Efficient Matching for State-Persistent Publish/Subscribe Systems

Hubert Ka Yau Leung
IBM Toronto Lab
hkyleung@ca.ibm.com

Hans-Arno Jacobsen
University of Toronto
jacobsen@eecg.toronto.edu

Abstract

Content-based publish/subscribe systems allow information dissemination and fine-grained information filtering in loosely coupled distributed systems. Stateless publish/subscribe systems send notifications to all subscribers whose subscriptions match an incoming publication. State-persistent publish/subscribe systems, a recently proposed model that stores the states of both publications and subscriptions, only send notifications upon state transitions. The information filtering process requires an efficient matching algorithm with high throughput and scalability. Although there have been studies on matching algorithms for stateless publish/subscribe systems, the matching problem for state-persistent publish/subscribe systems is still an open research problem. This paper presents a novel content-based matching algorithm and its data structures for state-persistent publish/subscribe systems. We will also present the complexity analysis and results of simulations that validates the analytical predictions.

1 Introduction

In a publish/subscribe system, publishers are information providers and subscribers are information consumers. Publishers send information in the form of publications to a broker, which acts as a mediator between the publishers and subscribers. Subscribers maintain a set of subscriptions at the broker, indicating situations in which they would like to be notified. A broker uses the publications and subscriptions to decide when to send notifications and who should receive them.

Algorithms used at a broker to compare publications with subscriptions and to determine which subset of subscribers requires notifications are commonly known as matching algorithms. Matching algorithms are dependent on the data model used by the publish/subscribe system. A data model defines how information is represented and structured in the system. In channel-based publish/subscribe systems, for instance, subscribers subscribe to channels and receive all publications published to these channels. Topic-based publish/subscribe systems structure information in hierarchical structures that categorize information by their topics and level of granularity. The relationships between topics can be deduced by their relative positions in the topic hierarchy. Solutions for channel-based or topic-based publish/subscribe systems have matured, resulting in several academic and industrial-strength solutions. TIBCO Rendezvous [8], implementations of CORBA Notification Service [14] and Java Messaging Service [11] all fall into the channel-based or topic-based categories.

Another alternative, and a much more flexible model, is the content-based model. In a content-based publish/subscribe system, subscriptions are specified as expressions evaluated over the publication contents. This approach provides more expressive power to filter publications and is more easily customized for individual subscribers. The increased expressiveness comes with the cost of a higher complexity in the matching algorithm. Many application domains of publish/subscribe systems have a high event

rate and a large number of subscriptions. A highly optimized matching algorithm is required to achieve a high throughput and to scale up to support a larger number of users. The more expressive are the subscription languages, the higher is the computational complexity of the matching algorithm. Hence, the desire to provide a more flexible subscription language and the need to improve the performance of the system are two contradicting goals. It is therefore important for publish/subscribe systems to strike a balance between performance and expressiveness allowed for publications and subscriptions.

The design of efficient, content-based matching algorithms has been an active research topic. Many algorithms have been proposed [2,3,4,10,12]. These algorithms use different data structures and try to optimize the algorithms by imposing restrictions on the expressiveness of content-based subscriptions. A commonality between the existing algorithms is that they are all designed for stateless publish/subscribe systems. Recently, the *subject space model*, a state-persistent model for publish/subscribe systems, has been proposed to be a more elegant model that can overcome limitations of traditional publish/subscribe systems [6]. A state-persistent publish/subscribe system stores the values of both publications and subscriptions, and maintains the states between each pair of publication and subscriptions. It only sends notifications upon state transitions. The requirements of a state-persistent publish/subscribe system bring additional challenges to the already complicated stateless matching problem. The major contribution of this paper is to present the algorithms and data structures required for solving the state-persistent matching problem.

The advantages of the proposed algorithms over existing algorithms and the contributions of this research are:

a) Publications can specify range values. All existing content-based matching algorithms assume values of publications are point data.
b) The algorithm does not give precedence on equality predicates to optimize performance, as many existing algorithms do. This algorithm performs equally well on equality predicates and range queries.
c) The subject space model and the matching algorithm distinguishes between inserting a new publication and updating an existing publication in the publish operation. Performance evaluation has shown that the update operation performs much better than the insertion operation. This is an advantage of the state-persistence and existing publish/subscribe systems cannot support this feature.
d) The complexity of the matching algorithm for the subject space model is independent of the total number of subscriptions of the system but is sensitive to the usage behaviors. This property of the algorithm allows a publish/subscribe system to scale up to support a large number of users.

This paper is organized as follows: Section 2 gives a more detailed discussion on state-persistence in publish/subscribe systems. Section 3 gives a concise definition of the subject space model as the background of this study. Section 4 describes the matching problem for state-persistent publish/subscribe systems. Sections 5 and 6 present the algorithms and data structures for indexing states and values. Section 7 presents the algorithms for the operations in a state-persistent publish/subscribe system. Sections 8 and 9 present the analytical and empirical performance evaluation of the proposed algorithm and show how this algorithm is superior to existing content-based matching algorithms.

2 Persistence in Publish/Subscribe Systems

Persistence is an important, yet overlooked, aspect of publish/subscribe systems. It refers to the storage of *data* and *states* of publish/subscribe systems. There are two types of data in a publish/subscribe system – publications and subscriptions. A system that only makes publications persistent behaves exactly like a conventional database, since subscriptions can only be executed once. The definition of a publish/subscribe system therefore requires that subscriptions be made persistent, but the persistence of publications is optional. States in a publish/subscribe system refer to the relationships between publications and subscriptions. Any given pair of publication and subscription can either be in the state of a match or mismatch.

2.1 Stateless

Publish/subscribe systems have traditionally saved neither the contents of publications, nor the states between publications and subscriptions. Most existing publish/subscribe systems are designed as stateless messaging systems. The objective of stateless systems is to deliver messages from publishers to subscribers. The contents of messages may be used as information for deciding who should get the messages, but the system does not keep them for future use.

Subscriptions in stateless publish/subscribe systems are *queries for future data*. Since the stateless publish/subscribe systems do not retain information on past information, it will only notify the subscriber about future messages that match the subscription.

Although publish/subscribe systems that only store subscriptions in the system are quite common and relatively easy to build when compared to data/state-persistent publish/subscribe systems, stateless publish/subscribe systems have limitations that prevent them from working efficiently or correctly in some application domains. Two limitations of stateless publish/subscribe systems are discussed below.

1. Without data persistence, a stateless publish/subscribe system does not allow publishers to publish a subset of the event attributes. Being able to publish only a subset of the event attributes is important because it can reduce network traffic and increase system efficiency by not republishing the values of attributes that have not been modified. In other words, stateless systems do not support update operations on published data.
2. Due to the lack of state-persistence, stateless publish/subscribe systems may result in a large amount of redundant notifications. Redundant notifications result because the update of the values in a publication or subscription may not necessarily change the states between them.

2.2 Data Persistence

With data persistence, all messages/events received by the broker component are stored, forming an event history. Publish/subscribe systems that allow data persistence store the data of both publications and subscriptions. This database capability sets it apart from pure messaging systems. Data is exchanged in the form of persistent objects, not transient messages.

Under the data persistence model, subscriptions can query on events both in the past and the future. Notifications sent to subscribers can contain existing information stored in the system that matches the requirement of the subscriptions. If events that match the subscriptions occur in the future, the subscriber will also be notified. Event correlation and causality reasoning can involve past and future events. Since historical data is available, publications just need to send updated information and do not need to resend unchanged data. Hence, the first limitation of the stateless model can be overcome.

It is common to use conventional relational databases as offline storage systems. Publish/subscribe implementations with message queues [7] or tuple space infrastructures [13] can also provide data persistence. The continuous sequence of messages entering the broker component of the system is referred to as a data stream. However, traditional databases are neither designed to store data streams efficiently, nor do they perform continuous queries on data streams. The STREAM research project from Stanford University [15] studies techniques for special storage management and query processing for data streams.

