Research Report

Event Matching in Symmetric Subscription Systems

Walid Rjaibi,[1] Klaus R. Dittrich,[2] and Dieter Jäpel[1]

[1]IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

[2]Database Technology Research Group
Department of Computer Science
University of Zurich
8057 Zurich
Switzerland

IBM **Research**
Almaden · **Austin** · **Beijing** · **Delhi** · **Haifa** · **T.J. Watson** · **Tokyo** · **Zurich**

Event Matching in Symmetric Subscription Systems

Walid Rjaibi[*]   Klaus R. Dittrich[†]   Dieter Jaepel[*]

Abstract

Publish/subscribe and database systems researchers have recognized the importance of the event matching algorithm to the performance and scalability of a content-based subscription system. A number of interesting event matching techniques as well as DBMS solutions have been proposed in recent research work in the area. Content-based subscription systems allow information consumers to define filtering criteria when they register their interest in being notified of events that match their requirements. However, information producers are not offered the same flexibility. Moreover, content-based subscription systems require that the value of each attribute in the event schema be known at publication time. Certain types of information producers need to be given the flexibility of selecting what types of information consumers can receive their publications as well as the ability to personalize their publications at individual attribute level. This paper introduces symmetric subscription systems as the next generation of content-based subscription systems which addresses the above issues, and proposes a novel event matching algorithm in this context.

1   Introduction

A *Publish/subscribe* system connects together information producers, which publish events to the system, and information consumers, which subscribe to particular types of events within the system. The system is responsible for identifying the set of subscriptions that are matched by a published event (if any), and for notifying the corresponding subscribers. The earliest publish/subscribe systems were subject-based. In such systems, information consumers subscribe to one or more subjects and the system notifies them each time an event classified as belonging to one of the subjects they subscribed to is published. Event matching is a straightforward task in these systems because events can be filtered only according to their subject. Any additional event filtering has to be done by the subscriber himself. Examples of such systems are OrbixTalk [15], TIB/Rendezvous [16] and CORBA [17].

An attractive alternative to subject-based systems is content-based subscription systems. These systems appear to be more promising in meeting subscriber needs of defining filtering criteria (or predicates) when they register their interest in receiving publications. Compared to subject-based systems, content-based systems allow subscribers to express a "query" against the content of a published event. Examples of content-based systems are Le Subscribe [6, 7], Gryphon [14, 11], NEONet [18] and READY [19]. Le Subscribe, Gryphon, NEONet, READY and, in general, most content-based subscriptions systems use quite similar publication and subscriptions languages[1]. In these systems, an event is distinguished based on its *event schema* [14]. An event schema defines the type of the information contained in each event, and the system usually supports multiple event schemas. For example, a content-based system for a book market may define an event schema as a tuple containing three attributes: A title, an author and a price of string, string and float types respectively. A subscription is then a conjunction of predicates, such as (author = "Al-Kharezmi") and (price < $100).

Content-based subscription systems allow information consumers to define filtering criteria so that they are only notified of events that match their requirements. However, information producers are not offered

---
[*]E-Business Solutions Research Group, IBM Zurich Research Laboratory, {rja, jae}@zurich.ibm.com

[†]Database Technology Research Group, Department of Computer Science, University of Zurich, dittrich@ifi.unizh.ch

---
[1]These systems differ from each other mainly by the richness of the set of comparison operators their subscription languages provide.

the same flexibility. This might be acceptable for simple applications like our book market example. But in other applications like an insurance market for example, it is certain that information producers will require the ability to define filtering criteria so that only information consumers who most closely match their criteria are sent the event notifications. Moreover, content-based subscription systems require that the value of each attribute in the event schema be known at publication time. In some applications, the value of one or more attributes may not be known at publication time because it is case or information consumer specific and can not be specified in advance. For example, an auto-insurance company might want to vary the price of an insurance offer depending on the driver's age and driving history.

To address the needs of a new range of applications such as insurance markets, the next generation of content-based subscription systems must be flexible enough to allow the information producers to select what types of information consumers can receive their publications as well as to personalize their publications at the individual attribute level. We will refer to such systems as *symmetric* subscription systems because they allow both the information consumer and the information producer to define filtering criteria. The cost of the gain in flexibility in a symmetric subscription system is an increase in the complexity of the event matching algorithm. Therefore, efficient event matching techniques are needed to achieve scalabilty in a symmetric subscription system.

The rest of this paper is organized as follows. Section 2 formally defines the event matching problem. Section 3 reviews event matching algorithms in content-based subscription systems. Section 4 introduces symmetric subscription systems and proposes a novel event matching algorithm in this context. Finally, conclusions and future research direction suggestions are presented in section 5.

2 The Event Matching Problem

The event matching problem can be expressed as follows. Given an event e and a set of subscriptions $\mathcal{S}$ determine all subscriptions in $\mathcal{S}$ that are matched by e. The definition of an event and a subscription is different depending on whether the subscription system is content-based or symmetric. In the next subsections we review the definition of an event and a subscription in a content-based subscription system, and propose our new definition of an event and a subscription in the context of a symmetric subscription system.

