Investigating the 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 to disjunctive normal forms (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[1] of these systems from their memory requirements [3]. This characteristic implies the need to economize the usage of memory resources. We further discuss this topic in Sect. 2.1.

We can distinguish between two classes of filtering approaches for pub/sub systems: (i) algorithms directly filtering on Boolean subscriptions [3, 4, 13] (referred to as

[1] We do not focus on distributed pub/sub systems in this paper. There scalability also depends on the network traffic generated by the system.

non-canonical approaches in the following), and (ii) algorithms filtering on subscriptions in canonical forms [2, 5, 6, 8, 14] (referred to as canonical approaches). Internally, algorithms of Class (ii) either filter on disjunctive normal forms (DNF) [5] or only support conjunctive subscriptions [2, 6, 8, 14]. 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 [10].

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 as shown in [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 (a detailed discussion can be found 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 the details of our 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 analyze the counting [2, 14] 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 (other non-canonical approaches, e.g., [4, 13], do not use indexes at all). 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 shows the importance of minimizing memory usage in pub/sub systems (Sect. 2.1), gives an overview of the analyzed algorithms (Sect. 2.2), and presents related work (Sect. 2.3). Our characterization scheme qualifying subscriptions can be found in Sect. 3. We theoretically analyze the memory requirements of filtering algorithms in Sect. 4. Section 5 includes a comparison of the theoretical memory usages, their graphical presentation, and considerations for implementations of algorithms. We practically verify our results in Sect. 6.1, followed by the correlation of memory usage to filtering efficiency in Sect. 6.2. Finally, we conclude and present our future work in Sect. 7.

2 Motivation, Analyzed Algorithms and Related Work

In this section, we firstly outline the importance of minimizing memory usage in pub/sub systems (Sect. 2.1). Then, in Sect. 2.2 we give an overview of the three analyzed filtering algorithms, namely the counting approach [2, 14], the cluster approach [6, 8] and the non-canonical approach [3]. Finally, we present related work comparing the memory requirements of filtering solutions for pub/sub systems in Sect. 2.3.

2.1 Impact and Importance of Memory Usage

One important lesson from the recent developments in computer hardware is the availability of reasonably cheap large main memories. However, this situation does not conversely imply that we do not have to consider the main memory requirements of algorithms: Today it is more feasible to apply space consuming approaches. Though, this observation only holds[2] (to a certain extend) in wired scenarios dealing with designated machines, e.g., powerful event filtering servers. In general and widely used peer-to-peer settings, we will find conventional machines equipped with reasonable (processor and memory) resources but not satisfying the latest developments. For mobile devices or sensors, this situation holds even more. Altogether, this is leading to the need to consider memory requirements of algorithms, especially if targeting real world scenarios not following the traditional client/server approach in wired settings.

In the area of pub/sub systems, the development of larger main memories has led to the implementation of main memory filtering approaches, e.g., [1, 2, 4, 6, 7]. These algorithms allow for a more efficient event filtering process than approaches comprising secondary memory data structures. This is due to the possibility of neglecting disk accesses (or, in general, secondary memory accesses) in main memory solutions and thus the optimization of these filtering approaches to purely situate in main memory.

However, such filtering approaches require main memories large enough to hold and efficiently filter incoming events against all registered subscriptions. Otherwise, their applicability in case of increasing problem sizes, i.e., their scalability, is highly restricted. This direct dependency of main memory filtering algorithms on available resources leads to another important quality measure for event filtering approaches next to time efficiency: space efficiency (memory usage) as an important determinant of scalability [12].

To outbalance the direct dependency of event filtering on available main memory resources, current algorithms have restricted their subscription languages to merely support simple conjunctive subscriptions [2, 6, 8, 14] or DNFs [5]. This leads to decreasing memory usages due to the ability to neglect the storage of combinations of Boolean expressions in subscriptions themselves. However, arbitrary Boolean subscriptions have to be converted to canonical forms (DNFs) to allow for their matching in such approaches. These canonical forms are exponential in size [11] leading to enormous memory requirements if supporting Boolean subscriptions.

This general practice of canonical rewriting has been effectively applied in the context of database systems. However, in pub/sub systems we find the converse problem

[2] Assuming substantial financial resources.

definition as in database systems: Database systems execute few queries at one time on a large amount of data; pub/sub systems have to filter incoming events against a huge number of continuously registered subscriptions. Thus, the internal representation of subscriptions and their memory requirements directly influence pub/sub systems[3].

Despite this neglect of filtering on arbitrary Boolean subscriptions in today's approaches, comparative evaluations of pub/sub systems ignore the effect of canonical conversions on space efficiency of filtering algorithms. Furthermore, they mainly target at the comparison of filter (time) efficiency. However, as we have described before, it is vitally required to consider the memory usage of filtering approaches because it is directly implying the scalability properties of algorithms. Hence, our investigation of the space usages of current filtering algorithms will yield to clarifications about their scalability characteristics.

2.2 Review of Analyzed Algorithms

We now give a brief outline of the three filtering algorithms (counting approach, cluster approach, non-canonical approach) we use in our later analysis. Our decision to focus on these one-dimensional indexing approaches is based on their balanced time and space efficiency characteristics: They consume less memory than multi-dimensional indexing algorithms [3] but allow for a more efficient filtering compared to non-indexing solutions [3, 9].

We restrict this section to a short review of the algorithms and refer to the original works for a thorough study and description of the approaches. An analysis of the theoretical memory requirements of these algorithms is addressed in the next section.

Review of the Counting Algorithm as Canonical Approach. The counting algorithm was originally proposed by Yan and Garcia-Molina in [14] 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., [2]. It only supports conjunctive subscriptions and requires the conversion of subscriptions involving disjunctions to DNFs. Then, each element (i.e., a conjunction) participating in the one disjunction of the DNF is treated as separate subscription [10].

The counting algorithm (cf. Fig. 1 for a graphical representation) requires a subscription predicate count vector to store the total number of predicates per subscription. Furthermore, a hit vector is utilized to accumulate the number of matching predicates per subscription in the filtering process. To determine subscriptions that involve a certain predicate, a predicate subscription association table is used. A subscription predicate association table is required if unsubscriptions (deregistrations) are supported to allow for the determination of all predicates a subscription consists of [2].

