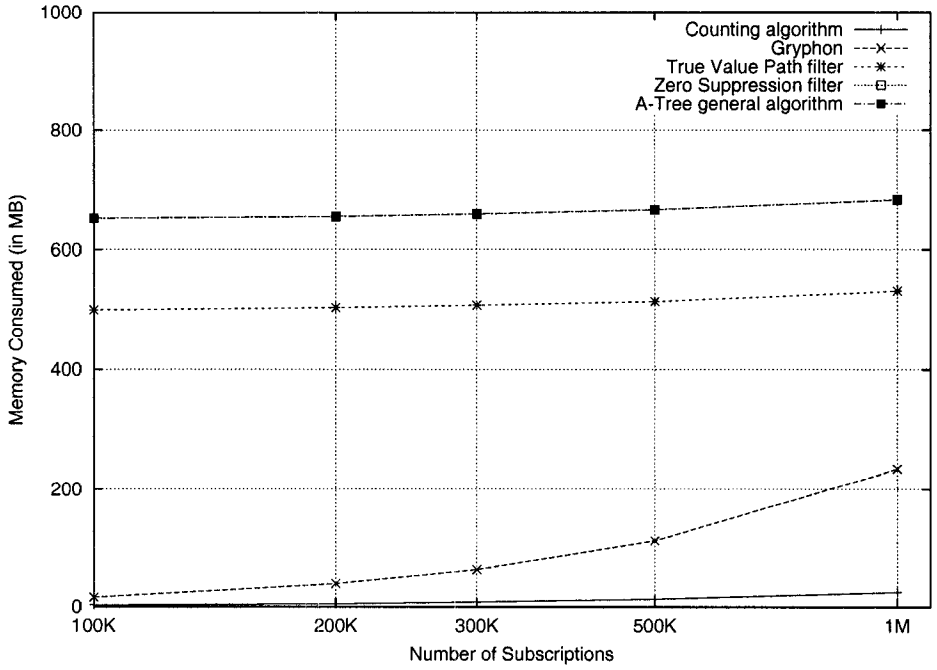


Figure 5.7: Memory consumption throughput with 6 predicates

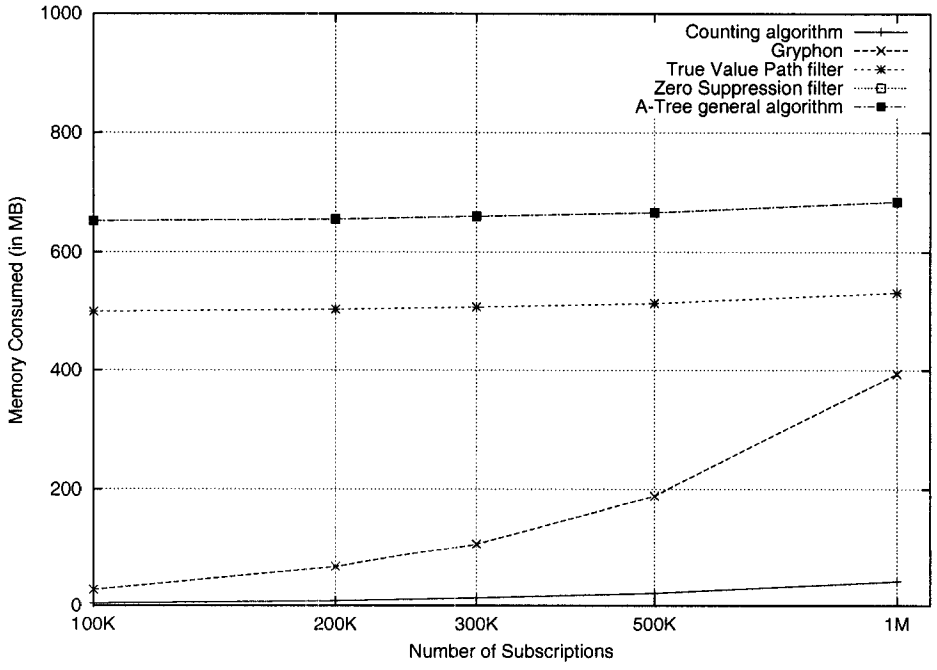


Figure 5.8: Memory consumption throughput with 10 predicates

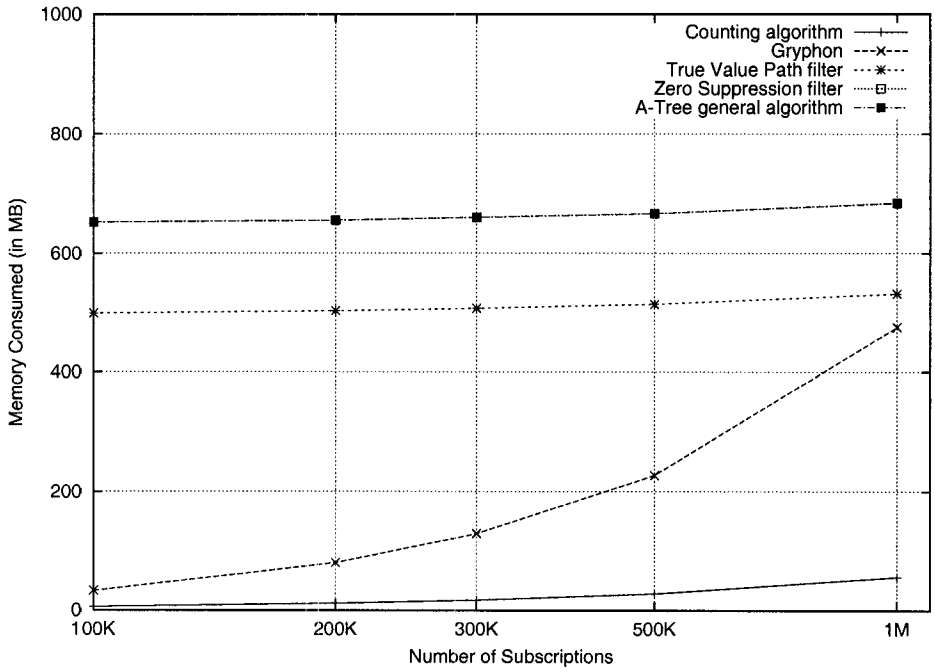


Figure 5.9: Memory consumption throughput with 14 predicates

Number of subscriptions	**Memory consumed (in MB) for $L = 14$**
100,000	153
200,000	307
300,000	461
500,000	768

Table 5.4: Memory requirement of brute force algorithm

Deletion Throughput

We show the deletion throughput of our three algorithms in Figure 5.10, 5.11 and 5.12. Deletion of a subscription is done in exactly the same way for any of the three A-Tree based algorithms. The results reflect this fact. We see that our A-Tree is maintainable in high loads of input. For example, with 1 million subscriptions, we get 20,000 deletions per second. Actually, the deletion of a subscription in all three of our algorithms is a much simpler task compared to insertion. The experimental results also prove this point. As the deletion algorithm is same for the A-Tree algorithm and its two variations, we see same deletion throughput in the figures. We currently do not have the deletion of subscription in our implementation of counting algorithm and Gryphon. Hence we are unable to show the deletion throughput for these two algorithms.