2.1 Events and Subscriptions in Content-based Subscription Systems

An event e is a set of (A_i, V_i) pairs, where A_i is a valid attribute name as defined in the event schema and V_i is a valid data value with respect to A_i's data type. A subscription s is a conjunction of predicates, each of which is a triplet $(A_i, \text{operator}, V_i)$. Operator is a comparison operator supported by the subscription language. The relational operators $\{=, !=, <, <=, >, >=\}$ are normally supported by all subscription languages. Content-based subscription systems usually support a few additional specific operators. For example, Le Subscribe [7] supports two additional specific operators called *kind of* and *contains*. See [19, 14, 11, 18] for further examples.

An event e matches a subscription s if every predicate in s is satisfied by some pair (A_i, V_i) in e. For example, the event {(subject, "Algorithms"), (author, "Al-Kharezmi"), (price, $50)} matches the following subscription which is expressed as a conjunction of two predicates: (author = "Al-Kharezmi") and (price $< 100).

2.2 Events and Subscriptions in Symmetric Subscription Systems

A symmetric subscription system extends the event definition above by adding an optional conjunction of predicates. Each predicate is a triplet $(A_i', \text{operator}, V_i')$, where A_i' is a valid attribute name as defined in the *subscription schema* and V_i' is a valid data value with respect to A_i''s data type. The subscription schema is the extension made to the subscription definition above so that a subscription in a symmetric subscription system can also include a set of (A_i', V_i') pairs that describe the information consumer. For example, a subscription schema for our book market example can be defined as a tuple containing two attributes: A name, an age of string and integer types respectively. An event can then contain a predicate on the age attribute so that a certain category of information consumers does not receive notifications when certain events are published.

An event e matches a subscription s if every predicate in s is satisfied by some pair (A_i, V_i) in e, and

every predicate in e is satisfied by some pair (A_i', V_i') in s. For example the event {(subject, "Adult Material"), (author, "Some Author"), (price, \$50), (age > 20)} matches the subscription {(name, "some customer"), (age, 45), (price < \$100)} because predicates from both sides are satisfied.

3 Event Matching in Content-based Subscriptions Systems

The most straightforward solution to identify the set of subscriptions that are matched by an event is a sequential search. In this method, the system traverses a list of subscriptions sequentially and tests each of them against the event. This solution might be acceptable when the total number of subscriptions and the average number of predicates per subscription are small. However, it clearly performs very badly when the number of subscriptions and the average number of predicates per subscription are large. Another major drawback to this solution is the overhead associated with the evaluation of the same predicate multiple times and the unnecessary evaluation of certain predicates. This can happen when multiple subscriptions have similar or interdependent predicates. In the first case, the same predicate is evaluated as many times as it appears in the set of subscriptions. In the later, a predicate p_j is needlessly evaluated when an interdependent predicate p_i has already been evaluated. For example, if the predicate (price < 20) is true then it is clear that the predicate (price < 100) is also true.

The problem of determining all the subscriptions that are matched by an event is similar to another problem encountered in artificial intelligence and active databases research some years ago. In artificial intelligence, forward-chaining expert systems must test each newly asserted fact against a collection of predicates to find those rules that match the fact. In active databases, each time a tuple t is updated, deleted or inserted into a table on which a trigger has been defined, the DBMS must determine whether or not t satisfies the trigger condition. If the trigger condition is satisfied the trigger action is executed. A number of predicate indexing techniques and testing network structures have been proposed in this context. See [12, 4] for an overview. These solutions have been the basis for developing a number of main memory algorithms to address the event matching problem in content-based subscription systems.

Event matching algorithms in content-based subscription systems can be divided into two categories: Predicate indexing based algorithms and testing network based algorithms. The solutions based on predicate indexing consist of two phases. The first phase determines all the predicates (in all subscriptions) that are satisfied by the event. The second phase finds all the subscriptions that are matched by the event based on the results of the first phase. The techniques based on testing networks pre-processes the set of subscriptions into a *matching tree*. Events enter the tree at the root node and are filtered through by intermediate nodes. An event that passes all intermediate testing nodes reaches a leaf node where a reference to a matching subscription is stored. In the following subsections, we present and compare examples of event matching algorithms from each category.

3.1 Predicate Indexing Based Algorithms

Algorithms that are based on predicate indexing techniques use a set of one-dimensional index structures to index the predicates in the subscriptions. They differ from each other by whether or not all the predicates in the subscriptions are placed in the index structures. The counting algorithm [7] is an example where all the predicates in the subscriptions are placed in the index structures. The Hanson *et al.* algorithm [4] is an example where not all the predicates in the subscriptions are placed in the index structures. The matching algorithms proposed in [7, 8] are examples of recent extensions to the Hanson *et al.* algorithm.

3.1.1 The Counting Algorithm

The counting algorithm [7] groups the predicates (from all subscriptions) into predicate families. A predicate family consists of predicates having the same attribute and the same comparison operator. For example, consider the two subscriptions defined as follows:
$S_1 \equiv (price = 100) \wedge (author = "Al - Kharezmi")$
$S_2 \equiv (price > 25)$
where $\equiv$ denotes logical equivalence. There are three predicate families in this example: (price =), (price >), and (author=). The counting algorithm allows various indexing structures to be used depending on the predicate family. For example, hashing is preferred for equality predicates because it provides fast access.