Although publish/subscribe systems under the data persistence model exhibit database behaviors, there is very little or no support for data update or deletion operations. Message queues and tuple space do not allow the update operation on stored data. These systems therefore behave like an *append-only* database. Although publications may serve to inform the system of a new piece of information, or to update the value of a stored record, the system does not distinguish between these two operations, and always appends the publication to the data as a new entry. Theoretically, append-only behavior implies that both the storage space of the system and the size of a notification are *unbound*, and thus poses implementation challenges. In practice, publications that are received more recently have more relevance and are more likely used to evaluate subscriptions. The system should discard older events without compromising the ability to evaluate continuous queries. A related problem of data expiration is studied in the context of data warehousing [5].

2.3 State Persistence

A state-persistent publish/subscribe system stores the states of publications and subscriptions. The state of a publish/subscribe system can be deduced from the historical data. The relationship between publications and subscriptions form the state of a publish/subscribe system. The relationship of a publication with respect to a subscription refers to whether it is a match with the subscription. Similarly, the relationship of a subscription with respect to a publication refers to whether it is a match with the publication.

In the stateless and data persistent publish/subscribe systems, the broker component notifies all subscribers whose subscriptions match with each incoming publication. In state-persistent publish/subscribe systems, however, the broker should only send notifications of a publication to those subscribers whose subscriptions undergo state transitions in the relationship with the publication. In other words, the broker component only notifies subscribers of publications that *enter* the states specified by their subscriptions.

State-persistent publish/subscribe systems have advantages over systems with data persistence only. Many applications only require that the most current value be kept, but not the event history. In geographical information systems, for example, if the requirement is to track the current locations of some moving entities, only the most updated version of the location information need to be stored, but not the event history. Since the location information is updated very frequently, persisting the data may result in storing a lot of useless information, and hence wasting much storage space. Unless the paths of movements are required for future use, storing the current values and the state information of publications and subscriptions is sufficient.

3 The Matching Problem

This section presents the formal problem statement, which is based on a refined version of the subject space model [6]. The subject space model defines the representation of information in state-persistent publish/subscribe systems. The matching algorithms are used to support the operations in subject spaces.

3.1 The Subject Space Model

We first look at the definitions of the subject space model. Under the subject space model, information is structured by subject spaces, which are metadata of the system. Intuitively, subject spaces are multidimensional spaces and data form regions in these spaces. Publications and subscriptions are declared as a correlation of the regions in the subject spaces.

Subject space

A subject space is a grouping for related publications and subscriptions that can be described by the same set of properties, and each dimension represents a property. We define a publish/subscribe system to be a set of subject spaces $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. The set of subject spaces is used for categorizing information in a publish/subscribe system. Subject spaces are the metadata of a publish/subscribe system and they help describe the values and relationships of publications and subscriptions.

Each subject space is defined as a tuple $\sigma = (D_\sigma, V_\sigma)$, where $D_\sigma = \{d_1, d_2, \ldots, d_n\}$ is the set of dimensions of the subject space and V_σ is the set of values allowed in this space. A subject space is a multidimensional space. Each dimension is defined as a tuple $d = (name, type)$ where *name* is the unique identifier of the dimension, and *type* is a data type. Each dimension d has a domain of values, $dom(d)$, which is the set of all possible values that can be represented by *type*. This model allows each dimension of the multidimensional space to have a different domain. Examples of dimension data types include real numbers, integers, strings, Booleans and enumerated values. User-defined data types that are subsets of these data types are also possible. The domain of a subject space is the Cartesian product of the domains of its dimensions. $V_\sigma = dom(d_1) \times dom(d_2) \times \ldots \times dom(d_n)$.

Region

Data exist in subject spaces in the form of regions. Intuitively, data or regions occupy some volume within a subject space. Formally, a region is defined as a tuple $r = (C_r, V_\sigma^r)$. $C_r = \{c_1, c_2, \ldots, c_j\}$ is the set of constraints of r. A constraint is a subset of the domain of a given dimension. The set of values of constraint c in dimension d is denoted as $dom(c_d)$, i.e. $dom(c_d) \subseteq$

$dom(d)$. V_σ^r is the set of values of region r with respect to subject space σ. V_σ^r can also be interpreted as the spatial extension of region r with respect to σ. Denote the set of dimensions of C_r as D_{C_r}. Let $D_{C_r} \cap D_\sigma = \{ d_{i_1}, d_{i_2}, \ldots, d_{i_p} \}$, and $D_\sigma \setminus D_{C_r} = \{ d_{i_{p+1}}, d_{i_{p+2}}, \ldots, d_{i_n} \}$. If $D_{C_r} \cap D_\sigma \neq \varnothing$, then $V_\sigma^r = dom(c_{d_{i_1}}) \times \ldots \times dom(c_{d_{i_p}}) \times dom(d_{i_{p+1}}) \times \ldots \times dom(d_{i_n})$. Otherwise, $V_\sigma^r = \varnothing$. A region r is said to be *valid* in σ if $V_\sigma^r \neq \varnothing$. A region can be valid in multiple subject spaces.

Object Regions and Interest Regions

There are two types of regions – interest regions and object regions. They have the same definitions as a region but they have different semantics. An interest region represents the set of values within the subject spaces a subscriber is interested in. $\mathcal{I}$ denotes a set of interest regions, and i represents a particular interest region in $\mathcal{I}$. An object region represents values a publisher provides, the state of an entity that may be of interest to one or more subscribers. $\mathcal{O}$ denotes a set of object regions, and o represents a particular object region in $\mathcal{O}$.

Matching Relations Between Regions

Regions are spatial extensions within the subject spaces. Intuitively, two regions match if they overlap each other. Define **m** as a relation between regions r_1 and r_2 such that r_1 *matches* r_2. Formally, r_1 **m** r_2 iff $V_\sigma^{r_1} \cap V_\sigma^{r_2} \neq \varnothing$.

Filter

A filter is an integral part of both publications and subscriptions. The definition of a filter applies to both publications and subscriptions. A filter is expressed as

$$\{R \mid \exists\, r_1, r_2, \ldots, r_n \in R : P(r_1, r_2, \ldots, r_n)\}.$$

$R = \{r_1, r_2, \ldots, r_m\}$ is a set of regions and $P(r_1, r_2, \ldots, r_n)$ is a *boolean* function that takes a number of regions as variables. The expression represents the set of R such that P is *true*.

Subscription

A subscription specifies conditions for notifications. A subscription $\mathcal{S}$ is defined as a tuple $\mathcal{S} = (\mathcal{I}_S, f_S)$. $\mathcal{I}_S$ is a set of interest regions that represents the constraints of the subscription. These interest regions can be in different subject spaces. f_S is an expression that represents the set of sets of object regions that satisfy the conditions indicated by the rule of the filter. $f_S = \{\mathcal{O} \mid \exists\, o_1, o_2, \ldots, o_n \in \mathcal{O} : P(o_1, o_2, \ldots, o_n)\}$.

Publication

A publication targets content to a subset of the subscribers. A publication $\mathcal{P}$ is defined as a tuple $\mathcal{P} = (\mathcal{O}_P, f_P)$. $\mathcal{O}_P$ is a set of object regions that represents the constraints of the publication. These object regions can be in different subject spaces. f_P is an expression that represents the set of sets of interest regions that satisfy the conditions indicated by the rule of the filter. $f_P = \{\mathcal{I} \mid \exists\, i_1, i_2, \ldots, i_n \in \mathcal{I} : P(i_1, i_2, \ldots, i_n)\}$.

Matching Relations Between Publications and Subscriptions

Define $\mathcal{M}$ as a relation between a publication $\mathcal{P}$ and a subscription $\mathcal{S}$, such that $\mathcal{P}$ matches $\mathcal{S}$.