Event filtering works in two steps: Firstly, all matching predicates are determined utilizing one-dimensional indexes (predicate matching step). Secondly (subscription matching), all subscriptions involving the matching predicates are derived by exploiting the predicate subscription association table. For each matching predicate a counter in

[3] We refer to [3] for a further discussion of these differences between database management and pub/sub systems as well as their consequences.

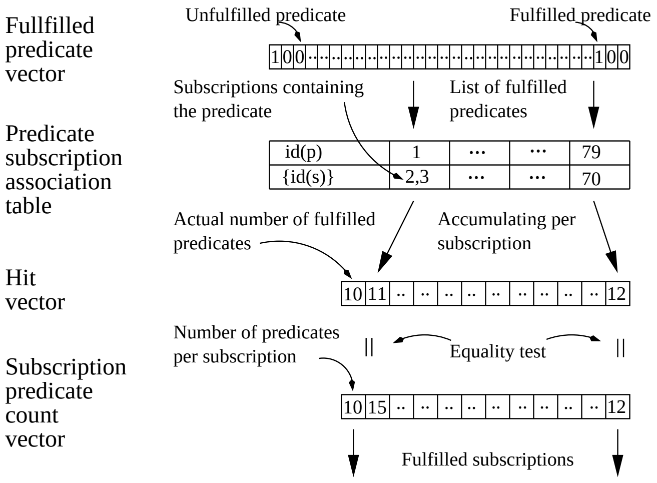

Fullfilled predicate vector
Predicate subscription association table
Hit vector
Subscription predicate count vector
Unfulfilled predicate
Fulfilled predicate
1 0 0 ·· 1 0 0
Subscriptions containing the predicate
List of fulfilled predicates
id(p) | 1 | ··· | ··· | 79
{id(s)} | 2,3 | ··· | ··· | 70
Actual number of fulfilled predicates
Accumulating per subscription
10 11 ·· 12
Number of predicates per subscription
Equality test
10 15 ·· 12
Fulfilled subscriptions

Fig. 1. Overview of the subscription matching process in the counting algorithm

the hit vector is increased for the respective subscriptions. Finally, hit vector and subscription predicate count vector are compared. If both vectors show the same value for a subscription (i.e., all predicates of a conjunctive subscription are matching), this subscription is matching. An overview of the subscription matching step and the utilized data structures is shown in Fig. 1. For a detailed description we refer to [2, 14].

Review of the Cluster Algorithm as Canonical Approach. The cluster algorithm is described in detail in [6] and is based on the algorithm presented by Hanson et al. 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 as explained for the counting algorithm.

The cluster algorithm uses one-dimensional indexes to allow for a fast determination of predicates matching an incoming event. In [6] the notion of access predicates is introduced which are predicates of subscriptions that have to be fulfilled by an incoming event in order to lead to fulfilled subscriptions.

The cluster algorithm (cf. Fig. 2 for a visual overview) applies a cluster vector that stores references to clusters. This cluster vector contains of a list of clusters for each access predicate. Subscriptions with the same access predicate are grouped in clusters according to their numbers of predicates. Inside the clusters, subscriptions are represented by their identifiers and their predicates.

Again, event filtering works in two steps: In the beginning, all matching predicates are determined by the help of one-dimensional indexes (predicate matching step). For all matching access predicates, we can find clusters with potentially matching subscriptions by utilizing the cluster vector. Then, subscriptions inside these clusters are evaluated by testing if all their predicates have been fulfilled (a list of matching predicates was the result of the first filtering step). This second step of filtering is called subscrip-

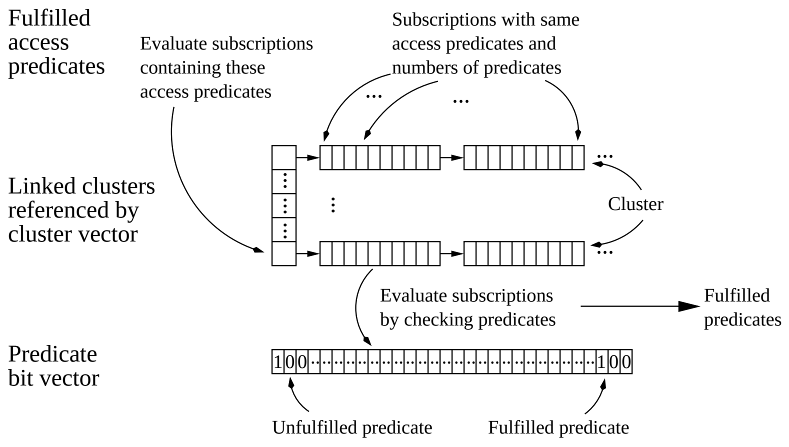

Fulfilled access predicates
Evaluate subscriptions containing these access predicates
Subscriptions with same access predicates and numbers of predicates
Linked clusters referenced by cluster vector
Cluster
Evaluate subscriptions by checking predicates
Fulfilled predicates
Predicate bit vector
1 0 0 1 0 0
Unfulfilled predicate
Fulfilled predicate

Fig. 2. Overview of the subscription matching process in the cluster algorithm

tion matching and is illustrated in Fig. 2 as well as the required data structures. The work in [6] describes this algorithm in detail.

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

Also the non-canonical algorithm exploits one-dimensional indexes to efficiently determine predicates matching incoming events. Subscriptions are encoded in subscription trees representing their Boolean structure and involved predicates[4]. A subscription location table is required to associate subscription identifiers to memory addresses of subscription trees. To access subscriptions out of given predicates, a predicate subscription association table is applied. Furthermore, a minimum predicate count vector states the minimal number of fulfilled predicates required for each subscription to match (variation of the original approach in [3]). A hit vector is used to accumulate the number of fulfilled predicates per subscription (also a variation from [3]).

Also this matching approach involves a two-step event filtering process beginning with the determination of matching predicates using one-dimensional indexes (predicate matching). Then, all potential candidate subscriptions are determined (subscription matching step) by the help of the predicate subscription association table.

The work in [3] proposes to evaluate subscription trees of all candidate subscriptions using the subscription location table. However, if using a minimum predicate count vector, only subscriptions with more matching predicates than the number stored in

[4] In subscription trees we push down the unary NOT operator using De Morgan's laws to avoid negations in inner nodes. Thus, we integrate all negations into leaf nodes. Then, we utilize the inverse operator for negated predicates (e.g., the negation of predicate $attribute = 10$ is expressed as $attribute \neq 10$).