B+-trees or IBS-trees are more suitable for non equality predicates. When an event occurs, the algorithm uses the set of indexes built for each attribute in the event schema to determine all the predicates that are satisfied. Let P denote this set of predicates. Then, for each $p_i \in P$, the set of subscriptions containing p_i is fetched. For each one of these subscriptions, its number of satisfied predicates is incremented by one. Lastly, the algorithm goes through the set of all subscriptions and verifies whether or not all their predicates have been satisfied. All subscriptions whose number of satisfied predicates equals its total number of predicates is a matching subscription. To illustrate the algorithm, consider the following example of a system containing three subscriptions S_1, S_2, S_3 defined as follows:

$S_1 \equiv pr_1 \wedge pr_2 \wedge pr_3$
$S_2 \equiv pr_1 \wedge pr_2$
$S_3 \equiv pr_3 \wedge pr_4$

Assume that an incoming event e of event schema ES_3 satisfies predicates pr_1 and pr_2 but does not satisfy pr_3 and pr_4. Further assume that predicates pr_1, pr_2, pr_3 and pr_4 involve attributes A_1, A_2, A_3 and A_4 respectively. Figure 1 describes the data structure that could be used to match events in a content-based subscription system using the counting algorithm.
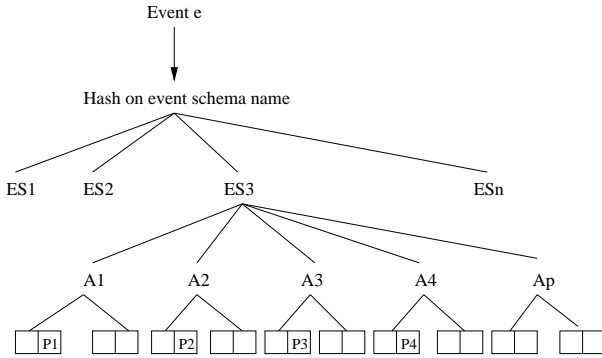

Figure 1: **Event matching using the counting algorithm**

The incoming event is first filtered based on its event schema [2]. Next, the algorithm concludes that the set of predicates that match event e is composed of predicates pr_1 and pr_2. During the second phase, the algorithm first determines the set of subscriptions involving pred-

---

[2] A content-based subscription system usually handles multiple event schemas.

icates pr_1 and pr_2. These are subscriptions S_1 and S_2. Next, the algorithm increments the number of satisfied predicates for S_1 and S_2. Subscription S_2 is retained because its number of satisfied predicates equals its total number of predicates. However, subscription S_1 is rejected because its number of satisfied predicates is less than its total number of predicates.

3.1.2 The Hanson *et al.* Algorithm

The Hanson *et al.* algorithm [4] was proposed in the context of active databases but it can also be applied to determine what subscriptions match an event. In a pre-processing step, the algorithm builds a set of one-dimensional indexes, one for each attribute in the event schema. For each subscription, the most selective predicate is placed in the corresponding one-dimensional index. This one-dimensional index is a balanced IBS-tree which allows efficient searching to determine which interval and equality predicates match a value. An *interval binary search tree* (IBS-tree) is a binary search tree augmented with additional information to make it possible to find all intervals that overlap a point. See [4] for a detailed description of IBS-trees. When an event occurs, the algorithm uses this set of one-dimensional indexes to determine all the subscriptions whose most selective predicate is satisfied. Then, for each one of these subscriptions, the remaining predicates are evaluated to find out if there is a complete match. Note that considering only those subscriptions whose most selective predicate is satisfied is an improvement over the counting algorithm. Indeed, the counting algorithm incurs the overhead of considering all subscriptions in its second phase. This is because it needs to determine for each subscription whether or not its number of satisfied predicates equals its total number of predicates.

To illustrate the Hanson *et al.* algorithm, consider again the example from the previous section. Suppose that predicates pr_1 and pr_4 are the most selective predicates. Predicates pr_1 and pr_4 are therefore placed on the IBS-tree of attributes A_1 and A_4 respectively. Figure 2 describes the data structure that could be used to match events in a content-based subscription system using the Hanson *et al.* algorithm.

In the first phase of the algorithm, the IBS-trees for the corresponding event schema are searched for the values in the event. This first phase concludes that only
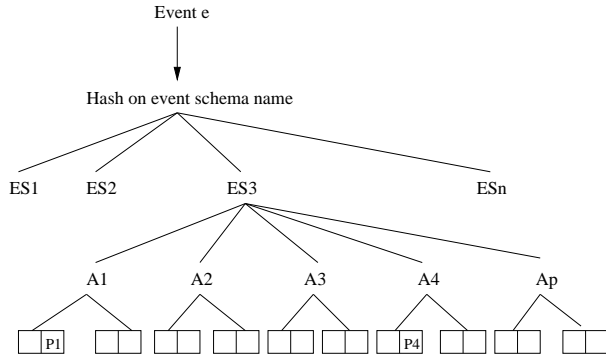
Figure 2: **Event matching using the Hanson** *et al.* **algorithm**

predicate pr_1 is satisfied. In its second phase, the algorithm concludes that only subscription S_2 is a matching subscription. Subscription S_1 is not a matching subscription because its predicate pr_3 is not satisfied. It is clear that subscription S_3 is not a matching subscription because its most selective predicate pr_4 is not satisfied.

