A Detailed Investigation of Memory Requirements for Publish/Subscribe Filtering Algorithms

Sven Bittner and Annika Hinze

University of Waikato, New Zealand
{s.bittner, a.hinze}@cs.waikato.ac.nz

Abstract. Various filtering algorithms for publish/subscribe systems have been proposed. One distinguishing characteristic is their internal representation of Boolean subscriptions: They either require conversions into DNFs (canonical approaches) or are directly exploited in event filtering (non-canonical approaches).
In this paper, we present a detailed analysis and comparison of the memory requirements of canonical and non-canonical filtering algorithms. This includes a theoretical analysis of space usages as well as a verification of our theoretical results by an evaluation of a practical implementation. This practical analysis also considers time (filter) efficiency, which is the other important quality measure of filtering algorithms. By correlating the results of space and time efficiency, we conclude when to use non-canonical and canonical approaches.

1 Introduction

Publish/subscribe (pub/sub) is a communication pattern targeting on the active notification of clients: Subscribers define Boolean subscriptions to specify their interests; publishers disseminate their information by the help of event messages containing attribute/value pairs. A pub/sub system is acting as broker; it filters all incoming event messages and notifies subscribers if their registered subscriptions are matching. An integral and essential part of pub/sub systems is this filtering process, i.e., the determination of all subscribers interested in an incoming event message (also referred to as primitive event filtering). Generally, filtering algorithms for pub/sub systems should fulfil two requirements [6]:

- Efficient event filtering (fast determination of interested subscribers)
- Scalable event filtering (supporting large numbers of subscriptions)

For efficiency, current pub/sub systems apply main memory filtering algorithms. Thus, we can directly deduce the scalability characteristics of the central components of these systems from their memory requirements [2, 3, 10]. This characteristic implies the need to economize the usage of memory resources.

We can distinguish between two classes of filtering approaches for pub/sub systems: (i) algorithms directly filtering on Boolean subscriptions [3, 4, 11] (referred to as non-canonical approaches in the following), and (ii) algorithms filtering on subscriptions in canonical forms [1, 5, 6, 8, 12] (referred to as canonical approaches). Internally, algorithms of Class (ii) either filter on disjunctive normal forms (DNF) [5] or only support conjunctive subscriptions [1, 6, 8, 12]. Thus, if supporting arbitrary Boolean subscriptions, these approaches always require conversions of subscriptions to DNFs. If algorithms only allow conjunctions, each disjunctive element of a DNF is treated as a separate subscription [9].

Canonical approaches store subscriptions plainly as canonical forms. Hence, if subscriptions merely utilize such forms, this class of algorithms allows for efficient event filtering. This results from the ability to neglect arbitrary Boolean expressions while filtering. However, due to the need of converting Boolean subscriptions to DNFs, subscriptions consume more space than required by their original forms [3]. Additionally, the matching process works over more (or, in case of supporting DNFs, larger) subscriptions. For non-canonical approaches holds the opposite: Subscriptions demand less memory for storage but involve a more sophisticated matching. Hence, the benefits and drawbacks (you can find a more detailed discussion in [3]) of both classes of filtering algorithms are twofold and necessitate a thorough analysis to allow solid statements about their advantages and disadvantages.

In this paper, we present a thorough analysis and evaluation of the memory requirements of canonical and non-canonical filtering algorithms. This includes a theoretical analysis as well as a practical investigation of space usages. Furthermore, we correlate the memory requirements of the analyzed algorithms to their filter efficiency (time efficiency). As representatives of canonical algorithms we analyzed the counting [1, 12] and the cluster approach [6, 8], which are known to be efficient and reasonably memory-friendly [3]. Non-canonical algorithms are represented by the filtering approach in [3] because of its time efficiency due to the utilization of indexes. Our decision to compare these particular algorithms is also driven by their similar exploitation of one-dimensional indexes for filtering. In detail our contributions in this paper are:

1. A characterization scheme for qualifying primitive subscriptions
2. A theoretical analysis and comparison of the memory requirements of canonical and non-canonical filtering algorithms
3. A practical verification of our theoretical results of memory usages
4. A correlation of memory usage and filter efficiency of filtering algorithms
5. Recommendations for the utilization of non-canonical and canonical algorithms

The rest of this paper is structured as follows: Section 2 gives an overview of the analyzed algorithms and presents related work. Our characterization scheme qualifying subscriptions can be found in Sect. 3 as well as our theoretical analysis of memory requirements. Section 4 includes a comparison of the theoretical memory usages and their graphical presentation. We practically verify our results in Sect. 5.1, followed by the correlation of memory usage to filtering efficiency in Sect. 5.2. Finally, we conclude and present our future work in Sect. 6.

2 Analyzed Algorithms and Related Work

In this section, we outline the three filtering algorithms used in our later analysis. Afterwards, we present related work that is evaluating and comparing different filtering approaches in Sect. 2.2.

2.1 Review of Analyzed Algorithms

We now give a brief overview of the algorithms analyzed in Sect. 3, namely the counting algorithm [1, 12], the cluster algorithm [6, 8], and the non-canonical algorithm [3]. We have chosen these algorithms for our analysis due to their time and space efficiency characteristics[1]. We restrict this subsection to a short review of the approaches and refer to the original works for thorough study and description of the algorithms.

Review of the Counting Algorithm. The counting algorithm was originally proposed in [12] for filtering on plain text in combination with secondary storage. Later, it was adopted as pure main memory filtering approach working on attribute/value pairs, e.g., [1]. It only supports conjunctive subscriptions and requires the conversion of subscriptions involving disjunctions into DNFs. Then, each element (i.e., a conjunction) participating in the one disjunction of a DNF is treated as separate subscription [9].

Filtering works in two steps: Firstly, matching predicates are determined by utilizing one-dimensional indexes (predicate matching). Secondly (subscription matching), all subscriptions involving these predicates are derived by exploiting a predicate subscription association table. Counters in hit vectors are increased for each matching predicate per subscription. Finally, hit and subscription predicate count vector are compared.

Review of the Cluster Algorithm. The cluster algorithm is described in detail in [6] and is based on the algorithm presented in [8]. Similar to the counting algorithm, only conjunctive subscriptions are supported by this approach. This requires a conversion to DNFs when supporting arbitrary Boolean subscriptions. Subscriptions are grouped into clusters according to their access predicates[2] and total number of predicates.