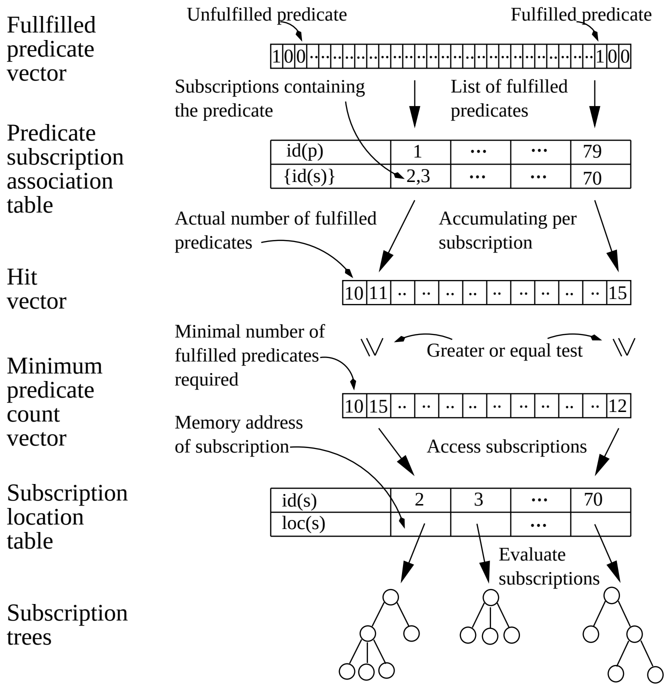

Fullfilled predicate vector
Predicate subscription association table
Hit vector
Minimum predicate count vector
Subscription location table
Subscription trees
Unfulfilled predicate
Fulfilled predicate
Subscriptions containing the predicate
List of fulfilled predicates
id(p)
{id(s)}
1
2,3
79
70
Actual number of fulfilled predicates
Accumulating per subscription
10
11
15
Minimal number of fulfilled predicates required
Greater or equal test
10
15
12
Memory address of subscription
Access subscriptions
id(s)
loc(s)
2
3
70
Evaluate subscriptions

Fig. 3. Overview of the subscription matching process in the non-canonical algorithm

this vector (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 the hit vector. A figurative overview of the data structures involved in the subscription matching process as well as the process itself is illustrated in Fig. 3.

2.3 Limitations of Previous Comparative 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.

Ashayer et al. evaluate several implementations of the counting algorithm in [2], 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 10 different values. Thus, the results of [2] cannot be generalized to more complex and sophisticated settings utilizing expressive Boolean subscription languages. Additionally, a satisfactory theoretical evaluation is missing.

Fabret at el. compare implementations of counting and cluster algorithm in [6]. However, the assumptions are similarly restricted as in [2]: only five predicates are used per subscription, and domains consist of 35 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 the behavior in general settings and are mainly targeting filter efficiency.

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

3 Characterization of Subscriptions

In this section, we present our approach of characterizing subscriptions. Since we target the evaluation of memory requirements, our methodology is based on attributes affecting the memory usage for storing subscriptions for efficient filtering. Our characterization approach also allows for a successful representation of the space requirements of the three filtering algorithms presented in Sect. 2.2.

Generally, we require parameters to express the characteristics of Boolean subscriptions (canonical and non-canonical algorithms) and to characterize canonical conversions (required by canonical approaches). We also need to model algorithm-specific parameters and to incorporate the relation between subscriptions and events affecting both time and space efficiency of algorithms.

Altogether, 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. 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 parameter p_e of Class E incorporates the relation between subscriptions and events, which influences both space and time efficiency of filtering algorithms. In the following, we describe these 14 parameters in detail.

Number of predicates per subscription $|p|$: This parameter directly characterizes subscriptions (Class S) and describes the average number of predicates used in subscriptions. Since subscriptions are Boolean expressions, $|p|$ states the average number of input variables.

Number of Boolean operators per subscription $|op|$: Subscriptions are Boolean expressions combined by Boolean operators[5]. The average number of operators used in a subscription is denoted by $|op|$. This parameter classifies into Class S.

Relative number of Boolean operators per subscription op^r: To reduce the number of characterizing parameters specifying fixed values in our later analysis, we introduce another parameter op^r in Class S. This parameter describes the relative number of operators per subscription, i.e., the proportion of operators $|op|$ compared to the number of predicates per subscription $|p|$. Thus, it holds $op^r = \frac{|op|}{|p|}$.

[5] We assume AND, OR and NOT as possible Boolean operators.

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

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

Number of subscriptions $|s|$**:** The number of subscriptions that have actually been registered with the pub/sub system is referred to as $|s|$. Since $|s|$ describes subscriptions, this parameter belongs to Class S.

Number of unique predicates $|p_u|$**:** In order to model predicate redundancy, we require to specify the number of unique predicates registered with a pub/sub system. Unique predicates utilize the same predicate identifier and are stored only once. However, such predicates might be specified in several subscriptions. The parameter $|p_u|$ describes the number of unique predicates (which is a characteristic of subscriptions and thus belongs to Class S) that have been registered with the system.

Predicate redundancy r_p**:** Predicate redundancy r_p describes the degree of uniqueness of predicates, i.e., r_p determines if there are subscriptions that utilize the same predicates. We define predicate redundancy r_p (a member of Class S since it characterizes subscriptions) as the proportion of shared predicates among all registered predicates, i.e., $r_p = 1.0 - \frac{|p_u|}{|p||s|}$. Generally, high predicate redundancy occurs in cases of small domain sizes and users with similar interests.

Width of subscription identifiers $w(s)$**:** We assume subscriptions to be uniquely identifiable. The parameter $w(s)$ describes the width of subscription identifiers in bytes.

If we are using 32-bit integers it holds $w(s) = 4$. This implementation-specific parameter belongs to Class A.

Width of predicate identifiers $w(p)$**:** We also presume that predicates can be uniquely identified. This allows for the usage of predicates in several subscriptions without the storage of these predicates more than once. The parameter $w(p)$ (again an implementation-specific parameter of Class A) specifies the width of predicate identifiers. Again, using 32-bit integers as identifiers implies $w(p) = 4$.

Width of subscription locations $w(l)$**:** When storing subscriptions themselves, e.g., in a tree structure [3], we have to be able to locate them (e.g., at a memory address). The width of such a location identifier is denoted by $w(l)$. If utilizing memory pointers on 32-bit machines, it holds $w(l) = 4$. Due to merely requiring this parameter out of implementation reasons, $w(l)$ classifies into Class A.