$(\mathcal{P}, \mathcal{S}) \in \mathcal{M}$ iff $\exists \mathcal{R} \subseteq \mathcal{I}_S : \mathcal{R} \in f_P \wedge \exists \mathcal{R} \subseteq \mathcal{O}_P : \mathcal{R} \in f_S$.

In order for a publication to match a subscription, some object regions of the publication must satisfy the subscription filter, *and* some interest regions of the subscription must satisfy the publication filter. If either of these two conditions is not met, this pair of publication and subscription is not a match. This demonstrates the symmetric property of publish/subscribe systems.

Notification Semantics

In a state-persistent publish/subscribe system, a broker only sends notifications upon state transitions. In other words, the broker sends notifications to a subscription $\mathcal{S}$ if a publication-subscription pair $(\mathcal{P}, \mathcal{S})$ is added to or removed from $\mathcal{M}$.

State transitions can take place in several situations, include adding a publication or subscription, updating a publication or subscription, and deleting a publication. Note that no notification needs to be sent if a subscription is deleted from the system. Below, we look at each of the operations that may trigger state-transitions and specify who should get the notification, if any, and what is the content of the notifications.

After adding a publication $\mathcal{P}$, send notifications to all subscriptions in $Q_P(\mathcal{P})$. The content

of the notification is $\mathcal{P}$ – the new publication. Define function Q_P as a mapping from a publication $\mathcal{P}$ to the set of subscriptions that matches $\mathcal{P}$. Formally, $Q_P(\mathcal{P}) = \{\mathbf{S} \mid \forall \mathcal{S} \in \mathbf{S} : (\mathcal{P}, \mathcal{S}) \in \mathcal{M}\}$.

After adding a subscription S, send notification to the subscription $\mathcal{S}$ if $Q_S(\mathcal{S})$ is non-empty. The content of the notification is $Q_S(\mathcal{S})$ – the set of all existing publications that satisfy the new subscription. Define function Q_S as a mapping from a subscription $\mathcal{S}$ to the set of publications that matches $\mathcal{S}$. Formally, $Q_S(\mathcal{S}) = \{\mathbf{P} \mid \forall \mathcal{P} \in \mathbf{P} : (\mathcal{P}, \mathcal{S}) \in \mathcal{M}\}$.

After updating a publication $\mathcal{P}$, send notifications to subscriptions in $Q_P^t(\mathcal{P})$. The content of the notification is $\mathcal{P}$ – the updated publication. After updating a publication $\mathcal{P}$, the system should notify the subscriptions that have become matches with $\mathcal{P}$. Let the update take place at time t. Assuming the time domain to be discrete, define function Q_P^t as a mapping from a publication $\mathcal{P}$ to the set of subscriptions that were not in $Q_P(\mathcal{P})$ when evaluated at t–1 and are members of $Q_P(\mathcal{P})$ when evaluated at t. Formally, $Q_P^t(\mathcal{P}) = Q_P(\mathcal{P})\big|_t \setminus Q_P(\mathcal{P})\big|_{t-1}$.

After updating a subscription $\mathcal{S}$, send notification to $\mathcal{S}$ if $Q_S^t(\mathcal{S})$ is non-empty. The content of the notification is $Q_S^t(\mathcal{S})$ – the set of publications that just became satisfied by the subscription after the update. After updating a subscription $\mathcal{S}$, the system should notify the updated subscription about publications that have become matches with $\mathcal{S}$. Let the update take place at time t. Assuming the time domain to be discrete, define function Q_S^t as a mapping from a subscription $\mathcal{S}$ to the set of publications that were not in $Q_S(\mathcal{S})$ when evaluated at *t–1* and are member of $Q_S(\mathcal{S})$ when evaluated at t. Formally, $Q_S^t(\mathcal{S}) = Q_S(\mathcal{S})\big|_t \setminus Q_S(\mathcal{S})\big|_{t-1}$.

3.2 The Matching Problem

Given the definitions of the subject space model in the previous section, the matching problem in a state-persistent publish/subscribe system is to store the values of publications and subscriptions, index the relationships between them, and to detect state transitions. Essentially, the matching algorithm is for deciding when to send notifications, which subscribers should receive notifications, and what to notify about, using the definitions of publications and subscriptions and their interactions. Unlike the stateless case, the fact that a publication matches a subscription, i.e. $(\mathcal{P}, \mathcal{S}) \in \mathcal{M}$, is not a sufficient condition for sending a notification. Notification is sent when a pair of publication and subscription is added or removed from the relation $\mathcal{M}$.

4 Operation Algorithms

The solution for the matching problem has two phases: *filtering* and *refinement*. In the filtering step, the broker uses the indexes to find a set of candidate publications or subscriptions to be used for evaluating the filter. Since operations on publications and subscriptions are symmetrical, we use the operations on publications to explain the filtering process without loss of generality. If a publication is to be added, updated or deleted, the filtering step eliminates subscriptions that will by no means become a match with this publication. The process narrows down the choices of subscriptions that will be used as parameters when evaluating the filter of the publication f_P. A good indexing algorithm should help perform the filtering process efficiently, and come up with a *necessary and sufficient* set of candidate subscriptions.

The refinement process will return a notification set, which is the set of subscriptions to be notified, or the set of publications to be included in the notification. Consider operations on publications. For each subscription from the result set of the filtering step, determine whether the publication-subscription pair is in the matching relation $\mathcal{M}$. If $(\mathcal{P}, \mathcal{S}) \in \mathcal{M}$, $\mathcal{S}$ will be added to the result set. Similarly, for operations on subscriptions, $\mathcal{P}$ will be added to the result set if $(\mathcal{P}, \mathcal{S}) \in \mathcal{M}$. Filters for both publication and subscription will be evaluated. If conditions of the publication and the subscription are satisfied, they are matched. Otherwise, we have a *false drop*.

The five operations supported by the subject space model were described and defined in Section 4. The functions $Q_P(\mathcal{P})$, $Q_S(\mathcal{S})$, $Q_P^t(\mathcal{P})$, $Q_S^t(\mathcal{S})$ and $\overline{Q}_P^t(\mathcal{P})$ determine which subscribers

should receive the notification and what is the information contained in the notification. Each of these operations may trigger state transitions, and each of them requires a different "matching algorithm". The algorithms of $Q_P(\mathcal{P})$ and $Q_P^t(\mathcal{P})$ are given Figures 1 and 2. Since publications and subscriptions are symmetrical, $Q_S(\mathcal{S})$ and $Q_S^t(\mathcal{S})$ can be obtained by inverting the roles of object regions and interest regions of $Q_P(\mathcal{P})$ and $Q_P^t(\mathcal{P})$ respectively. $\overline{Q}_P^t(\mathcal{P})$ is identical to $Q_P(\mathcal{P})$ except that the message of the notification is about deletion, as opposed to insertion, of a publication.