Again, event filtering works in two steps: In predicate matching all matching predicates are determined by the help of one-dimensional indexes. For all matching access predicates, clusters with potentially matching subscriptions can be found by utilizing a cluster vector. Then, the subscriptions inside these clusters are evaluated by testing if all their predicates have been fulfilled (subscription matching).

Review of the Non-Canonical Algorithm. The non-canonical algorithm is presented in [3]. It comprises no restriction to conjunctive subscriptions as the previous two approaches. Instead, it directly exploits the Boolean expressions used in subscriptions.

Also the non-canonical algorithm utilizes one-dimensional indexes to efficiently determine predicates matching incoming events. Subscriptions are encoded in subscription trees representing their Boolean structure and involved predicates. A minimum predicate count vector states the minimal number of fulfilled predicates required for each subscription to match (variation of [3]). A hit vector is used to accumulate the number of fulfilled predicates per subscription (also a variation from [3]).

Once more, this matching approach involves a two-step event filtering process beginning with the determination of matching predicates (predicate matching). Then, by

[1] Refer to [2] and Sect. 1 for a more detailed argumentation.

[2] Common predicates that have to be fulfilled by an event to lead to fulfilled subscriptions.

the help of a predicate subscription association table all potential candidate subscriptions are determined (subscription matching step). The work in [3] proposes to evaluate all candidate subscriptions. However, if using a minimum predicate count vector, only subscriptions with more than the minimum number of matching predicates (minimum number of fulfilled predicates required for matching $|p_{min}|$) have to be evaluated. An example is the following: If a subscription consists of three disjunctive elements that contain conjunctions of nine, five, and seven predicates it holds $|p_{min}| = 5$. The accumulation of matching predicates per subscription is obtained using a hit vector.

2.2 Previous Evaluations

There have already been some comparative evaluations of event filtering approaches for primitive events. However, nearly all of them merely target evaluations of time efficiency in specifically chosen settings. Additionally, a detailed theoretical analysis of memory requirements of filtering algorithms cannot be found so far. The results of current practical evaluations of space efficiency are too restricted to be generalizable.

In [1] several implementations of the counting algorithm have been evaluated, but there is no comparison of this approach to other filtering solutions. For the investigation of subscription matching, subscriptions consist of only one to five predicates over domains of only ten different values. Thus, we cannot generalize the results of [1] to more complex and sophisticated settings utilizing expressive Boolean subscription languages. Additionally, a satisfactory theoretical evaluation is missing.

The work in [6] compares implementations of counting and cluster algorithm. However, the assumptions are similarly restricted as in [1]: only five predicates are used per subscription, domains consist of thirty-five possible values. Furthermore, subscriptions mainly define equality predicates in [6]. Naturally, this leads to a well-performing cluster algorithm, which is specifically designed to exploit this characteristic. Hence, the results of [6] do not present general settings and are mainly targeting filter efficiency.

In [3] the counting and the non-canonical approach are compared briefly. Thus, this analysis allows only limited conclusions about the behaviors of these algorithms. Again, a theoretical analysis of memory requirements is missing.

3 Theoretical Characterization and Analysis of Memory Usage

In this section we firstly present our characterization scheme allowing for a general representation of Boolean subscriptions. Our theoretical analysis of memory requirements based on our characterization can then be found in Sect. 3.2.

3.1 Characterization of Boolean Subscriptions

We now present our approach of characterizing subscriptions (and their management in algorithms). Since we target an evaluation of memory requirements, our methodology is based on attributes affecting the memory usage for storing subscriptions for efficient event filtering. Our approach also allows for a successful representation of the space requirements of the three filtering algorithms presented in Sect. 2.1.

Table 1. Overview of parameters characterizing subscriptions (Class S – subscription-related, Class A – algorithm-related, Class C – conversion-related, Class E – subscription-event-related)

Symbol	Parameter Name (Calculation)	Class
$\|p\|$	Number of predicates per subscription	S
$\|op\|$	Number of Boolean operators per subscription	S
op^r	Relative number of Boolean operators per subscription ($op^r = \frac{\|op\|}{\|p\|}$)	S
$\|s\|$	Number of subscriptions	S
$\|p_u\|$	Number of unique predicates	S
r_p	Predicate redundancy ($r_p = 1.0 - \frac{\|p_u\|}{\|p\|\|s\|}$)	S
$w(s)$	Width of subscription identifiers	A
$w(p)$	Width of predicate identifiers	A
$w(l)$	Width of subscription locations	A
$w(c)$	Width of cluster references	A
S_s	Number of disjunctively combined elements after conversion	C
s_p	Number of conjunctive elements per predicate after conversion	C
s^r	Relative no. of conjunctive elements per predicate after conversion ($s^r = \frac{s_p}{S_s}$)	C
p_e	Number of fulfilled predicates per event	E

We have identified 14 parameters, which are compactly shown in Table 1. The parameters of Class S allow for both a representation of the characteristics of subscriptions and a determination of their memory requirements for index structures. These six parameters directly describe subscriptions in their quantity $|s|$ and their average number of predicates $|p|$ and operators $|op|$. Parameter op^r expresses the number of operators relatively to the number of predicates. To determine predicate redundancy r_p, we also require the number of unique predicates registered with the system $|p_u|$.

Class A of parameters explicitly deals with filtering algorithm-related characteristics influencing the internal storage of subscriptions. The behavior of canonical approaches is expressed by the three parameters of Class C. The number of disjunctively combined elements in a converted DNF (which have to be treated as separate subscriptions [9]) is described by S_s. The average number of such elements containing a predicate from an original subscription s_p also strongly influences the behavior of canonical approaches. We can combine these two parameters to the relative number of conjunctive elements per predicate s^r. The parameter p_e of Class E incorporates the relation between subscriptions and events, which influences both space and time efficiency of filtering algorithms. For a detailed description of these parameters, we refer to [2].

Altogether, these fourteen parameters allow to characterize subscriptions, to derive the major memory requirements of filtering algorithms, and to describe the relation between events and subscriptions affecting the time efficiency of event filtering.

3.2 Theoretical Analysis of Memory Requirements

After presenting the parameters required to analyze the memory usage of filtering algorithms, we now continue with our theoretical analyzes. Note that our theoretical observations do not take into account implementation issues and other practical considerations. Our results are a base line helping to find a suitable filtering algorithm. An actual comparison of the theoretical memory requirements can be found in Sect. 4 as well as considerations for practical implementations.