3.1.3 Extensions to the Hanson *et al.* Algorithm

The propagation algorithm proposed in [7] optimizes the second phase of the Hanson *et al.* algorithm. In a pre-processing step, the subscriptions are grouped into clusters. Each cluster groups together the subscriptions having the same *access predicate* and the same number of predicates. The access predicate is the most selective predicate for each subscription in the cluster. This is the predicate placed in the IBS-tree. Within a cluster, the remaining predicates for each subscription are stored in decreasing order of selectivity, that is, from the 2nd most selective to the least selective predicate. In its second phase, the propagation algorithm considers only those clusters whose access predicate is satisfied. For each cluster, a propagation strategy is used to evaluate the rest of the predicates for the subscriptions contained in the cluster. This propagation strategy works as follows. First the 2nd most selective predicates are evaluated. Subscriptions which fail this step are eliminated. Those which succeed are considered for the next step where the 3rd most selective predicates are evaluated, and so on. A subscription which succeeds all steps is a matching subscription.

The matching algorithm proposed in [8] is similar to the propagation algorithm in that it also groups

subscriptions into clusters. However, it does not restrict the cluster's access predicate to be composed of a single predicate. The schema based clustering technique proposed groups together subscriptions in terms of their number of predicates and a common conjunction of equality predicates as an access predicate. The set of attributes referred to in the conjunction of equality predicate is called the *schema* of the access predicate. The algorithm uses multi-attribute hashing to find out the relevant clusters when an event occurs. The performance evaluation presented in [8] demonstrates that there can be advantages to this technique.

3.2 Testing Network Based Algorithms

The matching algorithm used in the Gryphon [5, 14] subscription system initially pre-processes the subscriptions into a matching tree. In the matching tree, each non-leaf node contains a test and edges from that node represent the results of that test. A leaf node contains a subscription. The matching tree can also have special "do not care edges" called *-edges that represent the fact that subscriptions reachable through the edge do not care about the result of a test. These edges are necessary when some of the subscriptions are independent of that test. When an event occurs, the algorithm walks the matching tree by performing the test prescribed by each node and by following the edge that represents the result of the test, and the *-edge if it is present. The set of matching subscriptions contains all those leaf nodes reached. To illustrate the algorithm, consider two subscriptions defined as follows:
$S_1 \equiv (price = 100) \wedge (author = ``Al - Kharezmi'')$
$S_2 \equiv (price > 25)$
Figure 3 describes the matching tree that corresponds to subscriptions S_1 and S_2. Note that subscription S_2 does not depend on the test prescribed at the root node of the matching tree and therefore a *-edge is required.

The Gough *et al.* matching algorithm proposed in [22] is similar to the Gryphon algorithm in that it also uses a matching tree. However, it allows a subscription to appear in more than a single leaf node. When an event occurs, the Gough *et al.* algorithm generally needs to follow several paths in the matching tree while Gryphon's algorithm follows a single path. Therefore, the Gryphon algorithm is more efficient. However, its matching tree is more space consuming. Compared to the predicate indexing based techniques, the matching tree based techniques are more space consuming.
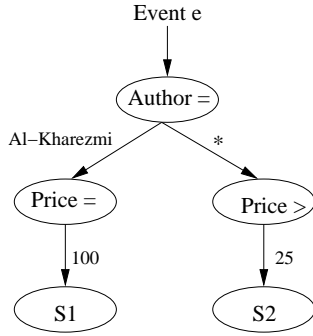
5

Figure 3: **Example of a matching tree**

Moreover, the matching tree data structure is more costly to maintain in systems where the rate of new subscriptions arivals (or subscription modifications) is quite high. Therefore, matching tree based techniques are better suited for systems where the subscription set is relatively stable during long periods of time.

4 Event Matching in Symmetric Subscription Systems

The advantages of a symmetric subscription system is that it allows the information producer to express filtering criteria and does not require the value of each attribute in the event schema to be known at publication time. The cost of this gain in flexibility is an increase in the complexity of the matching algorithm. The symmetry aspect requires that the predicates from both the information producer and the information consumer be satisfied before a subscriber is notified of a matching event. Allowing the value of an attribute to be determined at runtime rather than specified at publication time introduces another level of complexity. If determining the value of an event attribute A_i depends on some attribute A_i' from the subscription schema, then it is not possible to use a predicate indexing technique to determine in one step which subscription predicates $(A_i, \text{operator}, V_i)$ are satisfied. Moreover, the overhead of computing the value of A_i at runtime adds to the complexity of the event matching algorithm in a symmetric subscription system. Therefore, efficient event matching techniques are needed to achieve scalability in a symmetric subscription system.

The Websphere Matchmaking Environment (WME) [24, 25, 26] is an example of a symmetric subscription system. In the following subsections, we briefly introduce WME and present a novel event matching algorithm in this context.