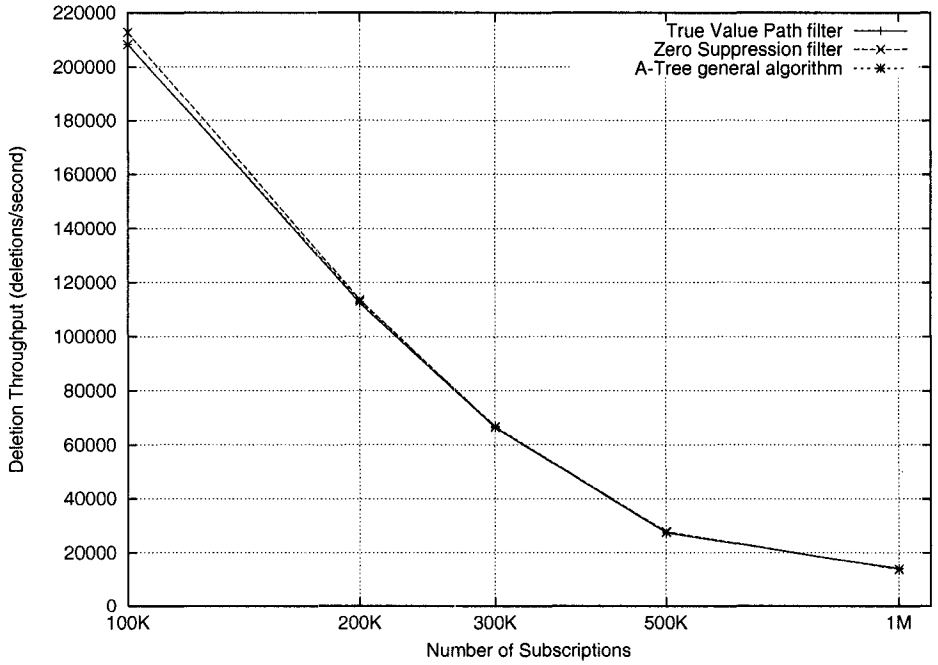


Figure 5.10: Deletion throughput with 6 predicates

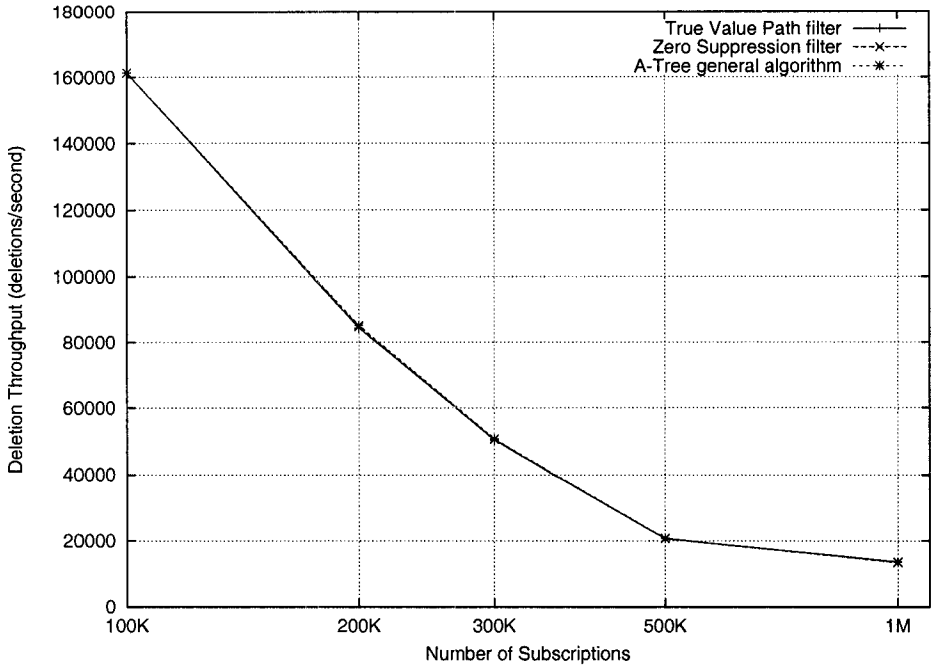


Figure 5.11: Deletion throughput with 10 predicates

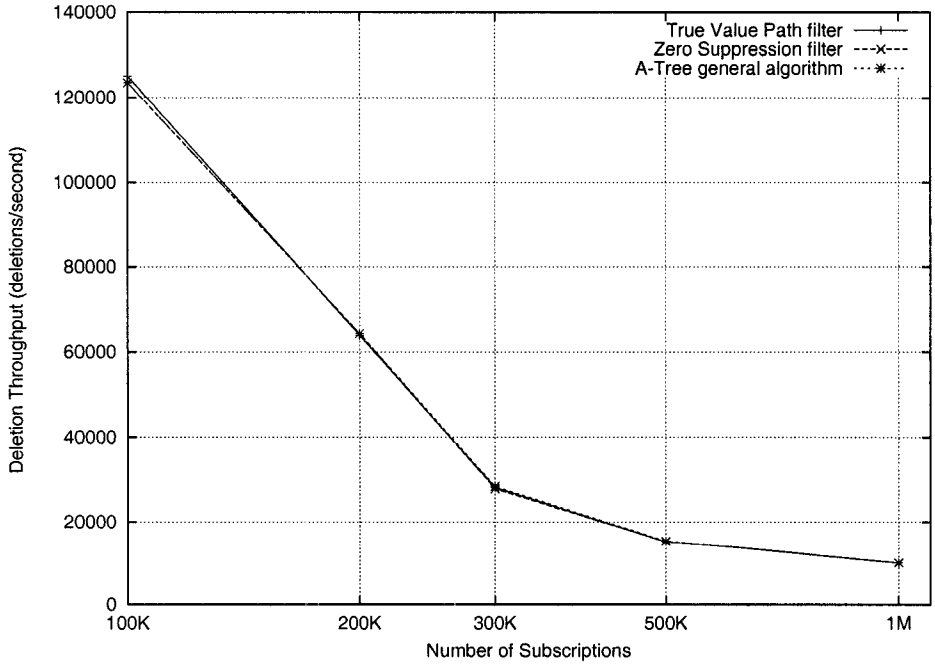


Figure 5.12: Deletion throughput with 14 predicates

5.5 Conjunctive and Disjunctive Workload

This workload consists only Boolean `AND` and `OR` operators with probabilities of occurrence of 0.8 and 0.2, respectively. We have executed the general A-Tree algorithm, zero suppression filter and the true value path filter to measure their performances. As mentioned before, Gryphon and the counting algorithms are not capable of handling such expressive languages.

Figure 5.13, 5.14 and 5.15 show the event matching throughput achieved by our three algorithms, with 6 and 10 predicates. For this workload, the *SNPS* was kept within the range 40% to 60%, and *SMPE* within the range 12% to 15%. We use the $>$ operator in each predicate. We get moderate rate of event throughput from all of the algorithms. The TVP filter outperforms the other two by a great margin. This again proves that the TVP filter is a promising choice as a matching algorithm to handle a large load of more expressive subscriptions.

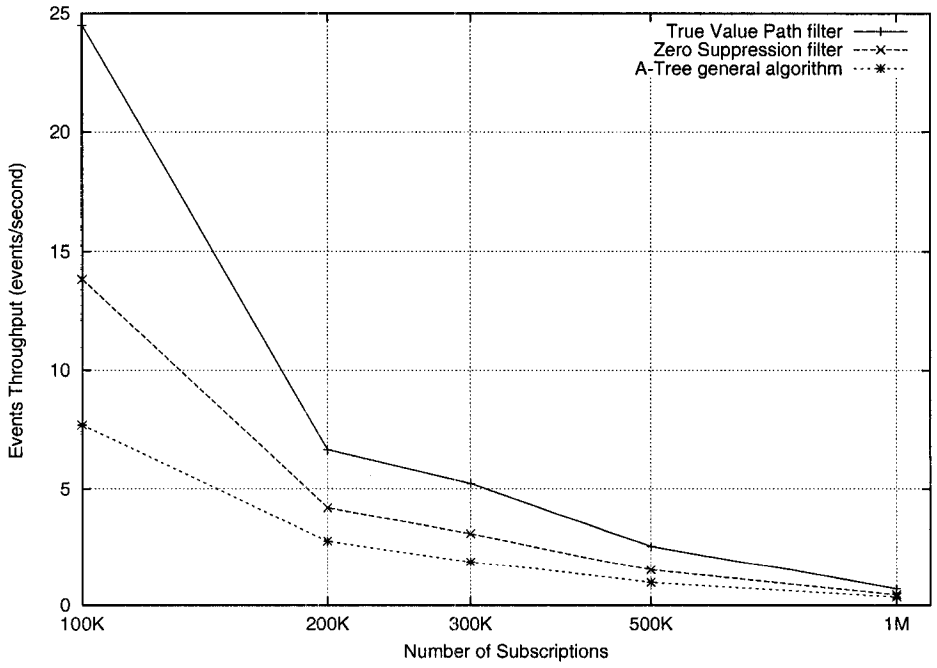


Figure 5.13: Event throughput with 6 predicates

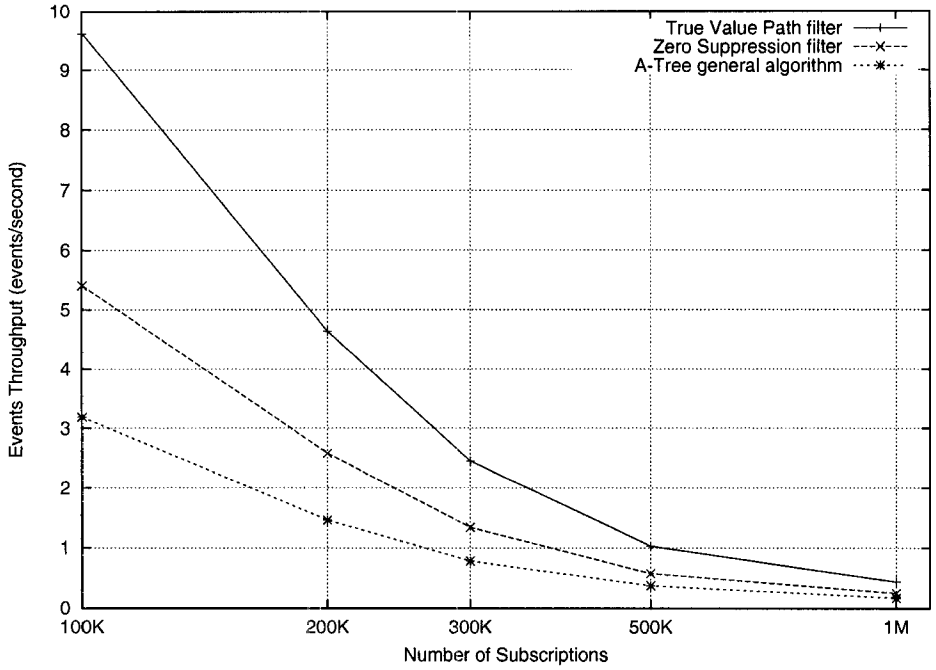


Figure 5.14: Event throughput with 10 predicates

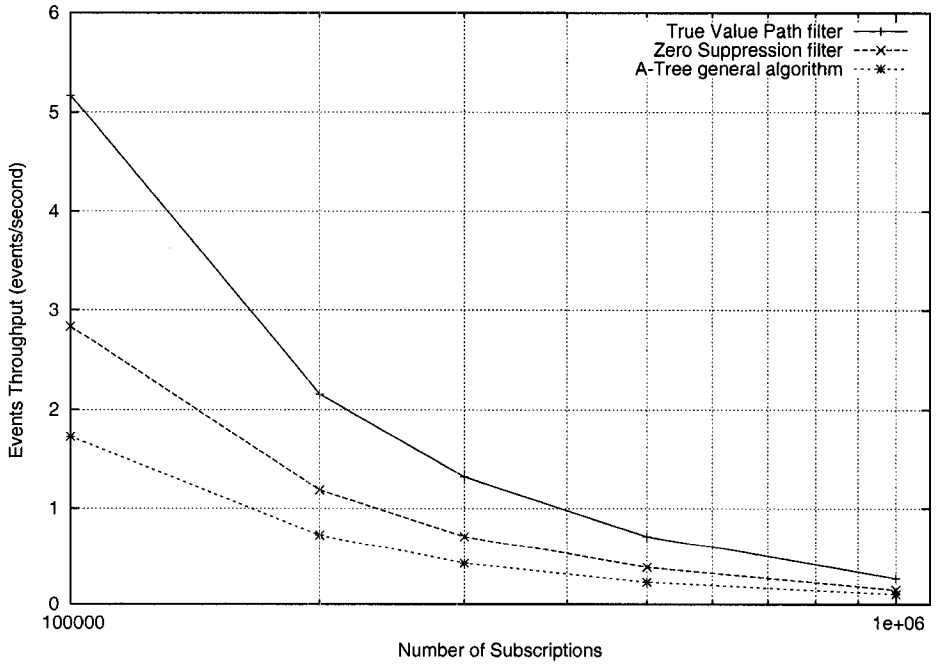


Figure 5.15: Event throughput with 14 predicates

Figure 5.16, 5.17 and 5.18 show the memory requirement for all the three algorithms. General A-Tree and zero suppression require the same amount of memory, whereas we find that TVP saves a substantial amount of memory due to the same fact we discussed in Section 5.4.1.