Theoretical Memory Analysis of the Counting Algorithm. We now analyze the memory requirements of the counting algorithm [1, 12] in respect to the characterizing parameters defined in Sect. 3.1. According to [1] and our review in Sect. 2.1, the counting algorithm requires a fulfilled predicate vector, a hit vector, a subscription predicate count vector, and a predicate subscription association table. To efficiently support unsubscriptions, we also necessitate a subscription predicate association table. In the following we describe these data structures and derive their minimal memory requirements. We start our observations for cases with no predicate redundancy ($r_p = 0.0$). Subsequently, we extend our analysis to more general settings involving predicate redundancy.

Fulfilled predicate vector: The fulfilled predicate vector is required to store matching predicates in the predicate matching step. In an implementation, we might apply an ordinary vector ($p_e w(p)$ bytes) or a bit vector implementation ($\frac{|p||s|}{8}$ bytes) depending on the proportion of matching predicates.
In cases of high predicate redundancy there is only a small number of unique predicates. Thus, a bit vector implementation might require less memory compared to an ordinary vector implementation. However, if the fraction of fulfilled predicates per event p_e and totally registered predicates ($|p||s|$ predicates in total) is quite small, utilizing an ordinary vector might be advantageous.

Hit vector: The hit vector accumulates the number of fulfilled predicates per subscription. For simplicity, we assume a maximum number of 255 predicates per subscription (we can easily relax this assumption). Thus, each entry in the hit vector requires 1 byte. Altogether, for $|s|$ subscriptions creating S_s disjunctively combined elements due the canonical conversion, the space requirements are $|s|S_s$ bytes for the hit vector. Since this vector consists of one entry per subscription, its memory usage is independent of predicate redundancy r_p.

Subscription predicate count vector: We also require to store the total number of predicates each subscription consists of. According to our assumption for the hit vector, each subscription can be represented by a 1-byte entry. Thus, we require $|s|S_s$ bytes in total due to the applied canonical conversions (cp. hit vector).
Similar to the hit vector, the subscription predicate count vector does not depend on predicate redundancy r_p (it consists of entries per subscription).

Predicate subscription association table: This table has to be applied to efficiently find all subscriptions a predicate belongs to. In an implementation, each predicate has to be mapped to a list of subscriptions due to the required canonical conversions. This also holds in cases of no predicate redundancy ($r_p = 0$). Least memory is demanded if predicate identifiers might be used as indexes in this table (this requires consecutive predicate identifiers). For storing the list of subscriptions we have to store the corresponding number of subscription identifiers at a minimum (neglecting additional implementation overhead, such as the length of each list). Thus, altogether we have to record the list of subscription identifiers (requiring $w(s)s_p$ bytes per predicate) for all registered predicates ($|p||s|$ predicates in total), which requires $w(s)s_p|p||s|$ bytes in total.

If considering predicate redundancy r_p, for unique predicates (including one of each redundant predicate) the following amount of memory is required in bytes: $(1.0 - r_p)w(s)s_p|p||s|$. Redundant predicates use $r_p w(s)s_p|p||s|$ bytes. Thus, r_p does not influence the size of the predicate subscription association table.

Subscription predicate association table: The previously described data structures are required to support an efficient event filtering. However, unsubscription are supported very inefficiently. This is due to missing associations between subscriptions and predicates [1].

Least memory for subscription predicate associations is utilized when using a subscription identifier as index in a subscription predicate association table. Each entry maps this identifier to a list of predicate identifiers (there is also some implementation overhead as described for the previous table). Thus, we have to store a list of predicates for each subscription ($|s|S_s$ subscriptions in total due to conversions). Each list has to hold $|p|\frac{s_p}{S_s}$ predicate identifiers, which leads to $w(p)|s|S_s|p|\frac{s_p}{S_s}$ $=w(p)|s||p|s_p$ bytes in total for a subscription predicate association table.

Predicate redundancy r_p does not influence this table because it contains entries for each subscription. Thus, redundant predicates do not allow for the storage of less associations between subscriptions and predicates.

When accumulating the former memory usages, we require the following amount of memory in bytes (we exclude the fulfilled predicate vector since it is utilized by all three analyzed algorithms)

$$\mathrm{mem}_{counting} = |s|(2S_s + w(s)s_p|p| + w(p)s_p|p|) \ . \tag{1}$$

This observation holds in all cases of predicate redundancy r_p as shown above.

Theoretical Memory Analysis of the Cluster Algorithm. This section presents an evaluation of the memory requirements of the cluster algorithm [6, 8] according to the characterizing parameters defined in Sect. 3.1. However, this algorithm has several restrictions (e.g., usage of highly redundant equality predicates) and strongly depends on the subscriptions actually registered with the pub/sub system. Thus, we are not able to express all memory requirements of this algorithm based on our characterization scheme. In our following analysis we neglect the space usage of some data structures (cluster vector, references to cluster vector) and focus on the most space consuming ones, which leads to an increased amount of required memory in practice.

To efficiently support unsubscriptions, [6] suggests to utilize a subscription cluster table to determine the cluster each subscription is stored in. In our opinion, this data structure is not sufficient for a fast removal of subscriptions: The subscription cluster table allows for the fast determination of the cluster a subscription is stored in. Thus, we are able to remove subscriptions from clusters. It remains to determine when predicates might be removed from index structures due to the inherent assumption of predicate redundancy in [6][3]. Also the necessity of canonical conversions leads to shared predicates. Thus, to allow for a deletion of predicates in index structures, we require an association between predicates and subscriptions utilizing these predicates, e.g, by the application of a predicate subscription association table or by storing these associations inside index structures themselves.

The memory requirements of the cluster algorithm are as follows. Again, we firstly derive the space usage of the algorithm in case of no predicate redundancy ($r_p = 0.0$). Secondly, we generalize our results to cases involving predicate redundancy.

Predicate bit vector: This vector is similar to the fulfilled predicate vector applied in the counting algorithm. However, we require a bit vector implementation (as stated in [6]) due to the requirement of accessing the state of predicates (fulfilled or not fulfilled) directly. Thus, we demand $\frac{|p||s|}{8}$ bytes for the predicate bit vector. High predicate redundancy does not influence these memory requirements.