Width of cluster references $w(c)$**:** Besides storing subscriptions themselves, we could cluster subscriptions according to their access predicates [6]. The width of a reference to such a cluster is stated by $w(c)$. Again, using memory pointers on 32-bit machines leads to $w(c) = 4$ (and thus $w(c)$ classifies into Class A).

Number of disjunctively combined elements after conversion S_s**:** Some (i.e., canonical) algorithms [2, 6, 8, 14] require purely conjunctive subscriptions or subscriptions in DNFs. Thus, to allow for the filtering of arbitrary Boolean subscriptions, a conversion to DNF is required. The parameter S_s describes the average number of conjunctive elements per subscription participating in the disjunction of a DNF. This parameter belongs to Class C since it describes characteristics of canonical conversions.

Number of conjunctive elements per predicate after conversion s_p**:** A conversion to DNFs implies that predicates of an arbitrary Boolean subscription participate in several conjunctive elements. The average number of conjunctive elements a predicate from an original subscription (i.e., before conversion) is involved in after its conversion is denoted by s_p[6]. Also this parameter classifies into Class C because it is purely conversion-related.

Relative number of conjunctive elements per predicate after conversion s^r**:** We can further reduce the number of fixed parameters by the help of s^r. This parameter s^r denotes the relative number of conjunctive elements a predicate belongs to after conversion, i.e., compared to the total number of disjunctively combined elements per subscription after conversion S_s. We define s^r as $s^r = \frac{s_p}{S_s}$. Since s_p and S_s belong to Class C, also s^r classifies into this class.

Number of fulfilled predicates per event p_e**:** In a pub/sub system, each of the incoming events fulfils a certain number of predicates. The average number of predicates fulfilled by an incoming event is denoted by p_e. This parameter classifies into Class E since it incorporates the relation between subscriptions and events.

Altogether, these 14 parameters allow to characterize subscriptions (Class S), to derive the major memory requirements of filtering algorithms (Class S, A and C), and to describe the relation between events and subscriptions (Class E) affecting the time efficiency of event filtering. Our theoretical analysis in Sect. 4 and comparison in Sect. 5

[6] In case of predicate redundancy each occurrence of the predicate is treated separately.

abstracts from an assignment of certain values by directly utilizing these characterizing parameters.

After this introduction of the parameters required for the determination of space requirements, we proceed with our detailed analysis of memory usages of three event filtering algorithms in the next section.

4 Theoretical Analysis of Memory Requirements

After presenting the parameters required to analyze the memory requirements of filtering algorithms, we now continue with our actual analysis in Sect. 4.1 to Sect. 4.3. 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. 5 as well as considerations for practical implementations.

4.1 Theoretical Analysis of the Counting Algorithm

We now want to analyze the memory requirements of the counting algorithm [2, 14] in respect to the characterizing parameters defined in Sect. 3. According to [2] and our review in Sect. 2.2, 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 implementation or a bit vector implementation. This decision should depend on the proportion of matching predicates. A bit vector implementation requires at least $\frac{|p||s|}{8}$ bytes; an ordinary vector implementation involves at least $p_e w(p)$ bytes to store matching predicates. Thus, the fulfilled predicate vector demands $\min(\frac{|p||s|}{8}, p_e w(p))$ bytes.
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 representing the hit counter. 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. This value is compared to the corresponding entry in the hit vector in the subscription matching step of the algorithm. According to our assumption for the hit vector, we can represent each subscription by a 1-byte entry in the subscription predicate count vector. Thus, we require $|s|S_s$ bytes in total due to the applied canonical conversion (cp. hit vector).
Similar to the hit vector, the subscription predicate count vector does not depend on predicate redundancy (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 conversion. 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 the predicate subscription association 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 $r_pw(s)s_p|p||s|$ bytes. Thus, predicate redundancy 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 [2]: To remove a subscription, we have to search through all entries in the predicate subscription association table. Whenever we find the subscription identifier of the subscription to be unsubscribed, we have to remove it from the list of subscriptions. If no further subscriptions remain, we can remove the predicate itself and its occurrence in the index structures. However, this operating expense is not feasible in practice and implies the need for subscription predicate associations.
Least memory for subscription predicate associations is utilized when using subscription identifiers 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 predicate subscription association 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 in case of neglecting efficient unsubscriptions, i.e., without the utilization of a subscription predicate association table, we require the following amount of memory in bytes (we exclude the fulfilled predicate vector since it is utilized by all three analyzed algorithms)

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

When efficiently supporting unsubscriptions (utilizing a subscription predicate association table) the required data structures use the following accumulated amount of memory in bytes

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

These observations hold in all cases of predicate redundancy r_p as shown above.

4.2 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. However, this algorithm has several restrictions (e.g., usage of highly redundant equality predicates) and strongly depends on 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. We refer to [6] for a detailed description of the data structures analyzed in the following.

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][7]. 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 (cf. Sect. 4.1). However, we require a bit vector implementation (as stated in [6]) due to the requirement of accessing the state of predicates

[7] 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.

(fulfilled or not fulfilled) directly. Thus, we demand $\frac{|p||s|}{8}$ bytes for the predicate bit vector.
Since the cluster algorithm assumes high predicate redundancy and uses a bit vector due to performance aspects, it takes implicitly advantage of high predicate redundancy in respect to memory usage. However, if the fraction of fulfilled predicates is quite small, the predicate bit vector cannot profit from this fact.

Cluster vector: This vector contains references to subscription cluster lists. The number of entries highly depends on the number of actual access predicates. In turn, this number is dependent on registered subscriptions and application semantics. Due to this unpredictability we neglect the memory requirements for this data structure in our following analysis. Furthermore, its memory usage is only a small fraction of the space requirements of other data structures.
Generally, predicate redundancy r_p results in a smaller cluster vector due to less access predicates (in fact, $r_p = 0.0$ contradicts the assumption of access predicates used in this algorithm).

References to cluster vector: Access predicates stored in indexes are provided with references to the cluster vector, i.e., to subscriptions containing these access predicates and requiring them to be fulfilled. Due to the strong dependency of access predicates from the current application and registered subscriptions, we neglect the memory requirements of references to the cluster vector in our analysis. Also their space usage is quite small compared to other data structures used in the cluster algorithm.
Predicate redundancy r_p decreases the memory requirements for references to the cluster vector. However, $r_p > 0.0$ is an inherent assumption of the cluster algorithm. Cases of no predicate redundancy conflict with the assumptions of shared access predicates and a clustering according to them.