4.1 Overview of Websphere Matchmaking Environment

WME is a set of components that allow the easy development, management and support of distributed automated matchmaking spaces for complex products and services. A key component of WME is its *symmetric matchmaking engine (MME)* which matches the needs of information consumers with the features of products and services advertised by information producers. The symmetry aspect stems from the fact that the matching process also takes into account the demands of information producers with respect to information consumers. Figure 4 describes the WME matchmaking engine symmetry aspect.
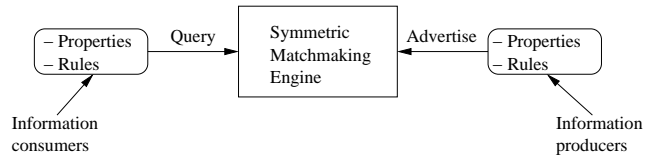

Figure 4: **Symmetric matchmaking engine**

Information producers use the WME advertising interface to advertise products and services into an MME. An advertisement consists of the following three arguments:

- **Matchmaking union name**: A string representing the type of the product or service advertised. It also defines the set of attributes that can be referred to in an advertisement or a query. The name must be a valid name as defined in the system's data dictionary. This is a mandatory argument.

- **Properties**: A list of (attribute, value) pairs describing the product or service advertised. The attribute must be a valid attribute name defined in the system's data dictionary for the product or service advertized. The value must be valid with respect to the attribute's data type. It can be specified explicitly if it will be known at advertising time or it will be computed dynamically during the matching process. In the later case, the value place holder consists of a program that will be used to compute the attribute's value during the matching process. This allows information producers to tailor their products and services for

6

each information consumer. An attribute whose value is not explicitly provided is referred to as a *dynamic* attribute. Similarly, an attribute whose value is explicitly provided is called a *static* attribute. Properties are a mandatory argument.

- **Rules**: A list of programs, each returning a boolean value, describing the demands of the information producer with respect to the information consumer. In WME, a program that is used to implement a rule or to compute the value of a property dynamically can be expressed in two ways: (1) as a *script* written in the WME script (programming) language, and (2) as a JAVA Call. A script is either evaluated locally by the MME if it was explicitly provided with the rule (or the property), or by another WME component called the *Script Processing Engine* (SPE) otherwise. JAVA Calls provide a mechanism to leverage legacy systems such as dynamic pricing engines and stock control systems. Rules are an optional argument.

Similarly, information consumers use the WME query interface to submit queries to the MME. The symmetric nature of the MME implies that the query also consists of three arguments: The matchmaking union name, properties and rules. In this case, properties describe the information consumer and rules describe the information consumer's filtering criteria.

To illustrate the above WME concepts, consider the following simple auto insurance example which shows how WME could be used to set up a matchmaking space where auto insurance companies advertise insurance offers and car drivers submit queries to obtain an insurance offer. Before the system can accept auto insurance advertisements and queries, the WME system administrator needs to create a matchmaking union for the auto insurance matchmaking space. The creation of the matchmaking union is accomplished using the WME data dictionary interface. The WME system administrator uses this interface to define the name of the union and its two record components. The first record defines the car driver looking for insurance and the second record defines an insurance offer. Let *AutoInsurance* denote the name of the matchmaking union, *Driver* denote the car driver record, and *Insurance* denote the insurance offer record. Tables 1 and 2 show the details of the *Insurance* and *Driver* records respectively.

Attribute	Type	Description
Name	String	Name of insurance company
Cover	String	Insurance coverage
Price	Float	Price of the insurance offer

Table 1: **Insurance record description**

Attribute	Type	Description
Name	String	Name of car driver
Age	Integer	Age of car driver
YearsNCD	Integer	Number of years driving without a claim

Table 2: **Driver record description**

The following are examples of advertisements. Advertisement A_1 does not specify any rules and the value of the price attribute is explicitly specified. Advertisement A_2 specify one rule and a script written in the WME script language that specifies how to calculate the value of the price attribute. The rule provided ensures that advertisement A_2 does not get returned as a result of a query submitted by a driver whose age is less than 40 or whose YearsNCD is less than 3. The script provided to calculate the value of the price attribute ensures that drivers who did not claim insurance for longer periods of time are rewarded.

- A_1 = (*AutoInsurance*, {(Name, "Company A"), (Coverage, "Basic"), (Price, $500)}, {})

- A_2 = (*AutoInsurance*, {(Name, "Company B"), (Coverage, "Full"), (Price, *if (YearsNCD > 7) price = 900; else price = 1200; return price;*)}, {*return (Age >= 40 && YearsNCD >= 3);*})

The following are examples of queries. Query Q_1 does not specify any rule and the value of the age attribute specified is 32. Advertisement A_2 can not be a match because Q_1 fails A_2's rule which specifies that the age attribute must be greater than 40. Therefore, only advertisement A_1 is a match for Q_1. Query Q_2 does not specify any rule but matches both A_1 and A_2. Note that the value of the price attribute for A_2 is $900 in this case. Query Q_3 is similar to Q_2 but specifies

a rule to indicate that advertisements where the price is greater than \$700 is not of interest. Therefore, only advertisement A_1 is a match for Q_3.