Algorithm $Q_P(\mathcal{P})$
Input: Publication $\mathcal{P} = (\mathcal{O}, f_P)$
Output: Notification set $\mathcal{N}$ – the set of matching subscriptions
Local Variables: Let $\mathcal{I}$ be the set of interest regions that matches $\mathcal{O}$, and let **S** be the set of candidate subscriptions.
(1) $\mathcal{I} = \varnothing$
(2) $\mathcal{N} = \varnothing$
(3) For each $o \in \mathcal{O}$ do
(4) $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{I' \mid \forall i \in I', i \mathbf{m} o\}$
(5) EndFor
(6) $\mathbf{S} = \{\mathcal{S} \mid \mathcal{S} = (I', f_S), \exists i \in I' : i \in \mathcal{I}\}$
(7) For each $\mathcal{S}$ in **S**,
(8) $\quad$ If $((\mathcal{P}, \mathcal{S}) \in \mathcal{M})$ then
(9) $\quad\quad \mathcal{N} \leftarrow \mathcal{N} \cup \{\mathcal{S}\}$
(10) $\quad$ EndIf
(11) EndFor

Figure 1 Algorithms for adding a publication (Q_P)

Algorithm $Q_P^t(\mathcal{P}_{upd})$
Input: The updated publication $\mathcal{P}_{upd} = (\mathcal{O}_{upd}, f_{P(upd)})$
Output: Notification set $\mathcal{N}$ – the set of subscriptions that was not a match with the publication before the update but will become a match with the publication after the update.
Local Variables:
Let $\mathcal{P}_{old} = (\mathcal{O}_{old}, f_{P(old)})$ be the state of the publication before update.
Let O' be the set of object regions that has been updated or newly included in the publication.
Let $\mathcal{I}$ be the set of interest regions that have undergone a state transition in the relationship with O'.
Let o'_{old} and o'_{new} be the states of an object region before and after update.
Let **S** be the set of candidate subscriptions.
(1) $\mathcal{I} = \varnothing, \mathcal{N} = \varnothing$
(2) $O' = \mathcal{O}_{upd} \setminus \mathcal{O}_{old}$,
(3) For each $o' \in O'$ do
(4) $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{I' \mid \forall i \in I', i \mathbf{m} o'_{new} \wedge (i, o'_{old}) \notin \mathbf{m}\}$
(5) EndFor
(6) $\mathbf{S} = \{\mathcal{S} \mid \mathcal{S} = (I', f_S), \exists i \in I' : i \in \mathcal{I}\}$
(7) For each $\mathcal{S}$ in **S**,
(8) $\quad$ If $((\mathcal{P}_{upd}, \mathcal{S}) \in \mathcal{M})$ then
(9) $\quad\quad \mathcal{N} \leftarrow \mathcal{N} \cup \{\mathcal{S}\}$
(10) $\quad$ EndIf
(11) EndFor

Figure 2 Algorithm for updating a publication (Q_P^t)

Lines 3 to 6 in both algorithms are executing the filtering step. The algorithms for gathering the candidate set for the addition and update operations differ. For the addition operations (Figure 1), line 4 collects the set of all regions that match at least one of the regions that define the publication or subscription. Line 6 produces a set of publications or subscriptions whose definitions include one or more of these regions, and this is the candidate set. In effect, the candidate set includes all subscriptions or publications that contain at least one region that matches a region of the publication or subscription to be added, respectively. The update operations, on the other hand, want to collect subscriptions or publications whose state with respect to the publication or subscription, respectively, may be changed after the update. Therefore, line 4 of the update algorithms (Figure 2) only collects regions that match with the newly added or updated regions of the publication or subscription to be updated.

Lines 7 to 11 in each algorithm are executing the refinement step. These algorithms assume that filters are defined as conjunctive queries. The rule of a filter is evaluated to true if each object region or interest region has a matching interest region or object region, respectively. Evaluating $(\mathcal{P}, \mathcal{S}) \in \mathcal{M}$ requires that all object regions of $\mathcal{P}$ have matches and all interest regions of $\mathcal{S}$ have matches.

5 Indexing States

This section presents an indexing technique for indexing the states between publications and subscriptions. The index is used in the filtering process by the subject space prototype.

5.1 Index Characteristics

Each index is responsible for indexing constraints along a dimension of the subject spaces. Recall that each dimension is associated with a data type that characterizes the data represented by that dimension. Index designs are dependent on data types. The index to be presented is de-

signed for dimensions whose data domains exhibit a total order. Examples of data types that have total order include integers, natural numbers and real numbers.

The subject space model defines a constraint as a subset of the domain of a given dimension. Although the use of sets is very powerful, set operations are very complex and inefficient. The implementation simplifies this matter by allowing constraints to be defined as intervals of values only. An interval of a domain with a total order is the subset of the domain with values bounded by two indicated values. The implementation allows the end points of an interval to be inclusive or exclusive. Under the assumption that constraints are expressed as intervals, the index must be able to store interval values efficiently, as opposed to data points only.

Furthermore, the index should also store state information. When an interval is updated, the index can efficiently report the intervals that are overlapping with the updated interval, but were not overlapping with the interval before the update.

5.2 Concepts

A one-dimensional interval can be represented as a point on a two-dimensional plane with the horizontal axis as the lower bound and the vertical axis as the upper bound (Figure 3). This transformation technique is studied by Seeger *et al.* in the context of spatial access methods [1].

All data points representing intervals should be in the upper left triangular portion of the plane, including the boundaries. The shaded triangular portion in Figure 3 is meaningless because the lower bound cannot be greater than the upper bound. Point values in the dimension are situated along the diagonal of this plane, where the lower bound equals the upper bound. Points along the upper edge of the plane are intervals that are bounded by the lower bound only, such as (3,∞). Similarly, points on the left edge of the plane denote intervals that have an upper bound only, such as (–∞,5). Since object regions and interest regions coexist in the subject space, points representing the intervals for object regions and interest regions in this dimension are shown in circles and crosses respectively.

5.3 Solving the Intersection Reporting Problem

Using this two-dimensional plane and the relative positions of the data points, it is easy to report the set of intervals that overlap with a given interval. Figure 4 shows a point representing an interval and the corresponding area on the plane in which all intervals represented by points in that area intersect with the given interval in this dimension.

In Figure 4a, the given interval is a point value. The rectangular shaded area indicates the values of all possible intervals that have lower bound values less than that of the given interval and upper bound values greater than that of the given interval. Hence, the given interval intersects with all intervals in the shaded area.

Figure 4b shows the situation where the given interval spans a range. Consider two intervals α and β. Let α have the range $[\alpha_L, \alpha_U]$, and β have a range $[\beta_L, \beta_U]$. α intersects β if one of the following four conditions is satisfied:

1. $\alpha_L \geq \beta_L$ and $\alpha_U \leq \beta_U$; or
2. $\alpha_L \leq \beta_L$ and $\alpha_U \geq \beta_U$; or
3. $\alpha_U \geq \beta_U$ and $\alpha_L \leq \beta_U$ and $\alpha_L \geq \beta_L$; or
4. $\alpha_L \geq \beta_L$ and $\alpha_U \leq \beta_L$ and $\alpha_U \geq \beta_U$.

The shaded area in Figure 4b is partitioned into four areas to reflect each of the four conditions above. Figure 4 has shown that given any interval in a dimension, all intervals overlapping the given interval can be found by gathering all points in the shaded area.

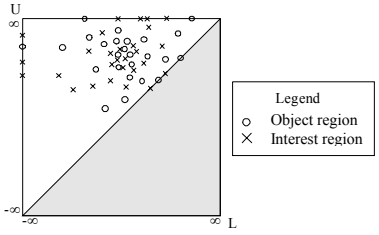

Figure 3 Object regions and interest regions indexed along a dimension

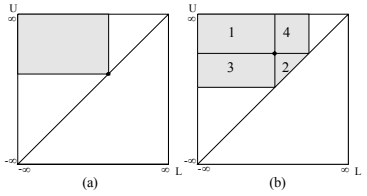

Figure 4 Area that contain intervals that overlaps with a given interval

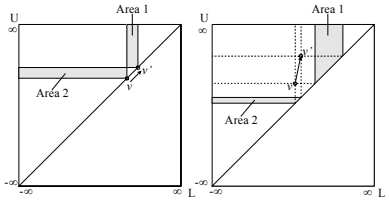

Figure 5 Two examples showing areas of state changes upon the update of a value

5.4 Indexing and Detecting State Changes

When a point value is updated, it moves on the plane as it is being updated. Figure 5 shows the value of an interval changes from v to v'. All intervals represented by points in Area 1 have *entered* the updated interval; all intervals represented by points in Area 2 have *left* the range of the updated interval in this dimension. All intervals in the unshaded area have no state transition.

5.5 Data Structures

For each dimension, the lower bound values and upper bound values are stored in two sorted indexes. Since the data domains under consideration are assumed to have total order, we may use any main memory data structures designed for storing and traversing sorted numbers.