Clusters: Subscriptions themselves are stored in clusters according to both their access predicates and their total number of predicates. Clusters consist of a subscription line storing an identifier for each subscription ($w(s)$ bytes required per subscription). Furthermore, they contain a predicate array holding the predicates each subscription consists of (on average $\frac{s_p}{S_s}|p|w(p)$ bytes per subscription if only storing predicate identifiers). Clusters storing subscriptions with the same number of predicates and access predicates are linked together in a list structure. However, we neglect the memory requirements for this implementation-specific attribute.
Altogether, clusters require $|s|S_s(w(s) + \frac{s_p}{S_s}|p|w(p))$ bytes to store $|s|S_s$ subscriptions. Predicate redundancy does not influence the size of clusters. This results from the observation that clusters store predicates for all subscriptions. This storage happens in all cases of r_p and does not vary according to the uniqueness of predicates.

Subscription cluster table: This table is an additional data structure required to support efficient unsubscriptions (see argumentation above). It allows for the determination of the cluster each subscription is stored in. Utilizing subscription identifiers as indexes for the subscription cluster table, we require $|s|S_s w(c)$ bytes for the storage of $|s|S_s$ cluster references. Also this table is focussed on a mapping of subscriptions. Thus, its size is independent of predicate redundancy r_p.

Predicate subscription association table: As shown above, an association between predicates and subscriptions is required to allow for an efficient support of unsubscriptions. This information could be stored in a separate predicate subscription association table or as part of indexes themselves. Both options require the same amount of additional memory. If using predicate identifiers as indexes (or storing associations inside indexes), we require $w(s)s_p|p||s|$ bytes for these associations of $|p||s|$ predicates. Each predicate is contained in $w(s)s_p$ subscriptions on average.
Similar to our observation for the counting algorithm, predicate redundancy does not influence the size of predicate subscription associations.

[3] The motivation for [6] is the existence of shared predicates (predicate redundancy) because the clustering of subscriptions is obtained via access predicates, i.e., predicates need to be shared.

Accumulating the memory requirements of the formerly mentioned data structures (excluding the predicate bit vector) leads to the following number of bytes

$$\text{mem}_{cluster} = |s|(S_s w(s) + s_p|p|w(p) + S_s w(c) + s_p|p|w(s)) \ . \quad (2)$$

Again, our observation represents the memory requirements of the cluster algorithm regardless of predicate redundancy r_p as shown before.

Theoretical Memory Analysis of the Non-Canonical Algorithm. As last algorithm for our analysis, we have chosen a variant of the non-canonical approach [3] as presented in Sect. 2. According to [3], inner nodes of subscription trees store Boolean operators and leaf nodes store predicate identifiers. Each leaf node requires $w(p)$ bytes to store its predicate identifier and 1 byte to denote itself as a leaf node. For inner nodes, we store the Boolean operator in 1 byte and use 1 byte to denote the number of children (this implies that at least 255 predicates are supported per subscription as in the other algorithms presented before). In contrast to [3], we do not store the width of the children of inner nodes in bytes. Hence, to access the last out of n children, we have to compute the widths of all $n - 1$ previously stored children.

The non-canonical approach inherently supports efficient unsubscriptions due to its characteristic to store associations between subscriptions and predicates and vice versa. In the following, we analyze the memory requirements of the non-canonical approach beginning with the case of no predicate redundancy ($r_p = 0.0$). Afterwards, our analysis is extended to general settings involving predicate redundancy $r_p > 0$.

Fulfilled predicate vector: This vector serves the same purpose as its counterpart in the counting algorithm. Therefore, it requires the same amount of memory according to its realization and depending on r_p ($\min(\frac{|p||s|}{8}, p_e w(p))$ bytes).

Subscription trees: The encoding of subscription trees has been presented above. For predicates stored in leaf nodes, we require $(w(p) + 1)|p|$ bytes per subscription. Inner nodes demand $2|op|$ bytes of memory for each subscription. Thus, for all registered subscriptions, we need $|s|((w(p) + 1)|p| + 2|op|)$ bytes. Subscription trees have to store operators and predicate identifiers in all cases. Thus, they do not depend on predicate redundancy r_p.

Subscription location table: This table is applied to associate subscription identifiers and subscription trees. If utilizing subscription identifiers as indexes in this table (consecutive identifiers necessitated), we require $w(l)|s|$ bytes. Since the subscription location table stores entries per subscription, its memory usage is not influenced by predicate redundancy r_p.

Predicate subscription association table: The predicate subscription association table requires less memory than its counterparts in the previously analyzed algorithms. This is implied by the fact that subscriptions do not need a conversion in canonical forms. Thus, predicates are involved in less subscriptions (only one subscription in case of $r_p = 0.0$). Altogether, we require $|s||p|w(s)$ bytes for the predicate subscription association table.

Similar to the counterparts of this table in the two other algorithms, the memory usage of the predicate subscription association table is independent of predicate redundancy r_p.

Hit vector: Similar to the hit vector in the counting approach, this vector accumulates the number of fulfilled predicates per subscription. The hit vector requires $|s|$ bytes of memory since no conversions to canonical expressions are required by the non-canonical approach and according to the common assumption of a maximum of 255 predicates per subscription.

Minimum predicate count vector: This vector stores the minimum number of predicates per subscription that are required to be fulfilled $|p_{min}|$ in order to lead to a fulfilled subscription. According to our assumption of a maximum of 255 predicates per subscription, the minimum predicate count vector requires $|s|$ bytes of memory.

The required data structures (excluding the fulfilled predicate vector) sum up to the following amount of memory in bytes

$$\text{mem}_{non-canonical} = |s|(w(p)|p| + |p| + 2|op| + w(l) + |p|w(s) + 2) \ . \qquad (3)$$

Analogous to the previously described algorithms, the theoretical memory usage of the non-canonical approach does not depend on r_p as shown in our analysis.

4 Comparison of Theoretical Memory Requirements

After our analysis of three filtering algorithms and the derivation of their theoretical memory requirements, we now compare the memory usage of the two canonical approaches (counting and cluster algorithm) to the non-canonical algorithm. From this analysis we can deduce under which circumstances a non-canonical approach should be preferred (in respect to memory usage and thus scalability) and which settings favor canonical filtering algorithms.

In our following analysis, we focus on differing data structures of algorithms, i.e., we neglect the fulfilled predicate/predicate bit vector, which is incidentally required by all three algorithms. Thus, we directly compare (1) to (3).