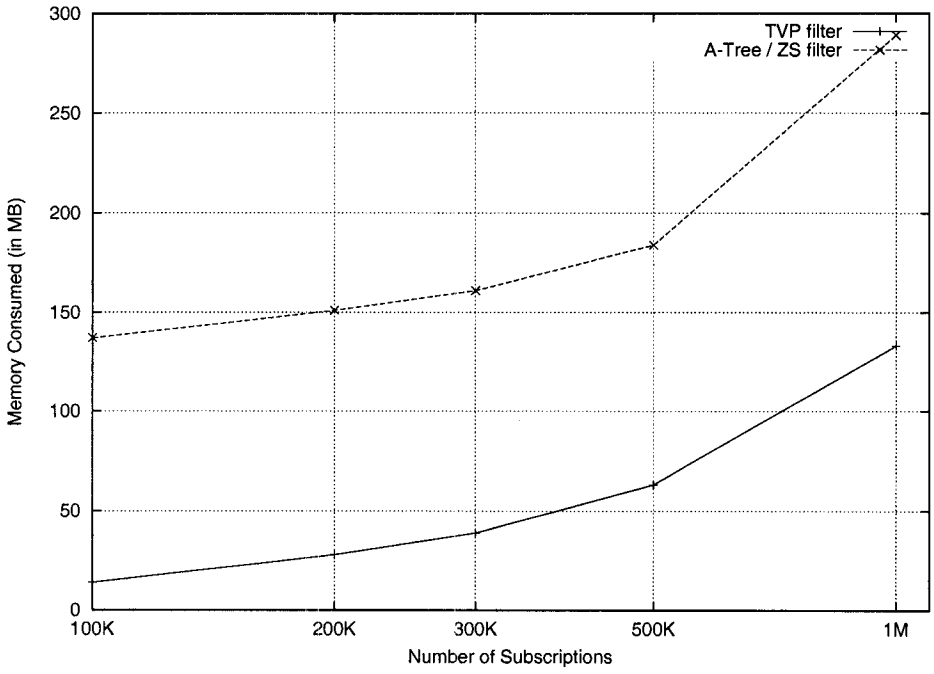


Figure 5.16: Memory consumption with 6 predicates

5.6 Mixed Workload

The third set of workload is only filtered with the general A-Tree algorithm and zero suppression filter, as other algorithms are not capable of handling this type of workload. In this type, we use the Boolean `AND`, `OR` and `XOR` operators with respective probabilities of occurrence of 0.6, 0.3, and 0.1 (Table 5.2). Our true value path filter also does not work in this type of workload due to the existence of the `XOR` operator in the subscriptions. Figure 5.19 and 5.20 show the event matching throughput achieved by our general A-Tree algorithm and zero suppression filter respectively, with 6 and 10 predicates. We use the >

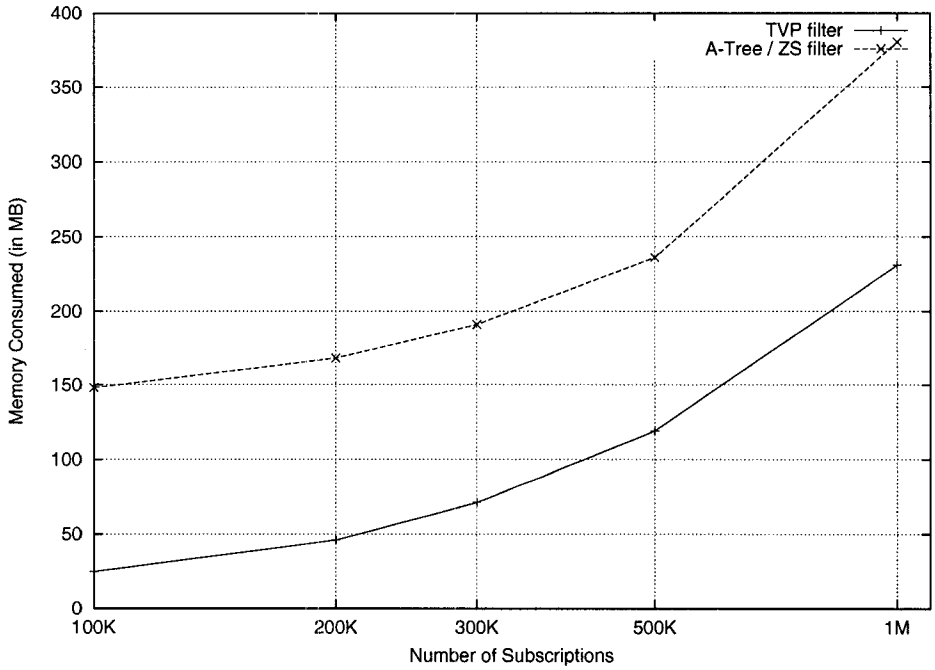


Figure 5.17: Memory consumption with 10 predicates

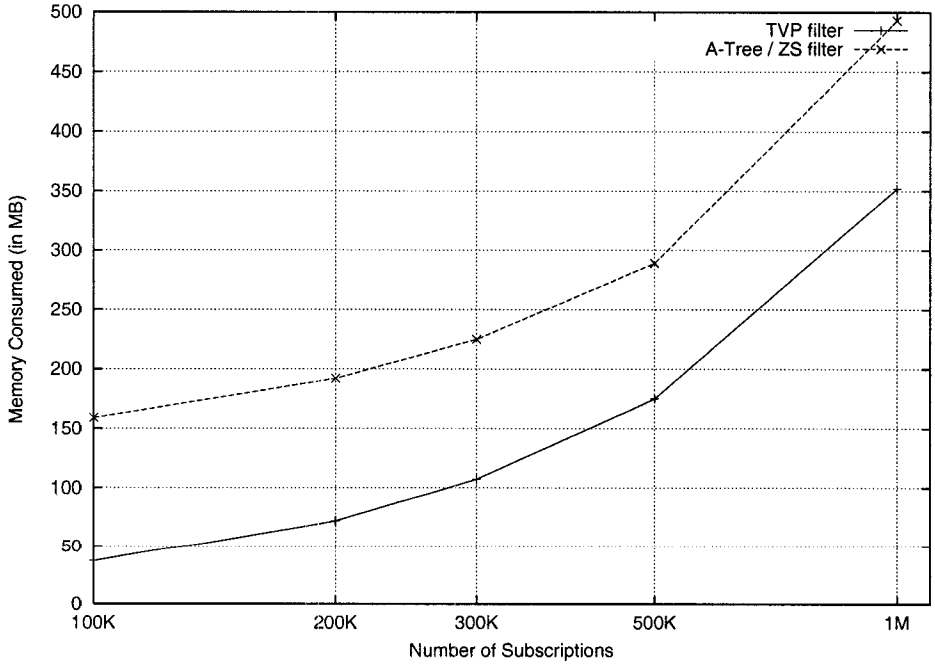


Figure 5.18: Memory consumption with 14 predicates

operator in each predicate. For this workload the *SNPS* was kept within the range 15% to 20%, and *SMPE* within the range 15% to 25%. We see that the event matching rate decreases much due to the use of other Boolean operators, compared to the conjunctive workload. The low value of *SNPS* is also another prime reason of low event throughput for this workload instance.

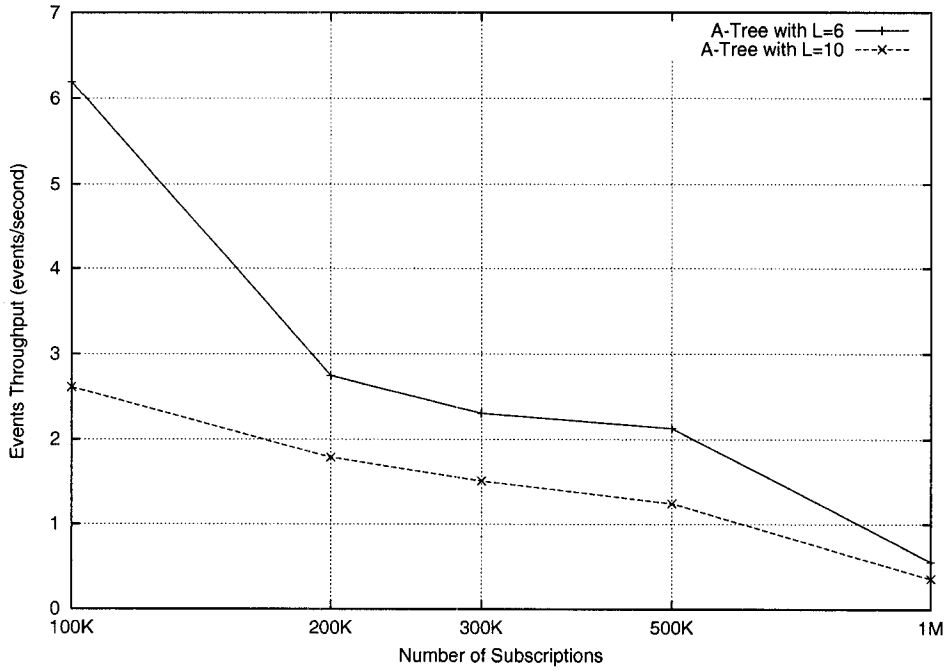


Figure 5.19: Event throughput of general A-Tree algorithm in a mixed workload

Figure 5.21 and 5.22 show the memory requirements of the vanilla A-Tree algorithm and zero suppression filter. We see that the A-Tree structure does not require excessive amount of memory due to the mix of operators we use in the workload.