- Q_1 = (*AutoInsurance*, {(Name, "John"), (Age, 32), (YearsNCD, 3)}, {})

- Q_2 = (*AutoInsurance*, {(Name, "Bob"), (Age, 48), (YearsNCD, 8)}, {})

- Q_3 = (*AutoInsurance*, {(Name, "Chris"), (Age, 42), (YearsNCD, 8)}, {return (Price <= 700);})

4.2 The Event Matching Algorithm

4.2.1 Preliminaries

The matchmaking union name defines the set of attributes that can be referred to in an advertisement or a query. In the rest of this paper, we will refer to this set of attributes as the matchmaking union domain. Similarly, we will refer to the set of attributes that can be referred to in an advertisement and in a query as the advertisement domain and the query domain respectively. Let $\mathcal{A}$, $\mathcal{Q}$ and $\mathcal{M}$ denote the advertisement domain, the query domain and the matchmaking union domain respectively. Mathematically, we have $\mathcal{M} = \mathcal{A} \cup \mathcal{Q}$.

For a given matchmaking union name, an event e (an advertisement) and a subscription s (a persistent query) can be defined as follows.

$e = \{(a_1, v_1), (a_2, v_2), ..., (a_n, v_n), r_1, r_2, ..., r_p\}$
$s = \{(a'_1, v'_1), (a'_2, v'_2), ..., (a'_m, v'_m), r'_1, r'_2, ..., r'_q\}$

where $n, m >= 1$, $p, q >= 0$, $a_i \in \mathcal{A}$, $a'_j \in \mathcal{Q}$, $1 \leq i \leq n$, and $1 \leq j \leq m$. $r_1, r_2, ..., r_p$ are optional rules that describe the demands of the information producers with respect to the information consumer. $r'_1, r'_2, ..., r'_q$ are optional rules that describe the requirements of the information consumer with respect to the advertisements. In WME, these rules are programs expressed in the WME script language and can potentially refer to any attribute in the matchmaking union domain $\mathcal{M}$. However, we contend that in most practical situations, the rules in an event will refer to attributes in $\mathcal{Q}$ and the rules in a subscription will refer to attributes in $\mathcal{A}$. After all, the purpose of the rules in an event is to express predicates on the subscriptions and the purpose of the rules in a subscription is to express predicates on the events. Therefore, we formally define an event rule as a conjunction of predicates, each referring to an attribute in $\mathcal{Q}$, and a subscription rule as as a conjunction of predicates, each referring to an attribute in $\mathcal{A}$. A predicate is a triplet (attribute, operator, value)

as defined in section 2. In this paper we focus on the relational operators $\{=, ! =, <, <=, >, >=\}$ because they are common among all content-based subscription systems and they are sufficient to support most usual subscriptions for event notifications.

4.2.2 The Dynamic Attributes Cache

A dynamic attribute is an attribute whose associated value is to be calculated using a provided program. Recall from section 4.1 that this program can be executed locally by the MME, remotely by an SPE, or completly outside the WME system through a JAVA Call to some back-end system. The advantage of dynamic attributes is that it allows information producers to tailor their products and services for each subscriber. The cost of this flexibility is the overhead associated with evaluating the provided programs for each subscription. For example, the program associated with the attribute *Price* in advertisement A_2 of the previous example needs to be evaluated for subscriptions Q_1, Q_2 and Q_3. Another implication of this flexibility is that it is not possible to use a predicate indexing scheme to determine what predicates satisfy an (attribute, value) pair if the attribute is dynamic. For example, if Q_1, Q_2 and Q_3 all have a predicate on the attribute *Price* then it is not possible to determine in one step which of Q_1, Q_2 and Q_3 has its predicate satisfied. This is because the computation of the value of the attribute *Price* depends on each Q_i.

In the case of a JAVA Call or a script to be executed remotely by an SPE, the value place holder of an attribute will consist of the name of the program to be used for the calculation of the dynamic value. For example, this can be a standard program that computes the price of an auto insurance based on the driver's age and the number of years driven without an insurance claim. Observe that this program will have to be executed on all the subscriptions each time an event enters the system. Therefore, caching program results could be a significant performance improvement.

The dynamic attribute cache stores the results of program invocations. Each program has a unique program ID which is used to access the cache. In addition to the program ID, each cache entry includes the program signature and an invocation table. When a cache entry is accessed, the program signature is used to ensure that the current invocation of the program matches

its signature. If there is a mismatch then the program has been modified and the cache entry is invalidated. The invocation table stores the results of program invocations with different input parameter values. It is composed of two columns: A hash key and a program result. The hash key is the result of applying a multi-attribute hashing (MAH) function on the the input parameter values of a program invocation. The program result column stores the value returned by the program for those input parameter values. Figure 5 describes the dynamic attributes cache structure.