Each dimension of a subject space uses two skip lists [16] for indexing lower and upper bound values of intervals respectively. The intervals are constraints of publications and subscriptions in this dimension. Each node of the skip list consists of two sets – IR and OR. IR and OR are the sets of interest regions and object regions that have constraints with end points of the value of the key. End points with values of $+\infty$ or $-\infty$ are not indexed.

The following is an example illustrating how to index object and interest regions. Consider a dimension d, object regions o_1, o_2, o_3, and interest regions i_1, i_2, i_3.

C_{o_1} ={d=[100,100]}, C_{o_2} = {d=(60,120))}, C_{o_3} = {d = [60,∞)},

C_{i_1} ={d=[100, ∞)}, C_{i_2} = {d = (–∞,100)}, C_{i_3} = {d = [60,100]}.

The data structure of dimension d consists of two lists, LB and UB, for indexing the lower bound endpoints and upper bound endpoints of the constraints respectively. Figure 6 illustrates the contents of the two lists.

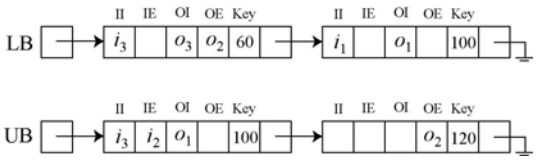

Figure 6 The content of a dimension index

5.6 Implementation of the Filter Process

Among the five algorithms presented in Section 6.4.2, the filter process can be divided into two groups – stateless filtering and state-persistent filtering. Stateless filtering finds all regions that match with a given region. No state information is used. The filter process of $Q_P(\mathcal{P})$, $Q_S(\mathcal{S})$ and $\overline{Q}_P^t(\mathcal{P})$ use stateless filtering. The update operations $Q_P^t(\mathcal{P})$ and $Q_S^t(\mathcal{S})$ use state-persistent filtering because they require state information to determine regions that enter the matching relation as a result of the update.

5.6.1 Stateless Filtering

$Q_P(\mathcal{P})$, $Q_S(\mathcal{S})$ and $\overline{Q}_P^t(\mathcal{P})$ can use the same filtering algorithm. Here, we use $Q_P(\mathcal{P})$ as an example to illustrate the process. Line 4 of algorithm $Q_P(\mathcal{P})$ in Figure 1 is

$$\mathcal{I} \leftarrow \mathcal{I} \cup \{I' \mid \forall i \in I', i \,\mathbf{m}\, o\}.$$

This line tries to find all interest regions that match with the given object region o, which can be valid in multiple subject spaces. Let the set of subject spaces in which o is valid be Σ_o. The object region o must then be compared with all interest regions in Σ_o. With the knowledge of the underlying index data structures, this operation can now be explained in more details by algorithm findAllMatchedInterestRegions in Figure 7.

Algorithm findAllMatchedInterestRegions(o)
Input: object region o
Output: matchSet – the set of interest regions overlapping the object region o
Local variable:
MismatchSet – the set of interest regions not overlapping with o
(1) MismatchSet = ∅
(2) $\Sigma_o = \{\sigma \mid o \subseteq V_\sigma\}$
(3) $I = \{I \mid \exists\, \sigma \in \Sigma_o,\ V_\sigma^i \subseteq V_\sigma\}$
(4) For each constraint c of o do
(5) Let d be the dimension of c
(6) mismatchSet ← mismatchSet ∪ collectInterestRegions(d, LB, c.upperBound, ∞)
(7) mismatchSet ← mismatchSet ∪ collectInterestRegions(d, UB, –∞, c.lowerBound)
(8) EndFor
(9) $I \leftarrow I \setminus$ mismatchSet

Figure 7 Algorithm findAllMatchedInterestRegions

Algorithm collectInterestRegion(d, *list*, *fromValue*, *toValue*)
Input: d – dimension

list – UB or LB
fromValue, *toValue* – 2 values on the list

Output: interestRegions – the set of interest regions with a constraint that has a bound within the range from *fromValue* to *toValue*

Local variables: curNode – a temporary pointer to a node in *list*

```
(1)  If fromValue = –∞ then
(2)     curNode ← first node in list
(3)  Else if fromValue = ∞ then
(4)     Return interestRegions = ∅
(5)  Else
(6)     curNode ← first node in list with key ≥ fromValue
(7)  EndIf
(8)  While curNode exists and curNode.key ≤ toValue do
(9)     If curNode.key = fromValue and fromValue is
           exclusive then
(10)       InterestRegions ← InterestRegions ∪ curNode.IE
(11)    Else If curNode.key = toValue and toValue is
           exclusive then
(12)       InterestRegions ← InterestRegions ∪ curNode.IE
(13)    Else
(14)       InterestRegions ← InterestRegions ∪ curNode.II
(15)       InterestRegions ← InterestRegions ∪ curNode.IE
(16)    EndIf
(17)    curNode ← the next node of curNode
(18) EndWhile
```

Figure 8 Algorithm collectInterestRegions

The set I in line 3 of algorithm findAllMatchedInterestRegions is the set of all interest regions that are valid in Σ_o, the subject spaces that contain object o.

Among the interest regions in I, some of them may have constraints that do not satisfy the constraints of object o. Using the indexes, lines 4 to 8 come up with the set of all interest regions that contain constraints not satisfied by o. Line 9 eliminates the set of interest regions that do not match with o from I. The resulting set $I = \{I' \mid \forall i \in I', i \mathbf{m} o\}$.

Lines 6 and 7 make calls to a procedure called collectInterestRegions. The algorithm of collectInterestRegions is shown in Figure 8. Recall that each dimension is represented by a two-dimensional plane and two skip lists are used to index the lower and upper bound values. Given a constraint of an object region, this procedure returns the constraints of interest regions that do not overlap with this constraint.

5.6.2 State-Persistent Filtering

The update operations of object regions and interest regions require state information to find the set of regions whose relationships with a given region have undergone state transitions. Here, we use the update of an object region as an example. The interest regions to be included by the filter process are those that did not overlap with the given object region before the update, and will overlap with the given object region after the update. The operation is executed in line 4 of the algorithm of Q_P^t given in Figure 2 as follows:

$\mathcal{I} \leftarrow \mathcal{I} \cup \{I' \mid \forall i \in I', i \mathbf{m} o'_{\text{new}} \wedge (i, o'_{\text{old}}) \notin \mathbf{m}\}$.

These interest regions must have a constraint specified in one of the dimensions that the object region is about to update. State-persistent filtering can be done using the main memory index alone. Algorithm findNewlyMatchedInterestRegions illustrates the process of solving the problem of finding the set of interest regions using the main memory index.

Algorithm findNewlyMatchedInterestRegions(o_{new}, o_{old})

Input: o_{new} – the updated state of object region
o_{old} – the state of the object region before update

Output: newMatch – the set of interest regions that overlap with o_{new} but not with o_{old}

Local variable: node – a pointer to a node on the skip list

```
(1)  newMatch = ∅
(2)  For each constraint c_new of o_new do
(3)     Let d be the dimension of c
(4)     Let c_old be the constraint of o_old in dimension d
(5)     If c_old.upperBound < c_new.upperBound then
(6)        newMatch ← newMatch ∪
              collectInterestRegions(d, LB, c_old.upperBound,
           c_new.upperBound)
(7)     EndIf
(8)     If c_old.lowerBound > c_new.lowerBound then
(9)        newMatch ← newMatch ∪
              collectInterestRegions (d, UB, c_new.lowerBound,
           c_low.lowerBound)
(10)    EndIf
(11) EndFor
(12) For each interest region i in newMatch
(13)    If i has fewer constraints than o then
(14)       For each constraint c_i of i do
(15)          Let c_o be the constraint of o in the same
                 dimension as c_i
(16)          If c_o exists and overlaps(c_i, c_o) = false then
(17)             newMatch ← NewMatch \ {i}
(18)          EndIf
(19)       EndFor
(20)    Else
(21)       For each constraint c_o of o do
(22)          Let c_i be the constraint of i in the same
                 dimension as c_o
(23)          If c_i exists and overlaps(c_i, c_o) = false then
(24)             newMatch ← NewMatch \ {i}
(25)          EndIf
(26)       EndFor
(27)    EndIf
(28) EndFor
```