5.7 Effect of Varying Predicate Size

In the past, researchers working on subscription matching algorithms did not address the issue of change in performance with the change of predicate size in each subscription. We consider this to be an important factor affecting the performance of any matching

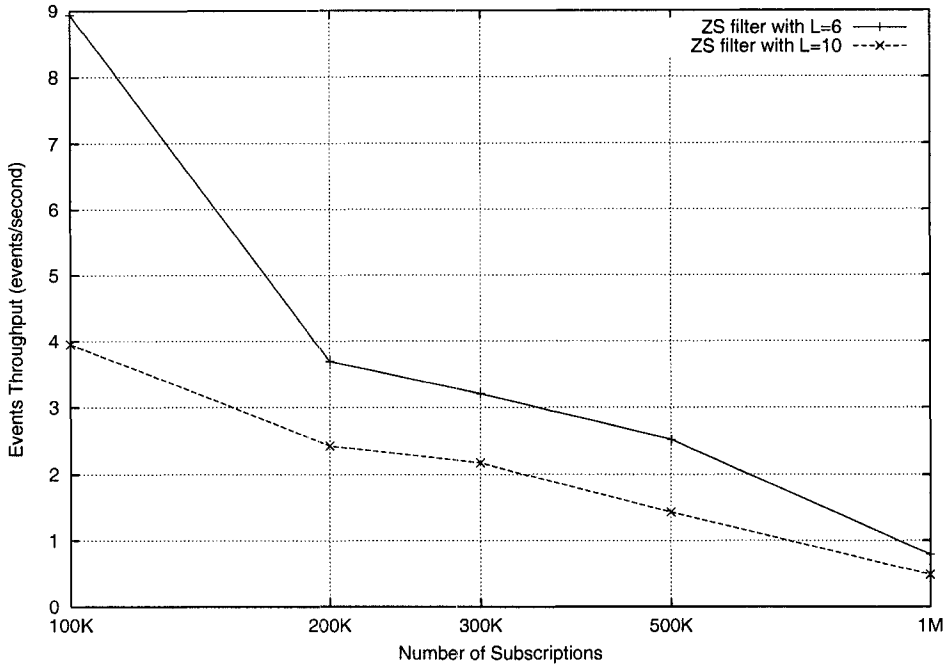


Figure 5.20: Event throughput of zero suppression filter in a mixed workload

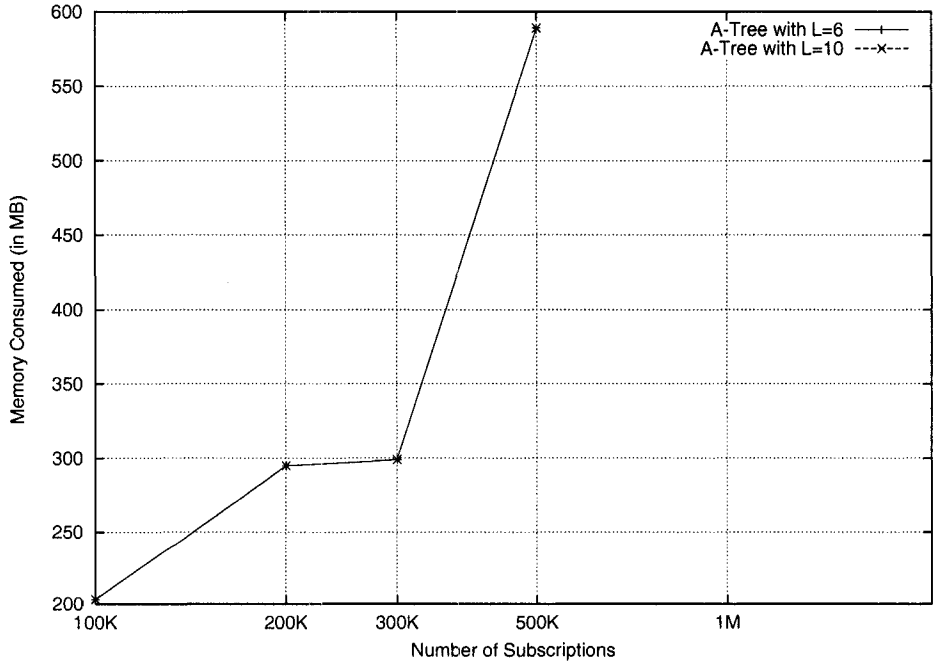


Figure 5.21: Memory consumption of general A-Tree algorithm for mixed workload

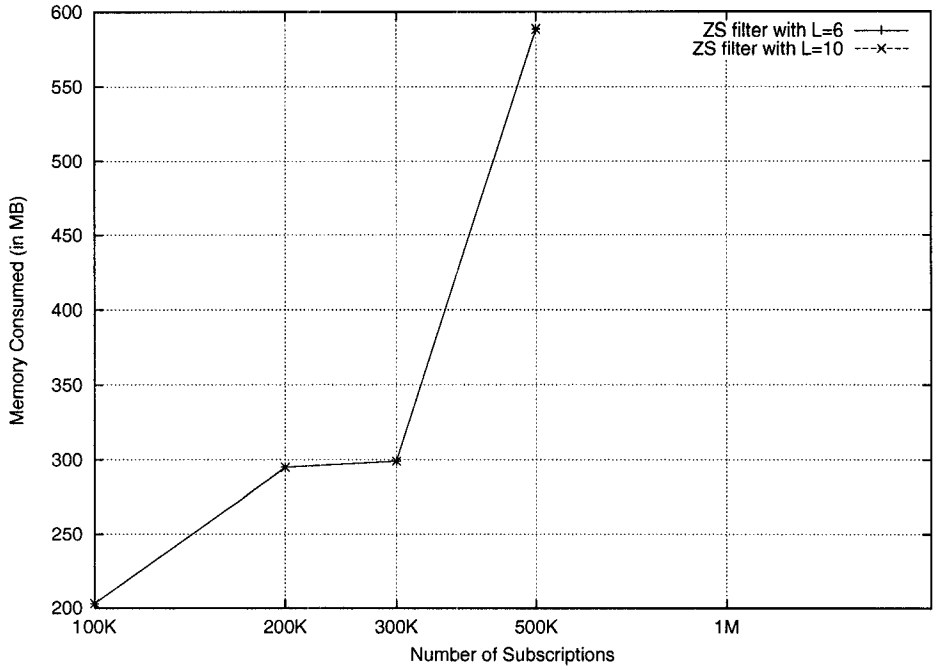


Figure 5.22: Memory consumption of zero suppression filter for mixed workload

algorithm. To observe the effect of different number of predicates in each subscription in the total subscription population, we set the workload parameters as shown in Table 5.5.

N_{subs}	100000, 200000, 300000, 500000, 1000000, 2000000, 3000000, 4000000
L	6, 8, 10, 12, 14
N_E	2000
A_E	within the range L_{min} to 18
operators	Boolean `AND`

Table 5.5: Workload parameters for varying predicate sizes

We vary L and N_{subs} and keep other parameters fixed. For each L, we run the algorithms for all different subscription populations where each subscription has exactly

L predicates. For simplicity, we use conjunctive workload here and we run our generalized and TVP filter algorithm on this type of workload. We use the $>$ operator in each predicate and keep the *SMPE* parameters for all of these experiments within the range of 40%–50%. Figure 5.23 and 5.24 show the results obtained from this experiment. Our experiments reveal that the event throughput is very sensitive to the number of predicates present in each subscription. The higher the number of predicates we use, the slower event matching rate we get from the matching algorithm. So number of predicates used in each subscription is a prime concern for performance.

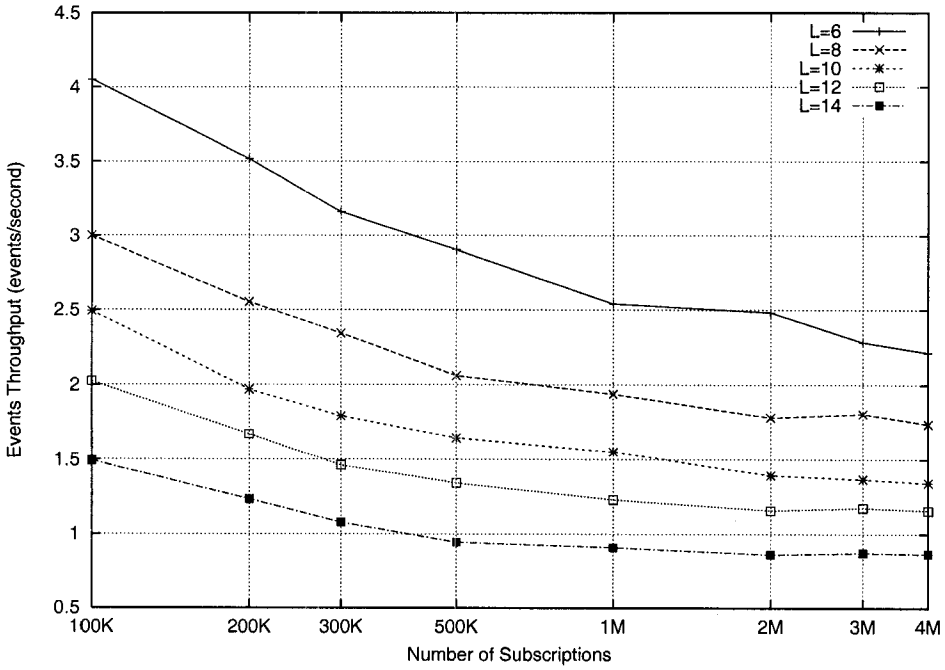


Figure 5.23: Event throughput for A-Tree algorithm with varying L

Using a different type of workload, we compare four algorithms to observe the effect of varying predicate length. Here, we use conjunctive workload with equality predicates. We kept 1 million subscriptions in the system for each algorithm. We vary L and observe the change in event throughput. Figure 5.25 shows the results of this experiment. We see that only Gryphon is largely effected by the increase in L.