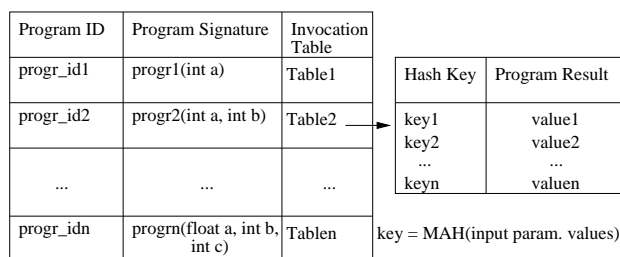| Program ID	Program Signature	Invocation Table		Hash Key	Program Result
progr_id1	progr1(int a)	Table1			
progr_id2	progr2(int a, int b)	Table2 →		key1 key2 ... keyn	value1 value2 ... valuen
...	...	...			
progr_idn	progrn(float a, int b, int c)	Tablen		key = MAH(input param. values)	

Figure 5: **The structure of the dynamic attribute cache**

When the value of a dynamic attribute needs to be determined, the program ID of the program associated with the dynamic attribute is looked up in the cache. If the program ID is found and the current invocation matches the program signature, the invocation table associated with that program ID is retrieved. Next, the cache's MAH function is used to compute the hash key for the input parameter values. This hash key is then searched in the invocation table. If the hash key is found then the associated program result is returned. Otherwise, the program is called to compute the desired result. This result is then entered in the invocation table with the associated hash key. If the invocation table is full the caching policy used is a standard Least Recently Used (LRU) policy. If the program ID is not found in the cache, then a cache entry is created for that program after it is called to compute the desired result. If no cache entry is available, an LRU policy is used to select a victim cache entry. When a program is removed or modified the cache entry depending on that program is invalidated.

Programs that are explicitly submitted with each event like advertisement A_2 in the previous example can not take advantage of the dynamic attribute cache because they are expected to be different for each event. Obviously, if they were intended to be the same program then there is a clear performance advantage in registering this program as an SPE program and only submit its name with each event.

4.2.3 The Event Matching Algorithm

The goal of an event matching algorithm is to efficiently determine the set of subscriptions that match an event. Our event matching algorithm initially preprocesses the subscriptions in a set of data structures that allow fast matching. The subscriptions are first didvided into two classes: Those that have predicates, like query Q_3 from the previous example, and those that do not. A subscription that does not have predicates need only to satisfy the event predicates to qualify as a matching subscription. But a subscription that has predicates can only qualify as a matching subscription if it satisfies the event predicates and its predicates are satisfied by the event itself. Figure 6 describes the steps of the algorithm.
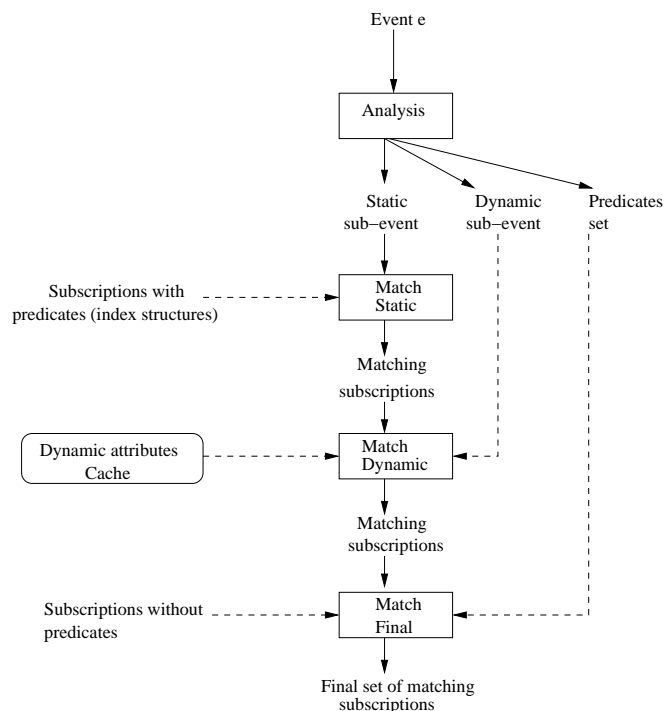

Figure 6: **The Event Matching Algorithm**

The first step in the algorithm is the "Analysis" process. This is the process where the event is analyzed and divided into two sub-events and a predicates set. The first sub-event is composed of the sub-set of static

attributes and their corresponding values from the original event. The second one consists of the the sub-set of dynamic attributes and their corresponding program names. In the rest of this paper, we will refer to these two sub-events as the static sub-event and the dynamic sub-event respectively. The predicates set represents the event's predicates.

The next step is the "Match Static" step, which determines all the subscriptions that match the static sub-event. Observe that the set of subscriptions that would match the static sub-event is the union of the subscriptions that do not have any predicates and those that have predicates satisfied by the static sub-event. The "Match Static" step only considers those subscriptions which have predicates. To efficiently determine the matching subscriptions at this step, the subscriptions' predicates are organized in a predicate index structure as detailed in section 3. A testing network based algorithm could also be used during this step. The matching subscriptions are then fed into the "Match Dynamic" step. This is the step where the subscriptions that have one or more predicates depending on a dynamic attribute are further processed. The dynamic attributes cache is used to efficiently determine the value of each dynamic attribute. Subscriptions that pass the "Match Dynamic" step are fed into the "Match Final" step together with the subscriptions that do not have any predicates. The "Match Final" step applies the event predicates over these subscriptions to determine the final set of matching subscriptions. For example, the predicates (Age $>=$ 40 && YearsNCD $>=$ 3) in advertizment A_2 from the previous example are evaluated at this step. To efficiently process this step, our event matching algorithm creates a B+-tree for each attribute in the query domain $\mathcal{Q}$ during the pre-processing step. These B+-trees are used to efficiently determine all those subscriptions that satisfy the event's predicates.