All memory requirements derived in the last section grow linearly with increasing numbers of subscriptions. Moreover, all of them cut the ordinate in zero. Hence, for a comparison we solely need to analyze the first derivations of (1) to (3) in $|s|$

$$\text{mem}'_{counting}(|s|) = 2S_s + w(s)s_p|p| + w(p)s_p|p| \ . \qquad (4)$$

$$\text{mem}'_{cluster}(|s|) = S_s w(s) + s_p|p|w(p) + S_s w(c) + s_p|p|w(s) \ . \qquad (5)$$

$$\text{mem}'_{non-canonical}(|s|) = w(p)|p| + |p| + 2|op| + w(l) + |p|w(s) + 2 \ . \qquad (6)$$

To eliminate some parameters, let us assume fixed values for parameters of Class A: $w(s) = 4$, $w(p) = 4$, $w(l) = 4$, and $w(c) = 4$, i.e., the widths of subscription identifiers, predicate identifiers, subscription locations and cluster references are 4 bytes each. Furthermore, let us reduce the number of characterizing parameters specifying fixed values by utilizing the relative notions of op^r and s^r as introduced in Sect. 3.1.

We now compare the memory requirements (using the gradients) of the canonical algorithms (Equations (4) and (5)) to the memory requirements of the non-canonical approach (Equation (6)). We use the following notation to denote the canonical algorithm compared to the non-canonical approach: $S_s(\frac{algorithm}{non-canonical})$.

The inequalities shown in the following denote the point when the non-canonical approach requires less memory for its event filtering data structures than the respective canonical solution. These points are described in terms of the characterizing parameter S_s, i.e., if more than the stated number of disjunctively combined elements is created by the canonical conversion to DNF, the non-canonical approach requires less memory.

$$S_s(\frac{counting}{non-canonical}) > \frac{|p|(2op^r + 9) + 6}{2 + 8s^r|p|} \quad . \tag{7}$$

$$S_s(\frac{cluster}{non-canonical}) > \frac{|p|(2op^r + 9) + 6}{8 + 8s^r|p|} \quad . \tag{8}$$

In the following subsection we illustrate these observations graphically.

4.1 Graphical Illustration of Interchanging Memory Requirements

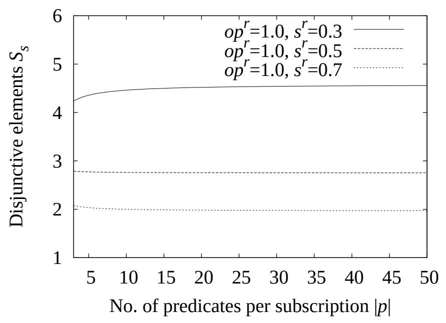
After the determination if the inequalities denoting the point when a non-canonical approach requires less memory than canonical algorithms (Equations (7) and (8)), we now present this turning point graphically.

Figure 1 shows the point of interchanging memory requirements for the counting algorithm and the cluster algorithm. We have chosen $op^r = 1.0$ in Fig 1(a) and Fig. 1(b); the parameter s^r is varied from 0.3 to 0.7. The abscissae of both figures show the number of predicates per subscription $|p|$, the ordinates are labeled with the number of disjunctively combined elements per subscription after conversion S_s. Both graphs denote which number of disjunctively combined elements have to be created by canonical conversions to DNF to favor the non-canonical approach in respect to memory requirements (cf. (7), (8)).

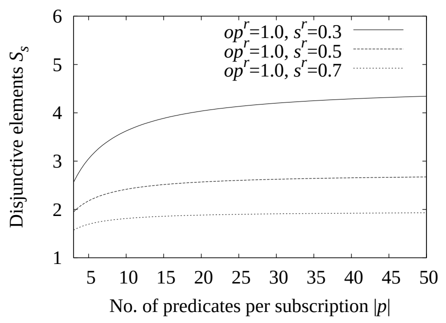
We can realize that the counting algorithm requires less memory in cases of small predicate numbers $|p|$ than the cluster algorithm. However, with increasing predicate numbers $|p|$ both algorithms behave nearly the same, i.e., for 50 or more predicates per subscription, it holds $S_s \approx 2.0$ (counting) and $S_s < 2.0$ (cluster) in case of $s^r = 0.7$. Thus, even if DNFs only consist of approximately 2 disjunctively combined elements, a non-canonical approach requires less memory. Smaller values of s^r favor the counting and the cluster algorithm. This is due to the fact of requiring less associations between predicates and subscriptions in these cases.

In Fig. 1(a) and Fig. 1(b), we have chosen $op^r = 1.0$, which describes the worst case scenario of the non-canonical algorithm. In practice, it always holds $op^r < 1.0$, since each inner node of a subscription tree has at least two children. Hence, a subscription tree containing $|p|$ leaf nodes (i.e., predicates) consists of a maximum of $|p| - 1$ inner nodes (i.e., operators). This implies $op^r \leq \frac{|p|-1}{|p|} < 1.0$. In practice, we have to expect much smaller values than 1.0 for op^r, because in subscription trees consecutive binary operators can be subsumed to n-ary ones.

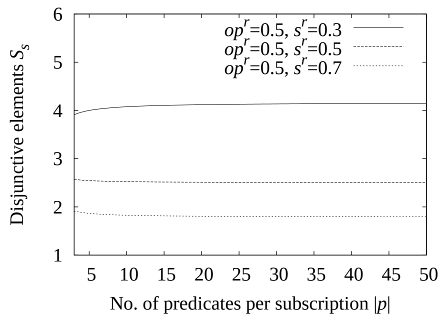
These observations for the characterizing parameter op^r lead to further improved memory characteristics of the non-canonical approach. Figure 1(c) shows this behavior using $op^r = 0.5$ for the counting approach; the cluster algorithms is presented in Fig. 1(d). Thus, even if subscriptions use only one disjunction, a non-canonical approach shows less memory usage and better scalability than the counting algorithm ($s^r = 0.7$).



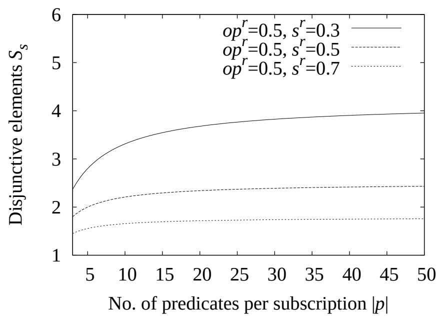
(a) Counting vs. non-canonical, $op^r = 1.0$



(b) Cluster vs. non-canonical, $op^r = 1.0$



(c) Counting vs. non-canonical, $op^r = 0.5$



(d) Cluster vs. non-canonical, $op^r = 0.5$

Fig. 1. Theoretically required number of disjunctively combined elements S_s to achieve less memory usage in the non-canonical approach compared to the counting and cluster algorithm using $op^r = 1.0$ (Fig. 1(a) and Fig. 1(b)) and $op^r = 0.5$ (Fig. 1(c) and Fig. 1(d))

4.2 Considerations in Practice

Our previous analysis shows a comparison of the theoretical memory requirements of three algorithms. However, a practical implementation requires additional space for managing data structures, e.g., to link lists together, store lengths of variable-sized arrays, or practically realize hash tables. Thus, a practical implementation implies increasing space requirements of filtering algorithms compared to our theoretical analysis.