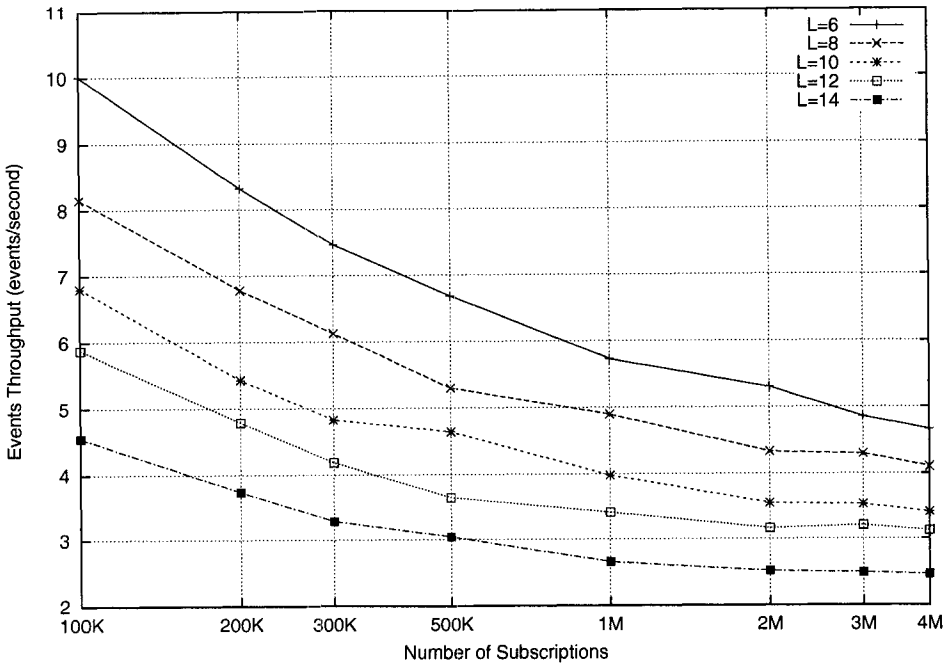


Figure 5.24: Event throughput for TVP filter with varying L

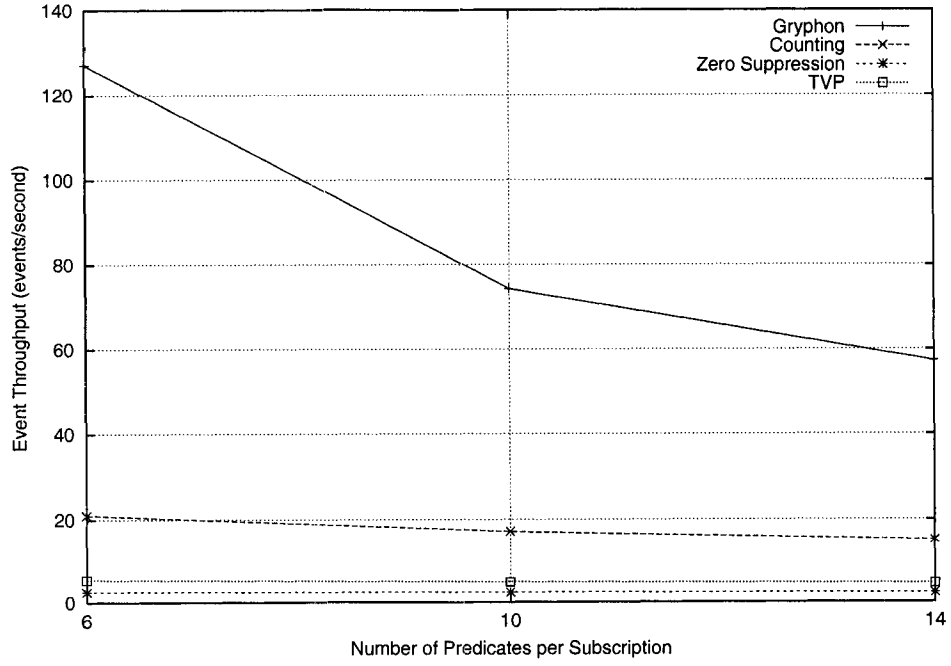


Figure 5.25: Event throughput for four algorithms with varying L

5.8 Effect of Expression Overlap

In this experiment, we want to explore the effect of varying the amount of shared expressions in the subscription population, which is a very important factor affecting the event match and insertion throughput.

We observe the event throughput performance for different *SNPS* using our TVP filter algorithm in Figure 5.26. However, it is not a deterministic process for our current workload to produce an input set with a precise *SNPS*. So the value of SNPS, shown in percentage in the figure, varied in the range of 1%–2%, which we did not consider as a large discrepancy. The workload setup for this experiment is almost the same as it is for the experiment shown in Section 5.7. For simplicity, we use a conjunctive workload for this experiment, and use 14 predicates in each subscription. We use the > operator in each predicate and keep *SMPE* within the range of 30% to 45%. In Figure 5.26, we

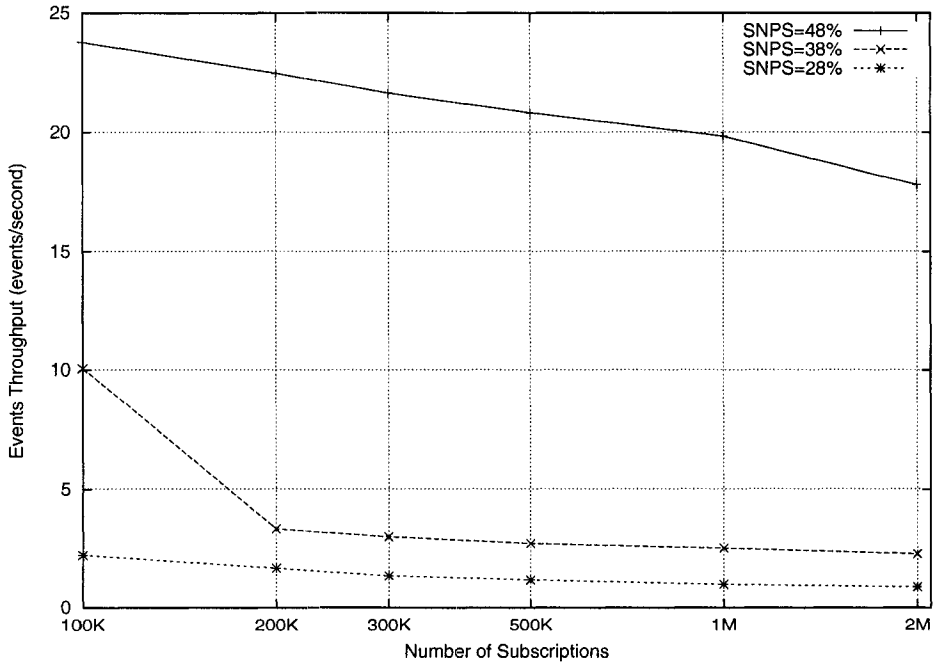


Figure 5.26: Event throughput for TVP filter algorithm with different SNPS

see that overlap plays a vital role in the event throughput. A higher number of reused

subexpressions in the A-Tree always provides us with better event matching rate. We need to traverse less number of nodes to match each event, hence a substantial amount of computation is saved.

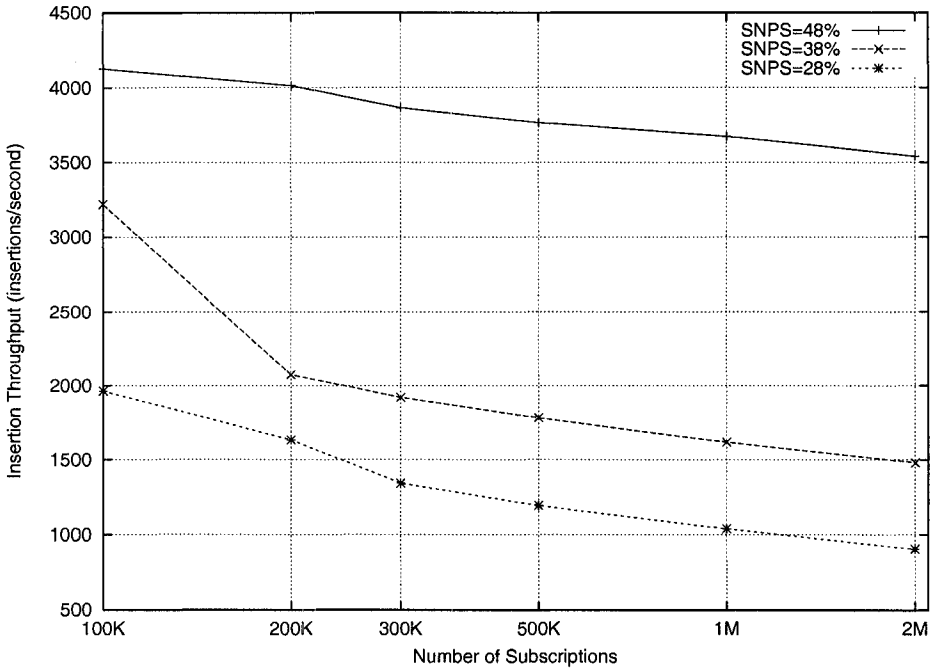


Figure 5.27: Insertion throughput for TVP filter algorithm with different SNPS

In Figure 5.27, we see the effect of different amounts of overlap in case of insertion throughput. We observe that the number of insertions per second is higher when we have higher SNPS. The reason is that if we have less overlap within the subscription population, insertion of a new subscription leads us to a number of failed searches in the edge list of some nodes in order to find existing subexpressions. Those failed searches incur extra overheads due to additional computations in case of insertions and hence the throughput decreases.

In Figure 5.28, we show the amount of memory consumed by the TVP filter for different amounts of overlap. The results are as we expected. The increase in overlap in subscription expressions gives us a more concise A-Tree structure, and it takes less memory. In Figure 5.29 we also show the number of nodes created and used by the

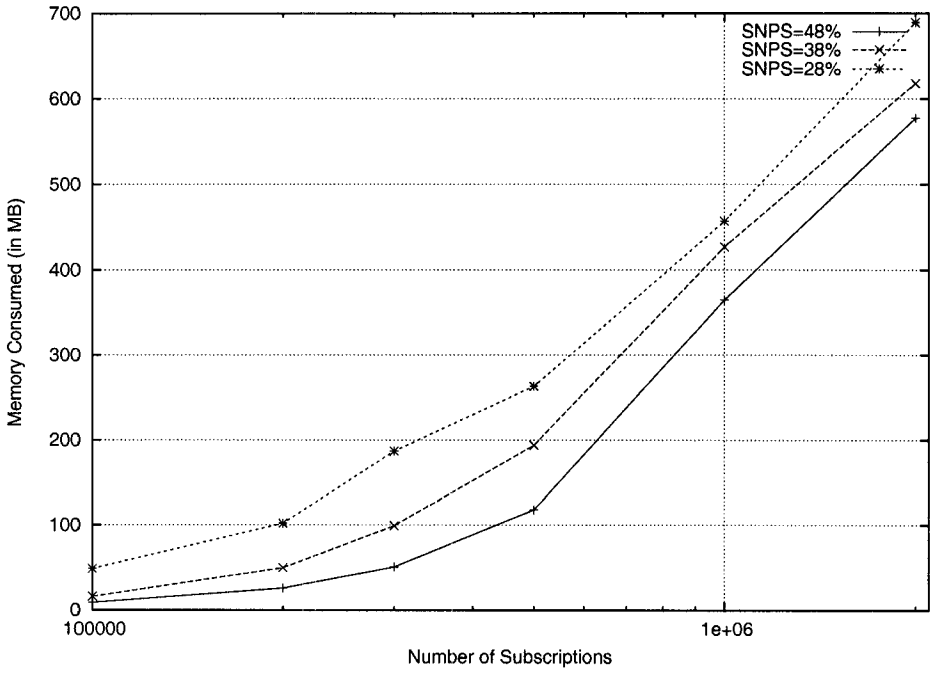


Figure 5.28: Memory consumed by TVP filter algorithm with different SNPS

algorithm in different situations. We see that the A-Tree structure comparatively uses less number of nodes for higher SNPS. This confirms the fact that our insertion algorithm is successfully taking advantage of overlapping expressions existing in the subscription population.