5 Conclusions and Future Directions

The event matching algorithm is critical to the performance and scalability of a subscription system. This paper has reviewed a number of interesting event matching algorithms that have been proposed in recent research work in the area of subscription systems. These algorithms can be classified into two categories: Algorithms based on predicate indexing techniques and algorithms based on matching trees. Com-

pared to the predicate indexing based techniques, the matching tree based techniques are more space consuming. Moreover, the matching tree data structure is more costly to maintain in systems where the rate of new subscriptions arrival (or subscriptions modification) is quite high. Therefore, matching tree based techniques are better suited for systems where the subscription set is relatively stable during long periods of time.

This paper also introduced symmetric subscription systems as the next generation of content-based subscription systems and presented Websphere Matchmaking Environment (WME) as an example of a symmetric subscription system. Symmetric subscription systems can be viewed as content-based subscription systems augmented with additional functionality to address the needs of a new range of applications such as insurance markets. They allow the information producers to select what types of information consumers can receive their publications as well as to personalize their publications at the individual attribute level. The cost of this gain in flexibility is an increase in the complexity of the matching algorithm. The symmetry aspect requires that the predicates from both the information producer and the information consumer be satisfied before a subscriber is notified of a matching event. Allowing the value of an attribute to be determined at runtime rather than specified at publication time introduces another level of complexity. The dynamic attributes cache proposed in this paper reduces the cost of personalizing publications at individual attribute level. Our event matching algorithm takes advantage of the dynamic attribute cache.

References

[1] P. Bernstein et al. The asilomar report on database research. *ACM SIGMOD record*, 27(4), 1998.

[2] E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parasarathy, J. Park and A. Vernon. Scalable Trigger Processing. In *Proc. of the International Conference on Data Engineering*, 1999.

[3] J. Chen, D. Dewitt, F. Tian and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. of the ACM SIGMOD Conf. on Management of data*, 2000.

[4] E. Hanson, M. Chaabouni, C. Kim and Y. Wang. A predicate matching algorithm for database rule systems. In SIGMOD'90, 1990.

[5] M. Aguilera, R. Strom, D. Sturman, M. Astley and T. Chandra. Matching events in a content-based

subscription system. In *Eighteen ACM Symposium on Principles of Distributed Computing (PODC'99)*, 1999.

[6] J. Pereira, F. Fabret, F. Llirbat, R. Preotiuc, K. Ross and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proc. of the 26th VLDB Conference*, 2000.

[7] J. Pereira, F. Fabret, F. Llirbat and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proc. of the Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS'2000)*, Eilat, Israel, September 2000.

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

[9] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. of AUUG'97*, Brisbane, Australia, September 1997.

[10] Y. Huang and H. Garcia-Molina. Publish/Subscribe in a Mobile Environment. In *Proc. of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE01)*, Santa Barbara, California, USA, May, 2001.

[11] G. Banavar, T. Chandra and B. Mukherjee. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the 19th International Conference on Distributed Computing Systems*, 1999.

[12] E. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review, vol. 8 no. 2*, June 1993 1999.

[13] A. Garzaniga, D. Rosenblum and A. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of the 19th Annual ACM SIGACT-SIGOPS Symposium on PRINCIPLES OF DISTRIBUTED COMPUTING (PODC)*, Portland, Oregon, USA, 2000.

[14] IBM Research. The Gryphon project home page, http://www.research.ibm.com/gryphon/home.html.

[15] IONA Technologies. OrbixTalk. *http://www.iona.com/products/messaging/index.html*.

[16] A. Chan. Transactional publish/subscribe: The proactive multi-cast of database-changes. In *SIGMOD'98*, 1998.

[17] Object Management Group. Common object service specification. *Technical report, Object Management Group*, 1998.

[18] New Era of Networks Inc. *http://www.neonsoft.com/products/NEONet.html*.

[19] R. Gruber, B. Krishnamurthy and E. Panagos. The architecture of the ready event notification service. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[20] J. Pereira, F. Fabret, H. Jacobsen and F. Llirbat. Web-Filter: A high-throughput XML-based publish and subscribe system. In *Proc. of the 27th VLDB Conference*, Roma, Italy, 2001.

[21] E. Hanson. Rule condition testing and action execution in Ariel. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, June 1992.

[22] K. Gough and G. Smith. Efficient recognition of events in distributed systems. In *Proc. of the ACSC-18*, 1995.

[23] L. Liu, C. Pu and W. Tang. Continual queries for internet scale event-driven information delivery. *TKDE 11(4)*, 1999.

[24] Y. Hoffner, C. Facciorusso, S. Field and A. Schade. Distribution issues in the design and implementation of a virtual marketplace. *Computer Networks 32(2000)*, 2000.

[25] IBM Research. *http://www.zurich.ibm.com/wme*.

[26] W. Rjaibi. The Design of the WebSphere Matchmaking Environment Query and Advertising Agents. *Technical Report*, IBM Zurich Research Laboratory, March 2002.