Next to this general increase in memory requirements, data structures have to be implemented in a reasonable way, e.g., they have to support dynamic growing and shrinking if this is demanded in practice. Generally, constantly required data structures require a dynamic implementation. For data structures solely used in the event filtering process, i.e., fulfilled predicate, predicate bit and hit vector, a static implementation is sufficient due to the requirement of initializing them for each filtered event.

We have implemented such dynamic data structures in a space-efficient manner for our practical evaluation. Their comparison to the memory requirements of standard implementations, i.e., STL hash (multi) sets, has resulted in much less space consumptions for our implementations. We present our practical analysis in the next section.

5 Practical Analysis of Memory Requirements and Efficiency

In Sect. 3 and 4 we have presented a theoretical investigation of memory requirements of filtering algorithms and described the influence of a practical implementation on our theoretical results. In this section we extend our theoretical work and show the applicability of our theoretical results to practical settings (Sect. 5.1). Furthermore, we present a brief comparison of efficiency characteristics of the compared algorithms (Sect. 5.2).

We compare the non-canonical approach to one canonical algorithm by experiment. Because of the restrictions of the cluster approach (cf. Sect. 3.2), we have chosen the counting algorithm as representative of canonical algorithms for our practical analysis. This allows the generalization of our results to other settings than equality predicate-based application areas and areas dealing with less predicate redundancy as assumed by the cluster approach. Furthermore, the counting algorithm behaves more space efficient than the cluster approach (cf. Fig. 1). In a practical implementation the cluster approach without efficiently supported unsubscriptions and the counting approach show nearly the same memory requirements [6].

5.1 Practical Analysis of Memory Requirements

In this section, we compare the memory requirements of the counting algorithm and the non-canonical approach. Our actual implementations of these algorithms follow our descriptions in Sect. 4.2.

In our experiments, we want to verify our results shown in Fig. 1(c), i.e., in case of $op^r = 0.5$. Here we present the memory usage of the required data structures[4] in case of $1,000,000$ registered subscriptions with a growing number of predicates per subscription $|p|$ and a growing number of disjunctively combined elements after conversion S_s. For the parameter s^r (relative number of conjunctive elements per predicate after conversion), we have chosen to present the cases $s^r = 0.3$ and $s^r = 0.7$.

Our results are presented in three-dimensional figures. Figure 2(a) shows both algorithms in case of $s^r = 0.3$; Fig. 3(a) presents the case of $s^r = 0.7$. The x-axes in the figures represent the number of predicates per subscription $|p|$ ranging from 5 to 50, z-axes show the number of disjunctively combined elements after conversion S_s in the range of 1 to 5. The actually required amount of memory for holding the required data structures is illustrated at the y-axes of the figures.

There are two surfaces shown in each of the figures. The brighter ones illustrate the behaviors of the counting algorithm, the darker ones represent the non-canonical approach. As shown in our theoretical analysis, the non-canonical approach does not change its memory usage with growing S_s. Thus, its surface does always show the same memory requirements (y-axis) regardless of S_s (z-axis), e.g, approx. 900 MB for $|p| = 50$. This holds for both figures, Fig. 2(a) and 3(a), since the memory requirements of the non-canonical approach are independent of s^r. The counting algorithm, however, shows increasing memory requirements with growing S_s as described in (1). Furthermore, according to (1), increasing s_p (and thus s^r) results in advanced space usage.

[4] We show the total memory requirements of our filtering process to allow for the incorporation of all influencing parameters, e.g., heap management structures.

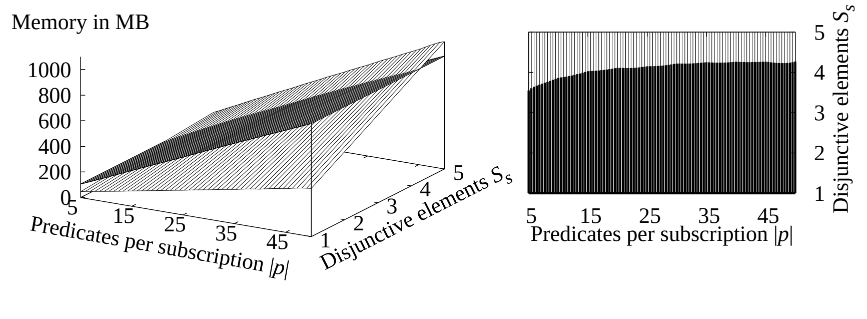
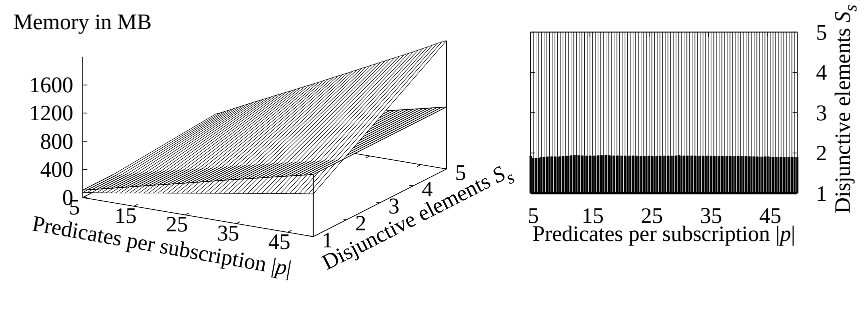


Fig. 2. Memory requirements in our practical experiments in case of $s^r = 0.3$, $op^r = 0.5$ and $|s| = 1,000,000$ (cf. Fig. 1(c) for theoretical results in the same scenario)

As depicted in our theoretical comparison in Fig. 1(c), there exists a point of interchanging memory requirements of canonical and non-canonical algorithms. This point is denoted by a cutting of the surfaces of the two algorithms. In Fig. 2(a), this cutting occurs at $S_s \approx 4$, in Fig. 3(a) it happens at $S_s \approx 2$. To exactly determine the point of cutting surfaces we present a top view of the diagrams in Fig. 2(b) and Fig. 3(b), respectively. Figure 2(b) shows that the point of interchanging memory requirements can be found between $S_s = 3$ and $S_s = 5$ dependent on $|p|$. For $s^r = 0.7$ (Fig. 3(b)), it is always located slightly below $S_s = 2$. Comparing these practical results to our theoretical results in Fig. 1(c), we realize that our theoretical analysis has predicted nearly the same behavior of the two algorithms: Even if only 2 ($s^r = 0.7$) or 4 ($s^r = 0.3$) disjunctively combined elements S_s are created by canonical conversions, a non-canonical approach is favorable. Thus, our practical experiments verify our theoretical results and show their correctness even in case of a certain practical implementation.