5.9 Effect of the Number of Matches per Event

We observe another important parameter of a subscription matching algorithm, *SMPE*. We observe the event throughput for three different values of *SMPE*. To control the *SMPE* in the workload, we use the $>$ operator in each predicate. We control the values of all the attribute variables of an event so that we have the expected number of matches for that event. We use only the `AND` operator and ensure that the number of attributes of each event (A_E) is equal to or greater than L. However, due to the random generation of predicates and events, this parameter is not precisely controllable in our workload

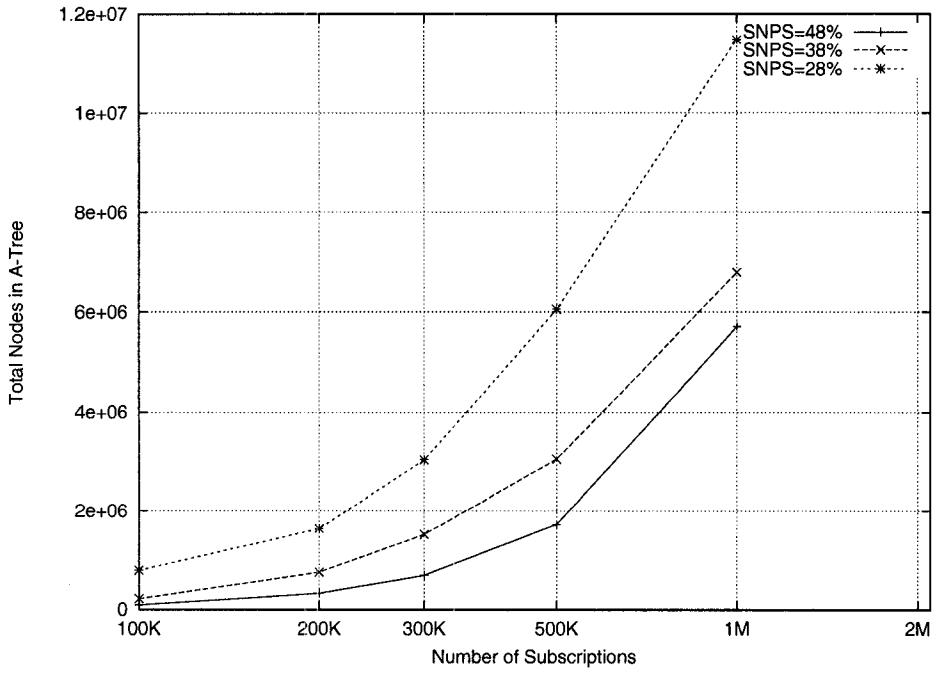


Figure 5.29: Number of nodes created by TVP filter algorithm with different SNPS

generator. So we generate each set of inputs with *SMPE* defined in a range. We run our TVP filter algorithm and counting algorithm with three different ranges of *SMPE*. In the workload, we keep 14 predicates in each subscription, and the *SNPS* is around 35%.

In Figure 5.30 and 5.31, we show the event matching throughput results for three different *SMPE* values. It is clear that if an event matches too many subscriptions in the system, it consumes more time because the algorithm has to traverse most of the nodes in the tree. In our algorithms, A lower *SMPE* value leads to a lower number of node traversals in the A-Tree. Hence we get better throughput. In reality, it is unusual to see an event satisfying 70% or 80% of the subscription population. Still in this experiment, we can deduce the important fact that *SMPE* plays a vital role in the event throughput for a subscription matching algorithm.

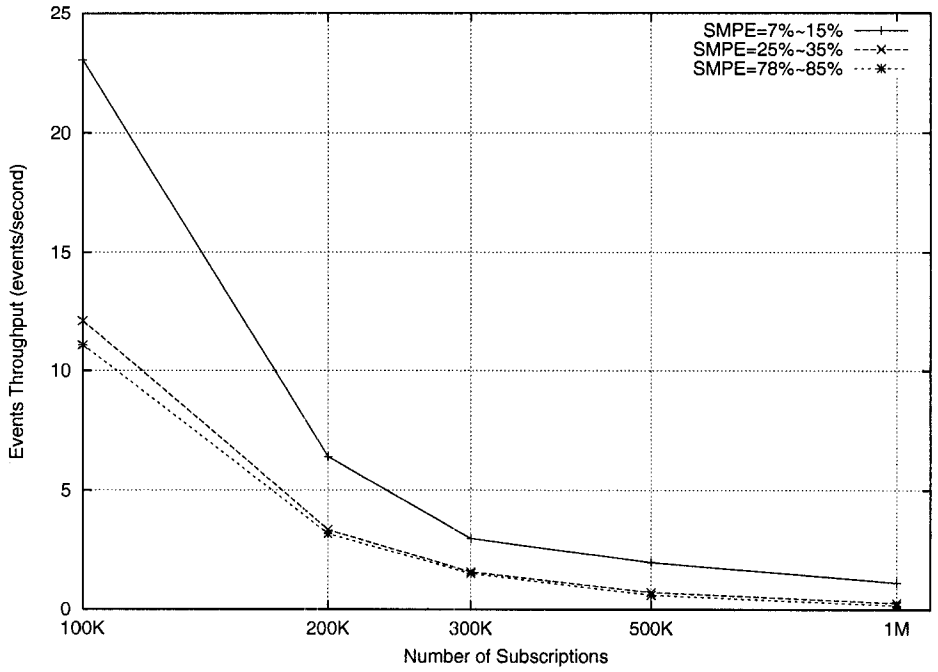


Figure 5.30: Event throughput of TVP filter algorithm with different SMPE

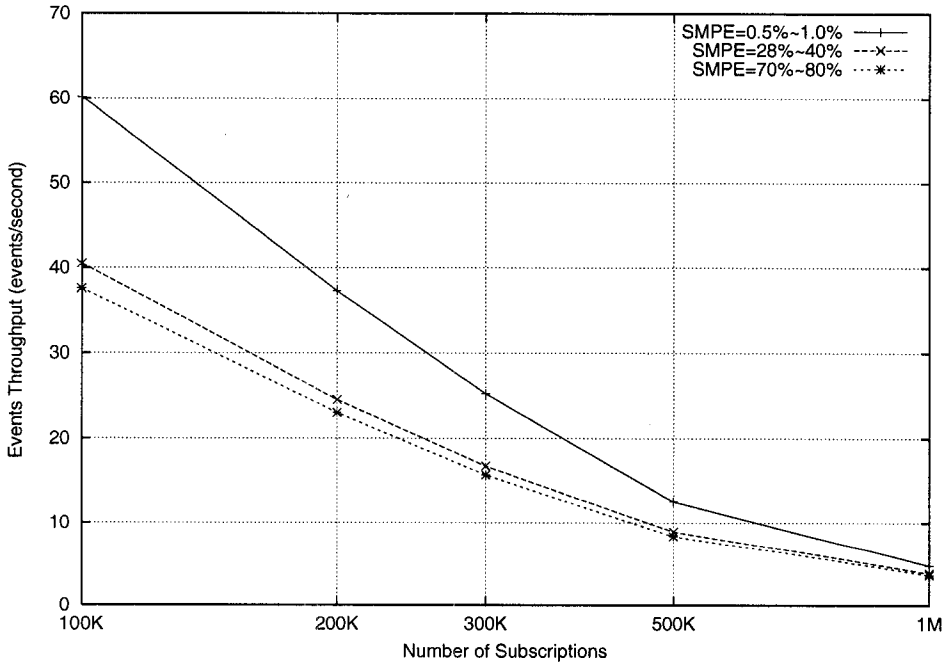


Figure 5.31: Event throughput of counting algorithm with different SMPE

Chapter 6

Conclusions and Future Work

In this thesis, we have developed the A-Tree matching algorithm, with two other optimizations on it. The A-Tree is a powerful data structure that is able to process any subscription expression over relational predicates. The bulk of current research in publish/subscribe systems focuses exclusively on processing conjunctive subscription languages. Therefore, the A-Tree algorithm will contribute significantly to the study of publish/subscribe matching algorithms by allowing a more flexible subscription language. The benefit of a more general subscription expression is the increased filter expressiveness with greater filtering power. Furthermore, several novel subscription languages have been suggested and it has been shown how to implement them with the A-Tree structure thus proving its flexibility over existing matching algorithms. The existing algorithms are constrained to the processing of attribute-value-pair type models. The experimental results demonstrate that the A-Tree exhibits scalable performance for subscription populations in the order of millions. It performs almost as well as the popular counting algorithm for higher number of subscriptions, but not as good as Gryphon. However both counting and Gryphon are specialized for conjunctive subscription languages and cannot be extended for more general subscription expressions, such as the ones processed by A-Tree. This shows that the flexibility gained by A-Tree comes at a performance price, which, as we

showed, is used, especially, due to its superior performance than counting, a popular and widely acceptable matching scheme. With some restrictions in the language, the two optimized versions of our algorithms show an even higher rate of event matching, where both of them allow conjunctive and disjunctive operators in a subscription.

6.1 Future Work

One potential direction of future research is the implementation and evaluation of different powerful subscription models as discussed in Section 4.2. We plan to implement the priority-based subscription (Section 4.1) and evaluate its performance.

Pipelining of the two stages is another interesting extension that is worth experimenting. We need to restructure the TVP filter algorithm so that it can be executed on a multi-processor machine. We are very much interested in observing the throughput we can achieve by this optimization.

We can optimize our algorithm with our own cache management so that we can reduce the number of cache misses. This can potentially give us substantial performance benefits. Having a set of nodes currently in the process of computation, we can actually make some rough estimate on the next block of nodes that will be accessed. If a node is currently being processed, it is pretty obvious that we will need to process some of its parents very soon. So with some appropriate cache prefetching mechanism, we can try to make a node available in the cache before we start processing it. This will inhibit many of the cache misses and result in an even faster algorithm.

We have measured the performance of our algorithms using our own synthetic workload generator. But we actually want to use data sets collected from real-life applications, e.g., airline ticket reservation or hotel reservation web portals, etc, and execute the same set of experiments. Unfortunately, we do not have access to any such dataset. We also plan to observe the pattern of predicate usage in the subscription population of a real

application.

Appendix A

Related Theorems and their Proofs

Here we present the proofs of the various theorems presented in the thesis. It is meant to serve as a source of supplementary information and a thorough understanding of the topics discussed here is not essential to understand the main ideas expressed in the thesis.

Definitions

In this section, we introduce a few terms used in our different proofs. We also define some new concepts related to our tree and its property.

- We first define the term *population*(L), which is the number of nodes in the tree such that each of those nodes has the level field set to L. In other words, the term refers to the number of nodes residing at level L.

- We define the level L to be an empty level if population(L) is 0, i.e. there are no nodes at level L.

- We define *level*(α) as the level number of a node α in the A-tree. When we use this term, we assume that α already exists in the tree.