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 list.
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, i.e., if predicates are redundant or unique.

Subscription cluster table: This table is one additional data structure required to support efficient unsubscriptions. 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 its storage of $|s|S_s$ cluster references.
Also this table is focussed on mappings to 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 being contained in $w(s)s_p$ subscriptions on average.

Similar to our observation for the counting algorithm (Sect. 4.1), for unique predicates we require $(1.0 - r_p)w(s)s_p|p||s|$ bytes to store their associations with subscriptions. Redundant predicates consume $r_p w(s)s_p|p||s|$ bytes. Thus, predicate redundancy does not influence the size of the predicate subscription associations.

Accumulating the memory requirements of the former mentioned data structures (excluding the predicate bit vector) leads to the following number of bytes in case of not supporting efficient unsubscriptions

$$\text{mem}_{cluster} = |s|(S_s w(s) + s_p|p|w(p)) \ . \tag{3}$$

The inclusion of efficient unsubscriptions increases the memory requirements to the following amount of bytes

$$\text{mem}_{cluster,unsub} = |s|(S_s w(s) + s_p|p|w(p) + S_s w(c) + s_p|p|w(s)) \ . \tag{4}$$

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

4.3 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. 2.2. Our variation includes another encoding of subscription trees and the exploitation of a minimum predicate count vector.

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). Children themselves are stored afterwards. In contrast to [3], we do not store the width of the children 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 special case of no predicate redundancy. 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. An actual realization should depend on the fraction of fulfilled predicates (cf. Sect. 4.1). Thus, either as ordinary vector or bit vector implementation it requires $\min(\frac{|p||s|}{8}, p_e w(p))$ bytes of memory.
According to our reasoning in Sect. 4.1, high predicates redundancy r_p favors a bit vector implementation and a small fraction of fulfilled predicates per event benefits an ordinary vector implementation.

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, storing Boolean operators and numbers of children, demand $2|op|$ bytes of memory for each subscription. Thus, for all registered subscriptions $|s|((w(p) + 1)|p| + 2|op|)$ bytes are used.
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 for storing these associations.
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 conversions to canonical forms. Thus, predicates are involved in less subscriptions (only one subscription in case of $r_p = 0.0$). Altogether, $|s||p|w(s)$ bytes are required for the predicate subscription association table.
Similar to the counterparts of this table in the other two algorithms (cf. Sect. 4.1 and Sect. 4.2), for unique predicates in cases of predicate redundancy $r_p|s||p|w(s)$ bytes are required. Moreover, redundant predicates consume $(1.0 - r_p)|s||p|w(s)$ bytes. Thus, summing up, the memory usage of the predicate subscription association table does not depend on predicate redundancy.

Hit vector: According to the hit vector in the counting approach, this vector accumulates the number of fulfilled predicates per subscription. 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, the hit vector requires $|s|$ bytes of memory.

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. In the subscription matching step, only those candidate subscriptions having at least $|p_{min}|$ fulfilled predicates (accumulated in hit vector), as stored in this vector, are evaluated. 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. Since the non-canonical approach inherently supports unsubscriptions, we do not distinguish two cases ($\mathrm{mem}_{non-canonical,unsub} = \mathrm{mem}_{non-canonical}$).

$$\mathrm{mem}_{non-canonical} = |s|(w(p)|p| + |p| + 2|op| + w(l) + |p|w(s) + 2) \ . \tag{5}$$

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

5 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 approach. 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.

5.1 Relating the Derived Memory Requirements

In our following comparison, we focus on differing data structures of algorithms, i.e., we neglect the fulfilled predicate vector/predicate bit vector, which is incidentally required by all three algorithms. Thus, we can directly analyze (1) to (5).

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, we solely have to analyze the derivations of (1) to (5) for a comparison of these functions (i.e., a comparison of the memory requirements of filtering algorithms).

The first derivation of (1), the counting algorithm without supporting unsubscriptions, in $|s|$ is

$$\mathrm{mem}'_{counting}(|s|) = 2S_s + w(s)s_p|p| \ . \tag{6}$$

The counting algorithm with an efficient support of unsubscriptions (Equation (2)) leads to the following first derivation

$$\mathrm{mem}'_{counting,unsub}(|s|) = 2S_s + w(s)s_p|p| + w(p)s_p|p| \ . \tag{7}$$

For the cluster algorithm not allowing for unsubscriptions (represented by (3)), we deduce as first derivation

$$\mathrm{mem}'_{cluster}(|s|) = S_s w(s) + s_p|p|w(p) \ . \tag{8}$$

In case of supporting unsubscriptions in the cluster algorithm (Equation (4)), its first derivation is defined as

$$\mathrm{mem}'_{cluster,unsub}(|s|) = S_s w(s) + s_p|p|w(p) + S_s w(c) + s_p|p|w(s) \ . \tag{9}$$

Finally, the non-canonical approach with its inherent support of unsubscriptions (Equation (5)) leads to the following first derivation in $|s|$

$$\text{mem}'_{non-canonical}(|s|) = w(p)|p| + |p| + 2|op| + w(l) + |p|w(s) + 2 \ . \quad (10)$$

To eliminate some of the parameters required in the determined derivations, let us assume the following fixed values: $w(s) = 4$, $w(p) = 4$, $w(l) = 4$ and $w(c) = 4$, i.e., the widths of subscription identifiers, predicate identifiers, subscription locations, and cluster references are 4 bytes each[8]. Furthermore, let us reduce the number of characterizing parameters specifying fixed values by utilizing the relative notions of op^r (operators) and s^r (conjunctive elements per predicate) as introduced in Sect. 3.

We now compare the memory requirements (using the derived gradients) of the canonical algorithms (Equations (6) to (9)) to the memory requirements of the non-canonical approach (Equation (10)). The inequalities shown in the following denote the points 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. That is, the non-canonical approach requires less memory if canonical conversions to DNF create more than the stated number of disjunctively combined elements. In the following, we refer to these points as turning points, since they describe in which cases of S_s non-canonical filtering is worthwhile. To allow for a better overview, we use the notation $S_s(\frac{algorithm}{non-canonical})$ to denote the canonical algorithm actually analyzed and compared to the non-canonical approach.