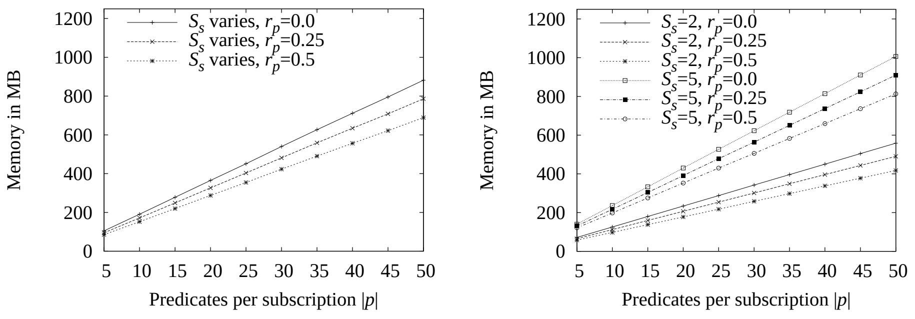
Practical Analysis of Influences of Redundancy. In our theoretical analysis we have shown that predicate redundancy r_p does not influence the memory requirements of algorithms. However, in a practical realization this property does not hold. The influence of r_p on our implementation is illustrated in Fig. 4. Ordinates show an increasing number of predicates per subscription $|p|$, abscissae are labeled with the required memory in MB. In this experiment we have registered $1,000,000$ subscriptions, further characterizing parameters are $op^r = 0.5$ and $s^r = 0.3$. The behavior of the non-canonical approach with varying predicate redundancy is shown in Fig. 4(a), the counting algorithm is presented in Fig. 4(b) for varying S_s and r_p.

Both algorithms show decreasing memory requirements with increasing r_p. This behavior results out of the decreasing memory overhead in a practical implementation: Both algorithms utilize a predicate subscription association table, which requires a dynamic implementation causing more memory usage. If there are less unique predicates,



(a) Perspective view (b) Top view

Fig. 3. Memory requirements in our practical experiments in case of $s^r = 0.7$, $op^r = 0.5$ and $|s| = 1,000,000$ (cf. Fig. 1(c) for theoretical results in the same scenario)



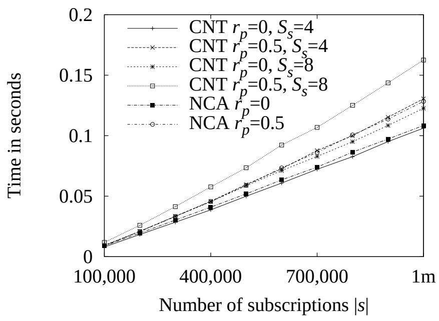
(a) Non-canonical algorithm (b) Canonical: counting algorithm

Fig. 4. Influence of predicate redundancy r_p on the algorithms in case of $|s| = 1,000,000$, $op^r = 0.5$ and $s^r = 0.3$

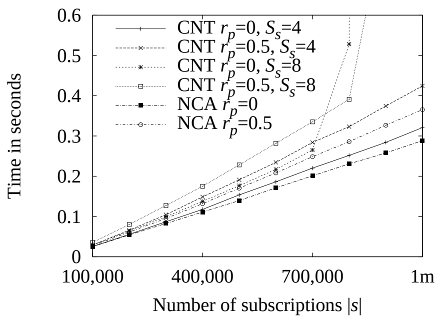
which is caused by predicate redundancy, the amount of memory overhead decreases. Thus, the total memory requirements decrease as observable in Fig. 4.

5.2 Practical Analysis of Efficiency

We are aware of the correlation between memory usage and filter efficiency of filtering algorithms: We cannot utilize the most space efficient algorithm in practice if it shows poor time efficiency. Vice versa, time efficient solutions, such as [7], might become inapplicable due to their memory requirements [3]. Thus, in our analysis we also compared the time efficiency of the counting (CNT) and the non-canonical approach (NCA) to confirm the applicability of the non-canonical approach in practice. In our experiments, we only have to compare the time efficiency of subscription matching, since predicate matching works the same in both algorithms. Time efficiency is represented



(a) $|p| = 10$



(b) $|p| = 30$

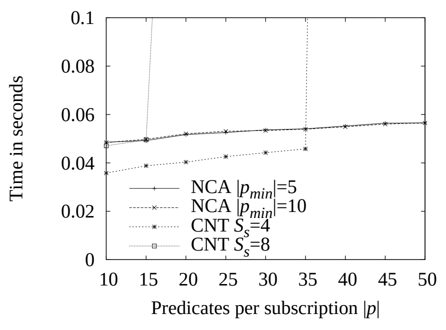
Fig. 5. Influence of number of subscriptions $|s|$ for varying r_p and S_s

by the average filtering time for subscription matching per event, i.e., increasing times denote decreasing efficiency. We ran our experiments several times to obtain negligible variances. Thus, in the figures we only show the mean values of filtering time.

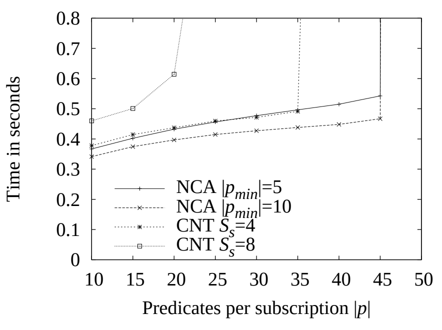
Figure 5 shows the influence of the number of subscriptions registered with the pub/sub system. In Fig. 5(a), we have used $|p| = 10$, Fig. 5(b) illustrates time efficiency in case of $|p| = 30$. We show the behavior of the counting algorithm for the two cases $S_s = 4$ and $S_s = 8$. Predicate redundancy is chosen with $r_p = 0.0$ and $r_p = 0.5$. We also present the non-canonical approach assuming the worst case behavior, i.e., if a candidate subscription is evaluated, its whole Boolean expression is analyzed. Thus, we always test entire subscription trees in our experiments. In this experiment, we have increased the number of fulfilled predicates per event p_e with growing subscription numbers: $p_e = \frac{|s||p|}{50}$. We have chosen the minimum number of fulfilled predicates required for matching $|p_{min}|$ with 5 in case of $|p| = 10$ and with 10 in case of $|p| = 30$.