- We define M as the maximum level number currently existing in the A-tree. So for

any node α, $level(\alpha)$ can be within the range starting from 1 to M.

- We define T_S as the binary tree representation of a subscription S.
- A node α is black or white when the computed result of α is `true` or `false` respectively.
- A black path exists from the node α to the node β if
  1. there is a path of some nodes in a tree T_S starting from α and ending at β,
  2. the nodes α and β are black,
  3. all the intermediate nodes in the path are black.

It is worth noting here that for the following proofs, it is always assumed that the tree is in a *stable/consistent* state. All of the following theorems indicate different properties of the A-tree structure and these properties will hold only if we are not in the middle of processing a subscription insertion or deletion.

Theorem 1. *If there exists at least one node at level L ($1 < L \leq M$) in the A-tree, there must be at least one node at level $L-1$.*

Proof. For $L = 1$ this proposition does not mean anything as we do not have any levels lower than 1. As we will prove it by induction, we will start with $L = 2$ as our basis. To prove the theorem for $L = 2$, we first assume that there exists a node α such that $level(\alpha) = 2$. It means that α was created and inserted in the tree structure to represent a function between 2 nodes of level 1. We recall the property of the level field from Equation 3.1 in Section 3.3.1 below:

$$level(R) = 1 + max\{level(P_i), \forall P_i | P_i \text{ is a direct child of } R\} \tag{A.1}$$

This indicates that the 2 nodes that point to α must reside in level 1. That is why the value of $level(\alpha)$ was set to 2 according to the property mentioned. Therefore our proposition holds for $L = 2$.

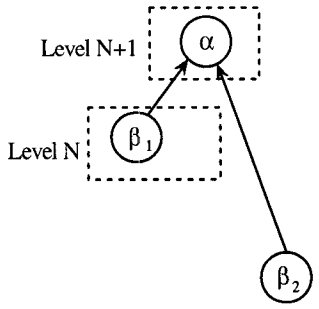


Figure A.1: Dependency of level numbering scheme for a node

Now, let us assume that α is a node residing at level $N+1$ ($M > N > 0$) and α has two children namely β_1 and β_2. We assume $level(\beta_1) \geq level(\beta_2)$. If it does not hold we can simply swap their names. Now, according to Equation A.1, $level(\alpha) = level(\beta_1) + 1$, since $max\{level(\beta_1), level(\beta_2)\} = level(\beta_1)$. But we assumed $level(\alpha) = N+1$. Hence, $level(\beta_1) = N$. So it is clear that if there is a node at level $N+1$, there must be at least one node at level N. □

Corollary 1. *No level L in the A-tree can be empty, given $1 \leq L \leq M$.*

Proof. We will prove this theorem by contradiction. We initially assume L to be the minimum level number such that level L is empty, i.e. $population(L) = 0$, and $1 \leq L \leq M$. We will try to prove that the above assumption can never be true.

If $L = M$, M should have been set to some lower level which is not empty. The topmost level M can become an empty level in some cases when a subscription is deleted. This might happen in a case where all the nodes at level M get deleted. But the deletion operation sets the new value of M to the highest non-empty level value again. So if we assume a correct implementation of the deletion algorithm, $population(M)$ can never be 0. Hence we will try to disprove our initial assumption for $1 \leq L < M$.

Now, we again assume L' ($L' \leq M$) to be the lowest level number such that $L' > L$ and $population(L') \neq 0$. So L' is non-empty and there is at least a single node existing at level L'. The assumption also implies that all the levels greater than L and less than

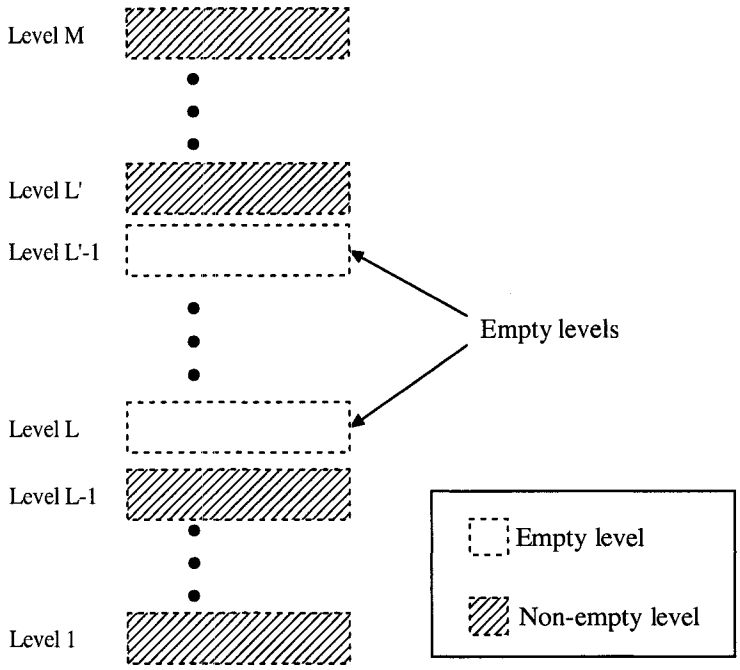


Figure A.2: Figure explaining the assumptions for theorem 2

L' are empty. As there exists at least a single node at level L', according to Theorem 1 $L' - 1$ cannot be empty. By considering the same property recurrently for $L' - 1$, $L' - 2$ etc., we get the fact that $L + 1$ is non-empty and so is L. Hence, there does not exist any L $(1 \leq L \leq M)$ such that $population(L) = 0$.

So in an A-tree, any level starting from 1 to M cannot be empty. □

From Theorem 1 and Corollary 1 we can deduce some important conclusions regarding the subscription insertion and deletion operations. We formalize the conclusions in the form of Corollaries, and prove them below.

Corollary 2. *Deletion of a subscription does not affect the level numbering of the remaining nodes in the tree.*

Proof. Deletion of a subscription can lead to a situation where only the topmost level M becomes empty. This might happen if all the nodes at level M get removed due to the deletion of the subscription. But if the topmost level becomes empty, the level labeling of

the remaining nodes in the tree does not get affected. Recall from our definition of level that for any node α, $level(\alpha)$ does not depend on $level(\beta)$ where $level(\alpha) \leq level(\beta)$. In other words, the level numbering of a node at a lower level can never be affected by the level numbering of a node at a higher level. The deletion algorithm sets the new value for M if the level M becomes empty. So it is clear that if only the topmost level becomes empty due to deletion of a subscription, the level numbering of the remaining nodes does not get changed. If any level other than M becomes empty, level numbering of some nodes may get affected.

According to theorem 2, no level in the tree can be empty. So even after deletion of a subscription, it is never possible to have an intermediate level to be empty. As we do not have any empty level, not a single node needs to have its level numbering changed.

Hence, we conclude that deletion of a subscription does not affect the level numbering of the remaining nodes in the tree. □

Corollary 3. *Insertion of a subscription does not affect the level numbering of the remaining nodes in the tree.*

Proof. The level numbering of the nodes in A-tree can be affected in two situations as described below,

- if we have to insert a new level
- if we have an empty level

By adding a new subscription, we can never have the second situation as insertion of a subscription does not involve deletion of any nodes from the tree.

The first situation can occur in a particular case when we need to add a new level $M + 1$. The new subscription is defined in such a way that a node α is required to be created having one of its children residing at level M. This implies that $level(\alpha) = M+1$. Thus we create a new level $M + 1$ and the value of M is set to the new increased value.

But this whole process does not affect the level numbering of any other node, as α resides above all other existing nodes.

Due to the insertion of a subscription, it is never possible to add a new level L where $L \leq M$. Because if we assume that we have to add a new level L, this implies that before adding the subscription, the level L was empty. But it violates the property of the tree described by Corollary 1. So insertion of a subscription does not add a new intermediate level. As we do not add any new intermediate level, the level numbering of preexisting nodes do not get affected.

Hence the insertion of a subscription never affects the level numbering of the existing nodes in the tree. □

Theorem 2. *If α and β are two nodes in the tree such that β is a child of α, use_count(α) $\leq$ use_count(β).*

Proof. The field *use_count*(α) refers to the number of subscriptions that use the node α. We initially assume that the property holds at any time. We will show that after addition or deletion of a subscription, this relation still holds.

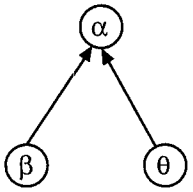


Figure A.3: Parent-child relationship

If due to insertion, α and β are created for the first time, then *use_count* of both the nodes are set to 1. The property holds for this case.

If α and β existed before adding a new subscription (let S), there can be three different cases. First, the new subscription S does not use either of α or β. This is a trivial case which does not violate the property.

Second, S uses β but not α. Due to insertion of S, β now points to a new parent other than α, or β is the topmost node used by S. For both of the cases according to our insertion algorithm, $use\_count(\beta)$ is increased by 1 while $use\_count(\alpha)$ remains unchanged. So the property holds.

One thing to note is that if a subscription uses α, it surely uses β as well. α represents a binary operation between two operands. Those operands are represented by the two children of α. If a subscription uses α, it obviously uses both children of α, one of which is β. Hence the third case is that S uses both α and β. In this case, both $use\_count(\alpha)$ and $use\_count(\beta)$ is increased by 1. So the property holds. Hence, adding a new subscription always maintains the property.

Now let us consider the case where we delete a subscription (let S). We will start with the assumption that before deletion our proposition was true. For deletion of S, we can again have three different situations. First, S does not use α or β, and so they do not get affected by the deletion. This is a trivial case which cannot violate the property.

Second, S uses both α and β. According to the deletion algorithm, both $use\_count(\alpha)$ and $use\_count(\beta)$ are decreased by 1. As the property was true before the deletion, it will also hold afterwards. After the deletion $use\_count(\alpha)$ might turn out to be 0. α is deleted from the tree in that case. If both of them turn out to be 0 by this operation, both the nodes are deleted from the tree. After the deletion of S, it is not possible to have $use\_count(\beta) = 0$ but $use\_count(\alpha) > 0$, because before deletion we had $use\_count(\alpha) \leq use\_count(\beta)$.

The third case occurs when S uses β but not α. We can have two different situations here. First, β has a parent other than α. Second, β is the topmost node used for S. So β is directly mapped to S, and β does not have any parent that is used by S. We also know that all subscriptions that use α also use β. So before deleting S, for both of the situations we had $use\_count(\alpha) < use\_count(\beta)$. According to the deletion algorithm, we will decrease $use\_count(\beta)$ by 1. So after deletion of S, the property still holds.