For the counting algorithm without supporting efficient unsubscriptions (Equation (6)) we derive the following inequality if comparing to the non-canonical approach (Equation (10))

$$S_s(\frac{counting}{non-canonical}) > \frac{|p|(2op^r + 9) + 6}{2 + 4s^r|p|} \ . \quad (11)$$

Supporting unsubscription in the counting approach (Equation (7)) leads to the following inequality

$$S_s(\frac{counting, unsub}{non-canonical}) > \frac{|p|(2op^r + 9) + 6}{2 + 8s^r|p|} \ . \quad (12)$$

The cluster algorithm without efficient unsubscriptions (Equation (8)) compared to the non-canonical approach (Equation (10)) is shown in this inequality

$$S_s(\frac{cluster}{non-canonical}) > \frac{|p|(2op^r + 9) + 6}{4 + 4s^r|p|} \ . \quad (13)$$

Finally, if supporting efficient unsubscriptions in the cluster algorithm (Equation (9)), we can describe the point of interchanging space efficiency as

$$S_s(\frac{cluster, unsub}{non-canonical}) > \frac{|p|(2op^r + 9) + 6}{8 + 8s^r|p|} \ . \quad (14)$$

In the following subsection we illustrate these observations graphically.

[8] These values hold on traditional 32-bit machines when using standard integers and memory pointers.

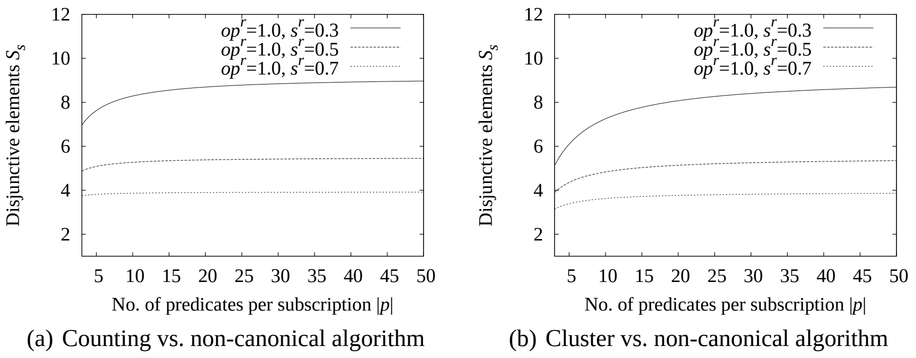


Fig. 4. Theoretically required number of disjunctively combined elements S_s to achieve less memory usage in the non-canonical approach compared to the counting and the cluster algorithm (both without a support of unsubscriptions) using $op^r = 1.0$

5.2 Graphical Illustration of Interchanging Memory Requirements

After the determination of the inequalities denoting the point when the non-canonical approach requires less memory than canonical algorithms (Equations (11) to (14)), we now present this turning point graphically.

Figure 4 shows the point of interchanging memory requirements in case of not supporting efficient unsubscriptions for the counting algorithm (Fig. 4(a)) and the cluster algorithm (Fig. 4(b)). We have chosen $op^r = 1.0$ in Fig. 4, which describes the maximal possible value for this parameter, i.e., the worst case behavior for the non-canonical approach. In the figure, parameter s^r is varied from 0.3 to 0.7. The abscissae of both subfigures show the number of predicates per subscription $|p|$; 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 DNFs to favor the non-canonical approach in respect to memory requirements (cf. (11) and (13), no support of unsubscriptions by both canonical algorithms).

We can realize that the counting algorithm (Fig. 4(a)) requires less memory in cases of small predicate numbers than the cluster algorithm (Fig. 4(b)), i.e., a greater value of S_s is required by the counting algorithm (the DNF consists of more disjunctively combined elements). However, with increasing predicate numbers both algorithms behave nearly the same, i.e., in case of 50 or more predicates per subscription it holds $S_s \approx 4.0$ for both algorithms in case of $s^r = 0.7$. 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.

The behavior of the algorithms with a support of efficient unsubscriptions is shown in Fig. 5. Again, we have chosen $op^r = 1.0$ and varied s^r from 0.3 to 0.7 in the figure. Figure 5(a) presents the behavior of the counting algorithm; the cluster algorithm is illustrated in Fig. 5(b). Compared to Fig. 4, we realize increased memory requirements of canonical algorithms when supporting unsubscriptions due to their requirements to

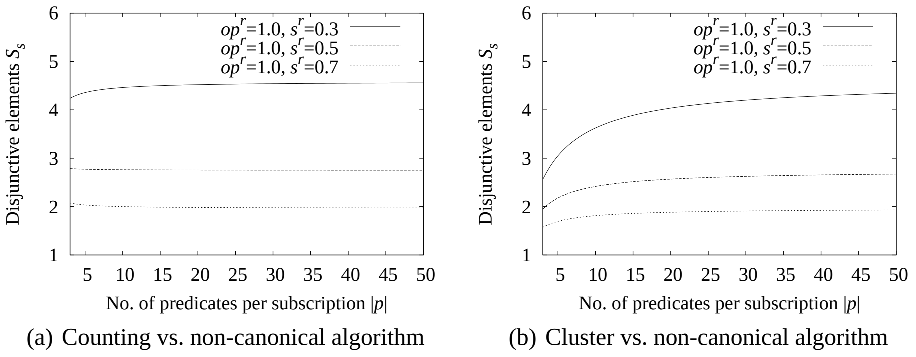


Fig. 5. Theoretically required number of disjunctively combined elements S_s to achieve less memory usage in the non-canonical approach compared to the counting and the cluster algorithm (both with an efficient support of unsubscriptions) using $op^r = 1.0$

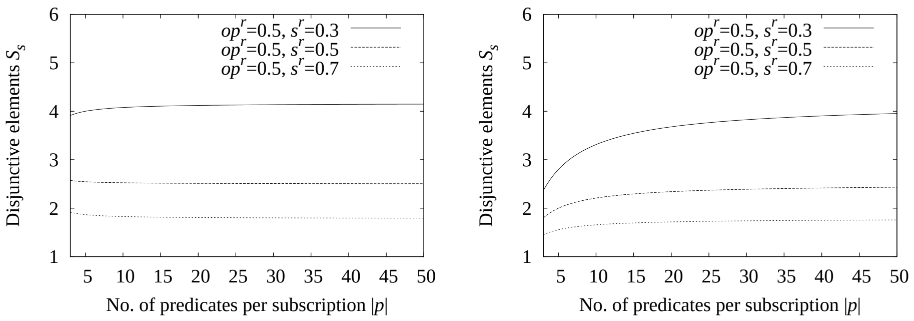
store additional associations between subscriptions and predicates and vice versa, respectively.