Figure 9 Algorithm findNewlyMatchedInterestRegions

Algorithm overlap(c_1, c_2)
Input: c_1, c_2 – two constraints
Output: Boolean – *true* if c_1 overlaps c_2, *false* otherwise
Local variable:
(1) If c_1 has an upper bound $\neq \infty$ and c_2 has an lower bound $\neq -\infty$ then
(2) If upper bound of c_1 is inclusive and
lower bound of c_2 is inclusive and
lower bound of c_2 > upper bound of c1then
(3) Return false
(4) Else If lower bound of $c_2 \geq$ upper bound of c_1 then
(5) Return false
(6) EndIf
(7) EndIf
(8) If c_1 has an lower bound $\neq -\infty$ and c_2 has an upper bound $\neq \infty$ then
(9) If lower bound of c_1 is inclusive and
upper bound of c_2 is inclusive and
upper bound of c_2 < lower bound of c1then
(10) Return false
(11) Else If upper bound of $c_2 \leq$ lower bound of c_1 then
(12) Return false
(13) EndIf
(14) EndIf
(15) Return true

Figure 10 Algorithm overlap

Section 6.6.2 explained how to use the index to detect state transitions. Figure 5 showed the areas on the two-dimensional plane representing each dimension that undergoes states transitions upon an update on a constraint. Lines 2 to 11 of algorithm findNewlyMatchedInterestRegions are for collecting all interest regions that have a constraint represented as a point in Area 1 on the plane of its corresponding dimension. With the knowledge of the values before and after update, we scan the skip list between these two values to collect the pointers of these interest regions. This task is done by algorithm collectInterestRegions, which is explained in Figure 8. The interest regions in the set newMatch after the **for** loop from line 2 to 11 may contain interest regions that have one or more dimensions not overlapping with the given object region. Lines 12 to 28 of algorithm findNewlyMatchedInterestRegions are for removing these irrelevant regions from the set. We try to optimize the comparison by looping through the constraints of the region that contains less constraint. This decision is done in line 13 and chooses to execute the loop from line 14 to 19 or the loop from line 21 to 26. These two loops have the same task – to remove regions that have a dimension not overlapping with the given object region. The algorithm for testing whether two constraints are overlapping each other is shown in "Algorithm overlap" (Figure 10). It uses the relative positions of the two points representing the two constraints to make the decision.

6 Analytical Performance Evaluation

This section presents the time analysis of the algorithms presented in the previous section. We will also compare the time complexity of these algorithms with existing matching algorithms for publish/subscribe systems.

6.1 Time Analysis

Section 4 presented the general approach to executing all operations required in the subject space model. From the algorithms shown in Figures 1 and 2, we can do a generic complexity analysis for these algorithms, which is independent of matching semantics and the underlying data structures. The indexing algorithms presented in Section 5.6 gave more detailed information on the behaviors of operations of the subject space model specific to the proposed indexing algorithm, and allow a refined complexity analysis on these operations.

Table 1 summarizes the time complexity of the five operations in the subject space model.

Table 1 Time complexities of operations for the subject space model

- $|\mathcal{O}|$ is the number of object regions of a given publication
- $|\mathcal{I}|$ is the number of interest regions of a given subscription
- $\mathcal{O}$ and $\mathcal{I}$ are the number of object regions and interest regions respectively that have undergone state transitions
- $|S_c|$ is the number of candidate subscriptions
- $|P_c|$ is the number of candidate publications
- C_x is the complexity of the algorithm indicated by the subscript x
- $|\overline{\mathcal{I}}|$ is the average number of interest regions of subscriptions in S_c
- $|\overline{\mathcal{O}}|$ is the average number of object regions of publications in P_c
- $|c_o|$ is the average number of constraints per object region in $\mathcal{O}$
- publication $|c_i|$ is the average number of constraints per interest region in $\mathcal{I}$

Operation	Time Complexity
$Q_P(\mathcal{P})$	$O(\|\mathcal{O}\| \cdot C_m + \|S_c\| \cdot C_{\mathcal{M}})$ $= O(\|\mathcal{O}\| \cdot \|c_o\| \cdot C_{\text{collectInterestRegions}} + \|S_c\| \cdot (\|\mathcal{O}\| + \|\overline{\mathcal{I}}\|))$
$Q_S(\mathcal{S})$	$O(\|\mathcal{I}\| \cdot C_m + \|P_c\| \cdot C_{\mathcal{M}})$

	$= O(\lvert\mathcal{I}\rvert \cdot \lvert c_i\rvert \cdot \mathrm{C}_{\text{collectObjectRegions}} + \lvert P_c\rvert \cdot (\lvert\mathcal{I}\rvert + \lvert\overline{\mathcal{O}}\rvert))$
$\mathcal{Q}_P^t(\mathcal{P})$	$O(\lvert\hat{\mathcal{O}}\rvert \cdot \mathrm{C}_{\mathbf{m}}^{\mathrm{t}} + \lvert S_c\rvert \cdot \mathrm{C}_{\mathcal{M}})$ $= O(\lvert\hat{\mathcal{O}}\rvert \cdot (\lvert c_o\rvert \cdot \mathrm{C}_{\text{collectInterestRegions}} + \lvert c_o\rvert \cdot \mathrm{C}_{\text{overlap}}) +$ $\lvert S_c\rvert \cdot (\lvert\mathcal{O}\rvert + \lvert\overline{\mathcal{I}}\rvert))$
$\mathcal{Q}_S^t(\mathcal{S})$	$O(\lvert\hat{\mathcal{I}}\rvert \cdot \mathrm{C}_{\mathbf{m}}^{\mathrm{t}} + \lvert P_c\rvert \cdot \mathrm{C}_{\mathcal{M}})$ $= O(\lvert\hat{\mathcal{I}}\rvert \cdot (\lvert c_i\rvert \cdot \mathrm{C}_{\text{collectObjectRegions}} + \lvert c_i\rvert \cdot \mathrm{C}_{\text{overlap}}) + \lvert P_c\rvert \cdot$ $(\lvert\mathcal{I}\rvert + \lvert\overline{\mathcal{O}}\rvert))$
$\overline{\mathcal{Q}}_P^t(\mathcal{P})$	$O(\lvert\mathcal{O}\rvert \cdot \mathrm{C}_{\mathbf{m}} + \lvert S_c\rvert \cdot \mathrm{C}_{\mathcal{M}})$ $= O(\lvert\mathcal{O}\rvert \cdot \lvert c_o\rvert \cdot \mathrm{C}_{\text{collectInterestRegions}} + \lvert S_c\rvert \cdot (\lvert\mathcal{O}\rvert + \lvert\overline{\mathcal{I}}\rvert))$

The two terms within the complexity expression reflect the two steps of the algorithm. $\mathrm{C}_{\mathbf{m}}$ is the complexity for finding all interest regions or object regions that match with a given object region or interest region, under the matching semantics **m**. $\mathrm{C}_{\mathbf{m}}^{\mathrm{t}}$ is the complexity for finding all object regions or interest regions that have undergone a state transition in its relationship with a given object region or interest region. Both $\mathrm{C}_{\mathbf{m}}$ and $\mathrm{C}_{\mathbf{m}}^{\mathrm{t}}$ are dependent on the indexing algorithm used for solving the problem. $\mathrm{C}_{\mathcal{M}}$ is the complexity for determining whether a pair of publication and subscription is in the matching relation $\mathcal{M}$. The dominant factors in the complexity of these operations are $\mathrm{C}_{\mathbf{m}}$, $\mathrm{C}_{\mathbf{m}}^{\mathrm{t}}$ and $\mathrm{C}_{\mathcal{M}}$.