Hence, the property always holds for our tree structure. □

Theorem 3. *Any non-leaf node in the tree will always have exactly two children.*

Proof. We initially assume that the property exists in the tree. We will show that insertion and deletion of a subscription maintain this property.

When a non-leaf node α is inserted for the first time, $use\_count(\alpha)$ is set to 1. According to the insertion algorithm, α will have two children. So the property holds for a newly inserted node. Insertion does not delete any node or any parent-child relationship. So after an insertion, the property holds.

Now we consider deletion of a subscription S. Let us assume three nodes α, β and θ where α is the parent of the two other nodes. If due to the deletion of S, we have to delete β but not α, we will get into the situation where a parent has one child. But we will show that it can never happen. β is deleted only if its *use_count* field has become 0. Before deletion of β due to the deletion of S, $use\_count(\beta)$ was 1. According to the property of theorem 3, $use\_count(\alpha)$ was also 1. It means that S uses α, as well as β. So the deletion algorithm will also decrease $use\_count(\alpha)$ while deleting S, and eventually α will be deleted from the tree. So the deletion of a subscription does not violate the property. Therefore any non-leaf node will always have exactly two children. □

Theorem 4. *The values of the two operands of a node α are always available at the moment of computing the result of α.*

Proof. Let us assume the level of α is L. Now, in our A-Tree algorithm, with or without the zero suppression filter, α is popped out of the queue Q_L only after finishing the processing on the queues $Q_1, Q_2, \ldots, Q_{L-1}$. The two operands of α are updated by its two children that reside at levels lower than L. As α was inserted in the queue Q_L, anyone or both of the two children of α had updated its operands. We definitely get one of the two possible situations here. First, if it is a general A-Tree algorithm, we have both of the operands ready. So we can compute the result of α right away. Second, in

the case of zero suppression filter, the value of one of the operands might be undefined but it is assumed to be `false` according to zero suppression scheme. So we can compute the result of α in this case also. □

In the following theorems, we intend to prove the correctness of the two optimizations we introduce in Section 3.5. So for the following theorems, we assume that we are using only Boolean `AND` and `OR` operators in a subscription.

Theorem 5. *If a non-leaf node α is black, at least one of its two children nodes will be black.*

Proof. According to our previous definition, α is black because the computed result in α is `true`. In the simplified case where we use Boolean `AND` and `OR` operators, the result of a non-leaf node α can be true in two cases. First, if the operator is Boolean `AND` and the computed results of both children β and θ (Figure A.3) are true (Table A.1). So both β and θ are black.

$Operand_1$	$Operand_2$	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Table A.1: Truth table for two Boolean operators

Second, if the operator is Boolean `OR`, the result of α can be `true` if the computed result in anyone of its children is `true`. For both cases, at least one of the two children will have `true` in it. So α will have at least one black child. □

Theorem 6. *If a subscription S is satisfied, result of the root node can be computed without using the results of white nodes of the tree.*

Proof. We assume a binary tree T_S to be a tree representation of a subscription S. We know that if the computed result of the root node of T_S ($root(T_S)$) is `true`, the subscription S is satisfied. Result of every non-leaf node α is computed using the results of the two children nodes β and θ (Figure A.3). If the operator of α is `AND` and α is black, both β and θ are black. We need the results of β and θ to set α to black. But for Boolean `OR`, there can be two ways to set α to black. First, both β and θ can be black, which is same situation as we described for Boolean `AND`. Second, one child (let β) can be black and the other one (θ) is white, e.g. β is `true` and θ is `false` . From Table A.1, we know that we can still compute the result of α. The node α can be set to black using the result of β only, without using the result of θ. This is true for every non-leaf node in the tree T_S. Hence starting from the leaves, we can propagate the `true` values to parents and compute the result at each non-leaf node without using the results of white nodes. □

Figure A.4 and A.5 show two colored tree examples of the subscriptions $S_1 = (A \vee B) \wedge (C \vee D)$ and $S_2 = (A \vee B) \vee (C \wedge D)$. For both of them, we assume that only the predicates C and D are satisfied by an incoming event. The shaded nodes are the black nodes in each example.

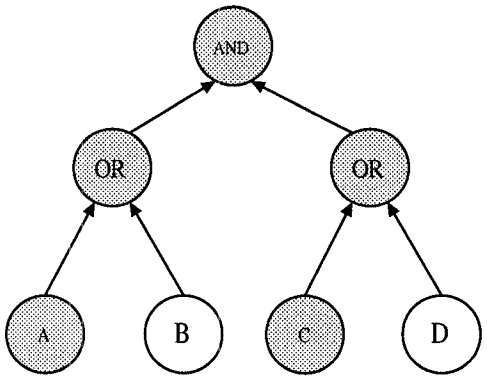


Figure A.4: Example subscription tree for subscription S_1

Theorem 7. *If a subscription S is satisfied, there is at least one black path from $root(T_S)$ to one leaf node of T_S.*

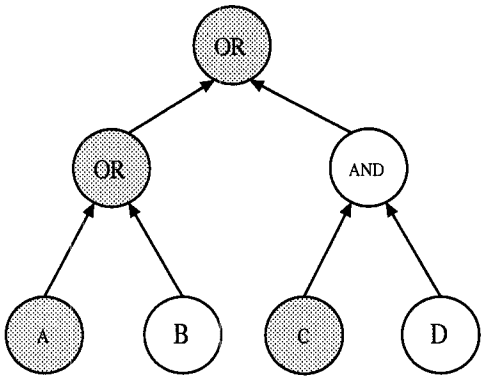


Figure A.5: Example subscription tree for subscription S_2

Proof. As S is satisfied, result of $root(T_S)$ is `true` and $root(T_S)$ is black. From Theorem 5 we know that at least of its two children will be black. If we deduce the same fact for the black child of $root(T_S)$, we will again find at least one black child for it also. This continues until we find a leaf node (λ), which is black. Hence there is always at least one black path from the root node to one leaf node. □

Corollary 4. *Zero Suppression finds every matched subscription.*

Proof. We assume any arbitrary subscription S that should be satisfied by an incoming event. As S will be satisfied after matching the event, the root node of T_S will be black after event matching. According to Theorem 7, there must be at least one black path starting from $root(T_S)$ and ending at a leaf node λ. Node λ contains a predicate that is satisfied by the current event.

In our Zero Suppression algorithm, we compute the result for every black node and skip the computation of result for the white nodes of S_T. The algorithm starts from the matched leaves, obviously including λ. Our propagation of `true` value will flow in reverse direction along with the black path that starts from $root(T_S)$, and the algorithm will definitely set $root(T_S)$ to `true` at some point of its execution. Although the algorithm does not compute the results of some white nodes, we still get correct result (Theorem 6). So Zero Suppression can find any matched subscription. □

Corollary 5. *True Value Path finds every matched subscription.*

Proof. We can deduce this directly from Corollary 4. In the same situation we described in the previous proof, we will definitely find the black path. So when we start the execution of TVP from λ, we will reach $root(T_S)$ and set it to `true` or black. So TVP always finds every matched subscription. □

Bibliography

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18-th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–61, 1999.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[3] David E. Bakken. *Middleware.* Kluwer Academic Press, 2001.

[4] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23-rd International Conference on Software Engineering (ICSE '01)*, pages 443–452, 2001.

[5] D. Chamberlin. Xquery: An XML query language. In *IBM Systems Journal*, volume 41. 2002.

[6] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with xpath expressions. In *Proceedings of the 18-th IEEE International Conference on Data Engineering*, pages 235–244, February 2002.

[7] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proceedings of the 28-th International Conference on Very Large Databases*, 2002.

[8] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. pages 379–390, 2000.

[9] J. Clark and S. DeRose. *XML Path Language (XPATH) Version 1.0.* W3C Recommendation, http://www.w3.org/TR/xpath, November 1999.

[10] Renee de la Briandais. File searching using variable length keys. In *Proceedings of the American Federation of Information Processing Societies Western Joint Computer Conference*, pages 295–298, Mar 1959.

[11] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks*, 31(11–16):1155–1169, 1999.

[12] Y. Diao, M. Altinel, M. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. In *Proceedings of the ACM Transactions on Database Systems*, volume 28, pages 467–516, Dec 2003.

[13] 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.

[14] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Efficient matching for content-based publish/subscribe systems. In *Proceedings of the 8-th International Conference on Cooperative Information Systems*, 2000.

[15] Françoise Fabret, H.-Arno Jacobesen, François Llirbat, Joao Pereira, Kenneth Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, 2001.

[16] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[17] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18-th International Conference on Very Large Databases*, pages 327–338, 1992.

[18] D. Gelernter. Generative communication in linda. *ACM Computing Surverys*, 7(1):80–112, January 1985.

[19] K. J. Gough and G. Smith. Efficient recognition of events in distributed systems. In *Proceedings of the 18-th Australasian Computer Science Conference, Adelaide, Australia*, Feb 1995.

[20] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics - A Foundation for Computer Science.* Addison-Wesley Pub Co, 1994.

[21] H.-A. Jacobsen H. Leung. Subject space: A state-persistent data model for publish/subscribe systems. In *Computer Science Research Group, University of Toronto, CRSG, nb. 459*, September 2002. (Appeared as student paper in CASCON 2002).

[22] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *Proceedings of the 15-th International Conference on Data Engineering*, pages 266–275. IEEE Computer Society Press, 1999.

[23] Eric N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS '91)*, pages 153–164, 1991.

[24] Hans-Peter Kriegel, Marco Potke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of the 26-th International Conference on Very Large Databases*, pages 407–418, 2000.

[25] Laks V. S. Lakshmanan and Sailaja Parthasarathy. On efficient matching of streaming XML documents and queries. In *Proceedings of the 8-th International Conference on Extending Database Technology*, pages 142–160, 2002.

[26] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowledge and Data Engineering*, 11(4):610–628, 1999.

[27] Sun Microsystems. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, 1988.

[28] B. M. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus—an architecture for extensible distributed systems. In *Proceedings of the 14-th Symposium on Operating Systems Principles*, pages 58–68, Dec 1993.

[29] Milenko Petrovic, Ioana Burcea, and Hans-Arno Jacobsen. S-topss: Semantic toronto publish subscribe system. In *Proceedings of the 29-th Very Large Databases Conference*, Sep 2003.

[30] David Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996.

[31] B.R. Rao. Making the most of middleware. In *Data Communications International*, pages 89–96. Sep 1995.

[32] Michael Shoffner. Write your own MOM! In *JavaWorld*. 1998.

[33] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–334, 1994.

[34] Tak W. Yan and Hector Garcia-Molina. Distributed selective dissemination of information. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, pages 89–98, 1994.