Thus, a non-canonical filtering approach is already favorable in cases of smaller numbers of disjunctively combined elements after canonical conversions S_s. In case of $s^r = 0.7$, it holds $S_s \approx 2.0$ (counting approach) and $S_s < 2.0$ (cluster approach). Thus, even if DNFs only consist of approximately 2 disjunctively combined elements, a non-canonical approach requires less memory (in case of $s^r = 0.7$). Such a DNF would be created due to conversions in case of only one disjunction (or more) per original subscription.

In Fig. 4 and Fig. 5, we have chosen $op^r = 1.0$ (relative number of Boolean operators per subscription), describing 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[9]. Hence, a subscription tree containing $|p|$ leaf nodes (i.e., predicates used in the subscription) 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 we can subsume consecutive binary operators to n-ary ones in subscription trees. For example, a purely conjunctive subscription results in $op^r = \frac{1}{|p|}$, since exactly one inner node is required in its corresponding subscription tree.

These observations for the characterizing parameter op^r lead to further improved memory characteristics of the non-canonical approach. Figure 6 shows this behavior using $op^r = 0.5$ with the support of efficient unsubscriptions. Similar the other figures, Fig. 6(a) presents the counting algorithm and Fig. 6(b) the cluster algorithm. Again, even if only one disjunction is used in subscriptions, a non-canonical approach shows less memory usage and better scalability than the counting approach ($s^r = 0.7$). This clearly indicates the advantages of a non-canonical filtering approach in case of subscriptions involving disjunctions.

[9] We can integrate negations into inner nodes and leaf nodes as presented in Sect. 2.2.



(a) Counting vs. non-canonical algorithm (b) Cluster vs. non-canonical algorithm

Fig. 6. Theoretically required number of disjunctively combined elements S_s to achieve less memory usage in the non-canonical approach compared to the counting and the cluster algorithm (both with an efficient support of unsubscriptions) using $op^r = 0.5$

5.3 Considerations in Practice

Our previous analysis shows a comparison of the theoretical memory requirements of the three algorithms. However, their practical implementations require additional space for management data structures, e.g., to link lists together, to store the lengths of variable sized arrays, or to 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 vector, predicate bit vector, and hit vector a static implementation is sufficient due to the requirement of initializing them for each filtered event. A complete overview of the required data structures and their practical realization can be found in Table 2.

For several constantly required data structures, we propose to use dynamic arrays. You can find an overview of these arrays in Fig. 7. These arrays consist of a directory holding pointers to several small fixed-sized arrays. This practical realization allows for our requirements of a dynamically growing data structure in case of registering new subscriptions, involving not much memory overhead for management purposes, and allowing for an efficient access. In our application, indexes in dynamic arrays are always predicate or subscription identifiers. In case of unsubscriptions, freed identifiers are saved and reused in case of registering new subscriptions. Thus, a dynamically shrinking data structure is not required[10].

Predicate subscription association (and subscription predicate association) tables have to satisfy the following requirements: Firstly, they must allow their entries to be ef-

[10] Our approach can be extended to support dynamic shrinking. However, this implies some replacement operations in various data structures used.

Table 2. Practical realization of required data structures

Data Structure	Practical Realisation
Fulfilled predicate vector/predicate bit vector	Simple fixed array
Hit vector	Simple fixed array
Subscription predicate count vector	Dynamic array
Predicate subscription association table	Dynamic table
Subscription predicate association table	Dynamic table
Cluster vector	Dynamic array
References to cluster vector	Pointer
Clusters	List of simple fixed arrays
Subscription cluster table	Dynamic array
Subscription trees	Different sized memory blocks
Subscription location table	Dynamic array
Minimum predicate count vector	Dynamic array

ficiently accessed. Secondly, the tables have to support entries with dynamically changing sizes, i.e., newly registered subscriptions might use existing predicates and unsubscriptions might not remove all predicate subscription associations of certain predicates. Finally, a practical realization should require a reasonable memory overhead.

Our proposal for the predicate subscription association table (similarly for a subscription predicate association table) is depicted in Fig. 8. Actual predicate subscription associations are stored in dynamic arrays (cf. Fig. 7 for their realization) according to their size, i.e., the number of associated subscriptions. These associations consist of the predicate identifier and the subscription identifiers of subscriptions containing the predicate. In case of unsubscriptions, entries are removed and, if associations are left, inserted into the dynamic array storing entries of the corresponding size. Removed entries are replaced by the last element inserted in an array to retain fully filled arrays. Access to these dynamic arrays is provided by an additional array holding references to them.

We apply another dynamic array storing references to actual associations. Indexes for this array are predicate identifiers. Thus, we can easily access the association of a certain predicate. In case of moving associations due to unsubscriptions, we replace the current reference. This is allowed by the fact of storing the predicate identifier in associations.

A comparison of these data structures to the memory requirements of standard implementations, i.e., STL hash (multi) sets, has resulted in much less space usage for our specialized implementations. Thus, our implementations are well suited data structures for the filtering process.

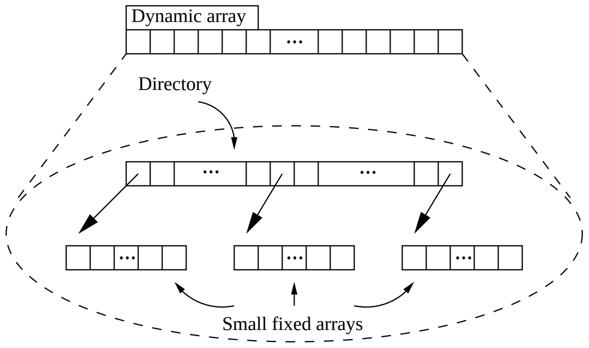


Fig. 7. Realization of dynamic arrays

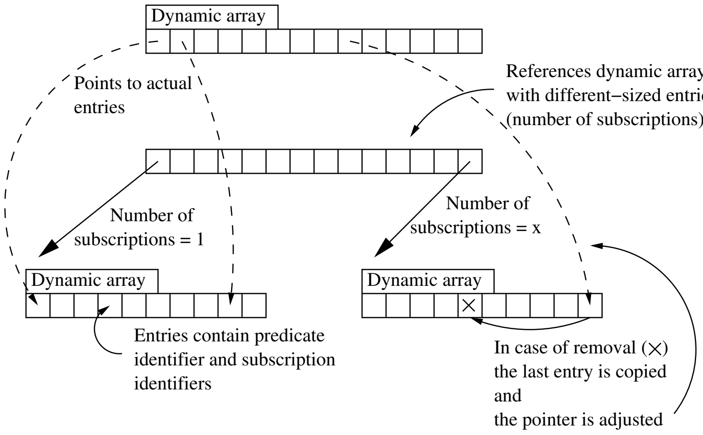

Dynamic array
Points to actual entries
References dynamic arrays with different−sized entries (number of subscriptions)
Number of subscriptions = 1
Number of subscriptions = x
Dynamic array
Dynamic array
Entries contain predicate identifier and subscription identifiers
In case of removal (×) the last entry is copied and the pointer is adjusted