A refined evaluation of $\mathrm{C}_{\mathbf{m}}$, $\mathrm{C}_{\mathbf{m}}^{\mathrm{t}}$ and $\mathrm{C}_{\mathcal{M}}$ can be obtained by considering the indexing algorithm used in the implementation of the prototype. In our implementation:

$$\begin{aligned}
\mathrm{C}_{\mathbf{m}} &= \mathrm{C}_{\text{findAllMatchedInterestRegions}} = O(\lvert c_o\rvert \cdot \mathrm{C}_{\text{collectInterestRegions}}) \\
&= \mathrm{C}_{\text{findAllMatchedObjectRegions}} = O(\lvert c_i\rvert \cdot \mathrm{C}_{\text{collectObjecttRegions}}) \\
\mathrm{C}_{\mathbf{m}}^{\mathrm{t}} &= \mathrm{C}_{\text{findNewlyMatchedInterestRegions}} \\
&= O(\lvert c_o\rvert \cdot \mathrm{C}_{\text{collectInterestRegions}} + \lvert c_o\rvert \cdot \mathrm{C}_{\text{overlap}}) \\
&= \mathrm{C}_{\text{findNewlyMatchedObjectRegions}} \\
&= O(\lvert c_i\rvert \cdot \mathrm{C}_{\text{collectObjecttRegions}} + \lvert c_i\rvert \cdot \mathrm{C}_{\text{overlap}})
\end{aligned}$$

The algorithms collectInterestRegions and collectObjectRegions involve scanning through certain areas on the two-dimensional plane representing each dimension and collecting the associated object regions or interest regions. $\mathrm{C}_{\text{collectInterestRegions}}$ and $\mathrm{C}_{\text{collectObjectRegions}}$ are therefore $O(n)$ where n is the number of nodes along the skip lists traversed in the process of collecting the regions. The complexity of the overlap algorithm (Figure 10) is $O(1)$. With the assumption that filters are conjunctive queries, $\mathrm{C}_{\mathcal{M}}$ is $O(n)$. Specifically, for operations on a publication, $\mathrm{C}_{\mathcal{M}} = O(\lvert\mathcal{O}\rvert + \lvert\overline{\mathcal{I}}\rvert)$, and for operations on a subscription, $\mathrm{C}_{\mathcal{M}} = O(\lvert\mathcal{I}\rvert + \lvert\overline{\mathcal{O}}\rvert)$.

From the above analysis, the complexity of each of the operations is dependent on several variables, and varies with each of them linearly. It can be concluded that the worst-case asymptotic complexity of all five operations is $O(n)$. With the assumption that the number of publications and subscriptions is much larger than the number of regions per publication or subscription and the number of constraints per region, the dominant factor in these complexity expressions is the number of the candidate publications or candidate subscriptions – $|S_c|$ and $|P_c|$. In the worst-case, the number of candidate subscriptions equals the number of all subscriptions in the system, or the number of candidate publications equals the number of all publications in the system.

The worst-case is unlikely to happen unless all subscriptions are extremely general and the publications can match all subscriptions. Let's now turn our attention to the *expected* behavior of these algorithms given more practical assumptions.

In an average case, we assume that the object regions and interest regions are evenly distributed in the subject spaces. Furthermore, we assume that:

1. an update operation only adjusts a subset of the regions of the publications or subscription,
2. an update operation only changes a subset of constraints of the regions updated, and
3. the updated value is in the proximity of the previous value.

With the above assumptions, we argue that the expected runtime of the two update operations $\mathcal{Q}_P^t(\mathcal{P})$ and $\mathcal{Q}_S^t(\mathcal{S})$ are less than that of the insertion and deletion operations $Q_{\mathrm{P}}(\mathcal{P})$, $Q_{\mathrm{S}}(\mathcal{S})$ and $\overline{\mathcal{Q}}_P^t(\mathcal{P})$. For example, if a publication P is being updated, let the states before and after the update be P_1 and P_2. The process of updating P from P_1 to P_2 generates less candidate subscriptions than the process of inserting a new publication with the state of P_2.

The first two assumptions show that an update operation needs to do less index lookup than

required by an insertion operation and will result fewer candidates for the refinement process. The third assumption means that a region only moves to a position in the subject space that is close to its original position in an update. Since regions are evenly distributed, the updated region will only "collide" with a small number of regions, compared to the potentially much larger number of regions that overlap with this region. In contrast, the goal of an insertion operation is to find the set of all regions that overlap with a given region. Hence, the number of candidates in the update operations is much lower than the number of candidates in the insertion operations. Since the number of candidates is the dominant factor affecting the runtime of the algorithm, the update operations are expected to run faster than insertion operations.

6.2 Comparison with Related Matching Algorithms

We proceed to compare the time complexity of the proposed matching algorithm with existing algorithms, and show the advantages of the subject space model and the proposed indexing algorithm. The time complexities of some existing matching algorithms are summarized in table 2. Worst-case complexity is assumed unless stated otherwise.

Table 2 Time complexity of existing matching algorithms

- |S| is the number of subscriptions *s*
- |A| is the number of attributes. An attribute is equivalent to a dimension in the subject space model.
- |P| is the number of predicates *p*. A predicate is approximately equivalent to a node in a skip list in the dimension indexes for the subject space model.

Algorithm	Time Cost
Naïve [4]	$\mathbf{\lvert S\rvert}\cdot\lvert\overline{P}\rvert$
Counting [9]	$\lvert P_{sat}\rvert\cdot\mathbf{\lvert S\rvert}$
Counting algorithm with equality-preferred approach [4]	$\lvert P_{neq\ sat}\rvert\cdot(\mathbf{\lvert S_{partial\ sat}\rvert + \lvert S_{neq}\rvert})$
Cluster propagation [3]	$K_r \cdot \lvert\mathcal{H}\rvert + \Sigma_{H\in\mathcal{H}}\mu(H)(C_h + K_h \cdot\lvert H.A\rvert) + \mathbf{\Sigma_{s\in S}}\ \nu(C(s).p)\cdot$ checking$(C(s).p, \mathbf{s})$
Gough [10]	$\lvert A\rvert \cdot \mathbf{\mathit{log}(\lvert S\rvert)}$
Gryphon [12]	Worst-case: $\mathbf{\lvert S\rvert}\cdot(\lvert A\rvert + 1)$
	Average case: $2(\lvert A\rvert+1)\cdot \mathbf{\lvert S\rvert^{1-\lambda}}$ $(\mathit{ln}V + \mathit{ln}\lvert A\rvert)$

In the table above, the factors of time cost contributed by the number of subscriptions are printed in bold font. It can be observed that all algorithms in this table, except the counting algorithm with equality-preferred approach, vary with some function of the *total* number of subscriptions in the system. In other words, in a system with a large number of subscriptions, the size of the index structure will be larger, and consequently requires a longer time to process the matching algorithm. The performance of the algorithms degrades in some function of the volume of data in the system.

The time complexity of the matching algorithm proposed in this research, however, is not dependent on the total number of subscriptions or publications in the system, but varies linearly with the number of *candidate* publications or subscriptions. This observation is significant because the number of candidates resulting from the filtering process is independent of the total number of publications or subscriptions in the system, but is sensitive to the *usage behavior* of the system. Usage behavior refers to the specifications of publications and subscriptions and the interactions between them. For example, if a new publication does not match with any of the subscriptions in the system, the time for processing this insertion operation in a system with millions of subscriptions will be comparable to the time for processing this insertion operation in a system with only hundreds of subscriptions, as opposed to having orders of magnitude of difference.