Figure 5 illustrates the average filtering times at the ordinates. Both algorithms show linearly increasing filtering times in case of growing subscription numbers. In case of $S_s = 8$ and $|p| = 30$ (Fig. 5(b)), the counting algorithm requires more memory than the available resources (sharp bends in curves). Thus, the operation system starts page swapping resulting in strongly increasing filtering times in case of more than $700,000$ and $800,000$ subscriptions (according to r_p, cf. Sect. 5.1). Generally, increasing predicate redundancy r_p leads to growing filtering times for both algorithms in the evaluated setting. This is due to the fact that more candidate subscriptions have to be evaluated (non-canonical algorithm) and more counters have to be increased in the hit vector (both algorithms). The counting algorithm in case of $S_s = 8$ always shows the worst time efficiency. According to the number of predicates $|p|$, either the non-canonical approach (Fig. 5(b)) or the counting algorithm with $S_s = 4$ (Fig. 5(a)) are the most efficient filtering approaches (nearly on par with the other approach).

The influence of $|p|$ is shown in Fig. 6. In Fig. 6(a), it holds $p_e = 50,000$, Fig. 6(b) shows the case of $p_e = 1,000,000$. For the non-canonical approach we analyzed the two settings $|p_{min}| = 5$ and $|p_{min}| = 10$. The counting approach is presented in two variants with $S_s = 4$ and $S_s = 8$. We have registered $1,000,000$ subscriptions.



(a) $p_e = 50,000$



(b) $p_e = 1,000,000$

Fig. 6. Influence of number of predicates $|p|$ for varying p_e

Again, sharp bends in the curves in Fig. 6 denote the point of exhausted main memory resources. The non-canonical approach shows the best scalability, followed by the counting approach in case of $S_s = 4$. We can also observe improved time efficiency in the non-canonical approach in case of higher $|p_{min}|$. This effect becomes more apparent with a high value of p_e (Fig. 6(b)) due to more candidate subscriptions requiring evaluation. In case of a small number of fulfilled predicates per event p_e, the counting (case $S_s = 4$) is more efficient than the non-canonical algorithm; large numbers of p_e clearly favor the non-canonical approach. The reason is the increased number of hits (incrementing the hit vector) in the counting approach due to canonical conversions.

Our efficiency analysis shows that counting and non-canonical approach perform similarly for increasing problem sizes. In some cases, the counting approach shows slightly better time efficiency, other settings favor the non-canonical approach. For large S_s, the non-canonical approach shows both better time and space efficiency. Thus, a non-canonical solution offers better scalability properties in these situations.

6 Conclusions and Future Work

In this paper, we have presented a detailed investigation of two classes of event filtering approaches: canonical and non-canonical algorithms. As a first step we introduced a characterization scheme for qualifying primitive subscriptions in order to allow for a description of various practical settings. Based on this scheme, we thoroughly analyzed the memory requirements of three important event filtering algorithms (counting [1, 12], cluster [6, 8] and non-canonical [3]). We compared our results to derive conclusions about the circumstances under which canonical algorithms should be preferred in respect to memory usage and which settings favor non-canonical approaches.

To show the applicability of our theoretical results in a practical implementation, we proposed an implementation and investigated its memory requirements by experiment. This practical evaluation clearly verified our theoretical results: Even when conversions to canonical forms result in only two canonical subscriptions (i.e., subscription use only one disjunction), a non-canonical approach is favorable.

We also correlated the memory requirements of the practically analyzed algorithms to their filter efficiency. Generally, non-canonical algorithms show approximately the same time efficiency as canonical ones. In case of increasing numbers of disjunctions in subscriptions, the time efficiency of non-canonical approaches improves compared to canonical solutions. In this case, a non-canonical approach also shows much better scalability properties as demonstrated in our analysis of memory requirements. Thus, if subscriptions involve disjunctions, non-canonical algorithms are the preferred class of filtering solutions due to their direct exploitation of subscriptions in event filtering.

For future work, we plan to describe different application scenarios using our characterization scheme. A later analysis of these scenarios will allow conclusions about the preferred filtering algorithm for these applications. We also plan to further extend the non-canonical filtering approach to a distributed algorithm.

References

1. G. Ashayer, H. A. Jacobsen, and H. Leung. Predicate Matching and Subscription Matching in Publish/Subscribe Systems. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, Vienna, Austria, July 2–5 2002.
2. S. Bittner and A. Hinze. Investigating the Memory Requirements for Publish/Subscribe Filtering Algorithms. Technical Report 03/2005, Computer Science Department, University of Waikato, May 2005.
3. S. Bittner and A. Hinze. On the Benefits of Non-Canonical Filtering in Publish/Subscribe Systems. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '05)*, pages 451–457, Columbus, USA, June 6–10 2005.
4. A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient Filtering in Publish-Subscribe Systems using Binary Decision Diagrams. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001.
5. A. Carzaniga and A. L. Wolf. Forwarding in a Content-Based Network. In *Proceedings of the 2003 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*, Karlsruhe, Germany, March 2003.
6. F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proceedings of the 2001 ACM SIGMOD*, pages 115–126, Santa Barbara, USA, May 21–24 2001.
7. J. Gough and G. Smith. Efficient Recognition of Events in a Distributed System. In *Proceedings of the 18th Australasian Computer Science Conference*, Adelaide, Australia, 1995.
8. E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD 1990)*, Atlantic City, USA, May 23–25 1990.
9. G. Mühl and L. Fiege. Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs. *IEEE DSOnline*, 2(7), 2001.
10. F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, pages 431–442, San Diego, USA, June 9–12 2003.
11. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG97)*, Brisbane, Australia, September 3–5 1997.
12. T. W. Yan and H. García-Molina. Index Structures for Selective Dissemination of Information Under the Boolean Model. *ACM Transactions on Database Systems*, 19(2), 1994.