The counting algorithm with equality-preferred approach also has this property because its execution time is dependent on the number of subscriptions whose equality predicates are satisfied, which is a subset of the total number of subscriptions in the system. As a result, it requires fewer predicate evaluation operations than the counting algorithm. The algorithms proposed in this research, however, do not make assumptions on certain types of predicates. Indeed, the semantics allowed by the matching algorithms for subject spaces are more general than all of the matching algorithms in table 2 because they allow both publications and subscriptions to specify range values, as opposed to point values only.

7 Experiments

This section presents the experimental performance results of the indexing technique presented in Section 5. It serves to provide empirical performance data of the algorithms and to confirm the behaviors predicted by the analytical evaluation in the previous section.

The performance tests were run on a dual-CPU Pentium III workstation, with 900MHz i686 CPUs and 1.5 GB RAM operating under Linux. The algorithms are implemented in Java. JVM version 1.4 was used.

7.1.1 Time Taken to Insert Publications

Operations for inserting publications and subscriptions are symmetrical, and should take about the same time. Also, the operation for deleting publications is also expected to take the same time as publication insertion. This experiment shows the time taken to insert a publication into a subject space given the number of existing publications and subscriptions.

In this experiment, publications are inserted into a subject space with 50 dimensions. Both the number of publications and subscriptions are varied, and time is sampled at different orders of magnitude of both variables. Each publication has one region of 20 constraints and each subscription has one region of 2 constraints. Figure 11 shows the result of this experiment.

Different application domains may operate in different regions of the graph. For example, a network monitoring system may have publications in the order of 100,000 and relatively few subscriptions, in the order of 100. This system is operating near the left end of the top curve of the graph.

Consider the curve for 100,000 publications. It takes 172 milliseconds to add a publication into the subject space if there are only 10 subscriptions in the system, and it takes 776 milliseconds to add a publication into the subject space if there are 100,000 subscriptions in the system. Note that although the number of subscriptions in the system has increased by 4 orders of magnitude, insertion time only increased by 4.5 times. This result confirms the prediction of the analytic performance evaluation, which claims that execution times of operations do not vary linearly with the total number of subscriptions in the system, but are dependent on the usage behavior of the system.

7.1.2 Time Taken to Update Publications

This section presents the performance results on update operations. This experiment uses a subject space of 50 dimensions. Each publication has 1 region of 25 constraints and each subscription has 1 region of 2 constraints. The numbers of publications and subscriptions are varied, and the time for updating a publication is sampled at different orders of magnitude of both variables. In each update, the value of all 25 constraints of the object region of the publication is changed. Figure 12 shows the scenario in which the updated value is chosen within the range from 0 to 99. Figure 13 shows the scenario in which the updated value is either one higher or lower than the original value. One may notice that it takes much less time to update values incrementally than to update the value randomly. Applications with data that exhibit a locality of reference can take advantage of this feature of subject space model.

The analytical evaluation also showed that the matching time should vary linearly with the

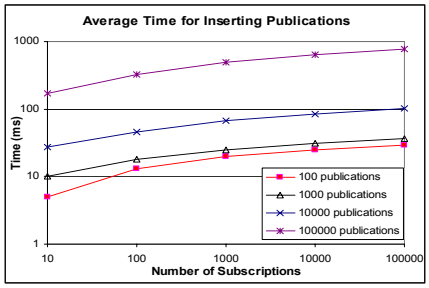

Figure 11 Inserting Publications

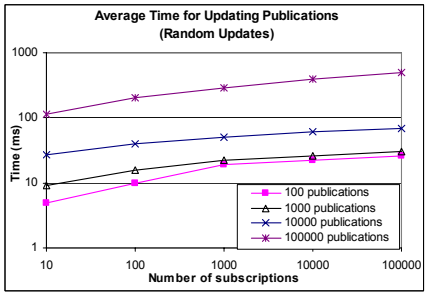

Figure 12 Updating Publications (Random)

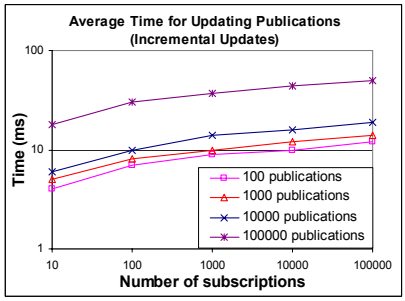

Figure 13 Updating Publications (Incremental)

number of candidate publications or subscriptions. The number of candidates, however, is not observable externally. In order to verify this claim, we set up an experiment to plot the time required for an update against the number of matches as a result of the update (Figure 14). For a good filtering algorithm, the number of candidates varies monotonically with the number of matches. This experiment shows that the update time is linearly dependent on the number of matches, which is a result of the usage behavior of the system.

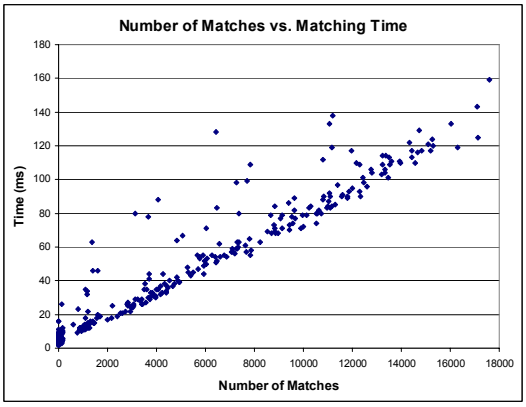

Figure 14 Matching time varies directly with the number of matches

8 Conclusion

This paper shows a solution for the matching problem of state-persistent systems. The proposed algorithm is scalable and has a complexity that varies with the usage behavior of the system as opposed to the amount of data in the system. It also allows better expressiveness than existing algorithms by its symmetrical treatment of publications and subscriptions and allows values to be expressed as interval of values.

The algorithm has also demonstrated the advantages of state-persistent publish/subscribe systems. The ability to do data updates is an advantage over stateless systems.

Trademark Statement

About the Authors

Hubert Ka Yau Leung is a software developer at IBM Toronto Lab. He received his M.A.Sc. degree in the Department of Electrical and Computer Engineering at the University of Toronto and a B.A.Sc. degree from the University of Waterloo.

Hans-Arno Jacobsen is a Professor of Electrical and Computer Engineering and Computer Science at the University of Toronto. His principal areas of research include middleware systems, distributed systems, aspect-oriented programming, and data management.

Reference

[1] B. Seeger and H.-P. Kriegel. Techniques for design and implementation of spatial access methods. In *Proc. 14th Int. Conf. on VLDB*, pp. 360-371, 1998.

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

[3] F. Fabret, H-Arno Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, D. Shasha, Filtering Algorithms and Implementation for Very Fast Publish/subscribe Systems. *SIGMOD* 2001.

[4] F. Fabret, F. Llirbat, J. Pereira and D. Shasha. Efficient Matching for Content-based Publish/Subscribe Systems, Technical report, INRIA, 2000.

[5] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. of the 1998 Intl. Conf. On VLDB*, pages 500-511, August 1998.

[6] H. Leung. Subject Space: State-Persistent Model for Publish/Subscribe Systems, CASCON, 2002.

[7] IBM. *MQSeries Publish/Subscribe User's Guide*, version 1, release 0.6, GC34-5269-08. January 2002.

[8] Inc., TIBCO/Rendezvous Concepts, October 2000.

[9] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proc. of the Int. Conf. on COOPIS*, 2000.K. J. Gough and G. Smith. Efficient recognition of events in distributed systems. In Proceedings of ACSC-18, 1995.

[10] M. Happner, R. Burridge, and R. Sharma. Java Message Service. Sun Microsystem Inc., October 1998.

[11] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription sys-tem. *PODC '99*, 1999.

[12] Nicholas Carriero and David Gelernter. Linda In Context. *Communications of the ACM*, Volume 32 Number 4, April 1989.

[13] Object Management Group. Notification Service Specification, Version 1.0.1. August 2002.

[14] S. Babu and J. Widom. Continuous Queries over Data Streams. In *SIGMOD Record*, September 2001.

[15] William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6): 668-676, June 1990.