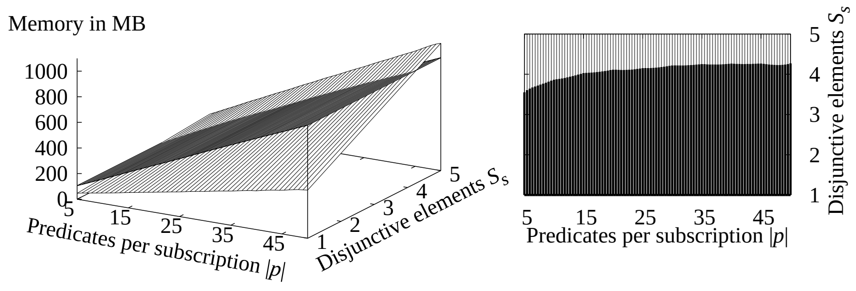
Fig. 8. Realization of dynamic tables (in case of the predicate subscription association table)

After this description of our practical realization of the required data structures, we present our practical analysis utilizing these implementations in the next section.

6 Practical Analysis of Memory Requirements and Efficiency

In Sect. 4 and Sect. 5, 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. 6.1). Furthermore, we present a brief comparison of the efficiency characteristics of the compared algorithms (Sect. 6.2).

We compare the non-canonical approach to one canonical algorithm by experiment. Because of the restrictions of the cluster approach (cf. Sect. 4.2), we have chosen the counting algorithm as representative of canonical algorithms for our practical analysis.



(a) Perspective view (b) Top view

Fig. 9. Memory requirements in our practical experiments in case of $s^r = 0.3$, $op^r = 0.5$ and $|s| = 1,000,000$ (cf. Fig. 6(a) for theoretical results in the same scenario)

This allows the generalization of our results to other settings than equality predicate-based application areas and areas dealing with less predicate redundancy than assumed by the cluster approach. Furthermore, the counting algorithm is more space efficient than the cluster approach (cf. Fig. 4, Fig. 5, and Fig. 6). In a practical implementation, the cluster approach without efficiently supported unsubscriptions and the counting approach show nearly the same memory requirements [6].

6.1 Practical Analysis of Memory Requirements

In this section, we compare the memory requirements of the counting algorithm and the non-canonical approach. Our implementations of these algorithms follow our descriptions in Sect. 5.3.

In our experiments, we aim at verifying our results shown in Fig. 6(a), i.e., in case of $op^r = 0.5$. Here we present the memory usage of the required data structures[11] 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 9(a) shows both algorithms in case of $s^r = 0.3$; Fig. 10(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, and 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 ap-

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

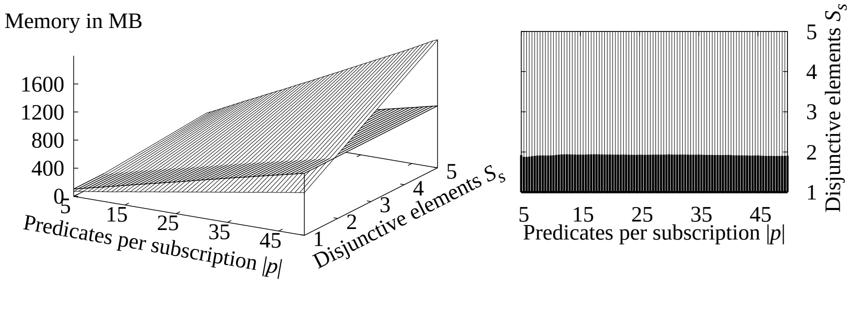


Fig. 10. Memory requirements in our practical experiments in case of $s^r = 0.7$, $op^r = 0.5$ and $|s| = 1,000,000$ (cf. Fig. 6(a) for theoretical results in the same scenario)

proach. As expected (and shown in our theoretical analysis) the non-canonical approach does not change its memory usage with growing numbers of disjunctively combined elements after conversion 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 in case of $|p| = 50$. This holds for both figures, Fig. 9(a) and Fig. 10(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 numbers of disjunctively combined elements after conversion S_s as showed in (2). Furthermore, according to (2) increasing s_p (and thus increasing s^r) results in advanced space usage, e.g., approx. 700 MB of memory in case of $|p| = 50$, $S_s = 3$ and $s^r = 0.3$ compared to approx. 1200 MB of memory in case of $|p| = 50$, $S_s = 3$ and $s^r = 0.7$ for the counting approach.

As depicted in our theoretical comparison in Fig. 6(a), there exists a point of interchanging memory requirements of canonical and non-canonical algorithms. This turning point is denoted by a cutting of the surfaces of the two algorithms. In Fig. 9(a) this cutting occurs at $S_s \approx 4$, and in Fig. 10(a) it happens at $S_s \approx 2$. To exactly determine the point of cutting surfaces, we present a top view of the three-dimensional diagrams in Fig. 9(b) and Fig. 10(b), respectively. Figure 9(b) shows that we can find the turning point between $S_s = 3$ and $S_s = 5$ for $s^r = 0.3$ dependent on the number of predicates per subscription $|p|$. For $s^r = 0.7$ (Fig. 10(b)), it is always located slightly below $S_s = 2$.

Comparing these practical results to our theoretical results in Fig. 6(a), we realize that our theoretical analysis has predicted nearly the same behavior of the two algorithms: Even if only two ($s^r = 0.7$) or four ($s^r = 0.3$) 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: The usage of only one disjunction in subscriptions leads to a beneficial behavior of the non-canonical approach.

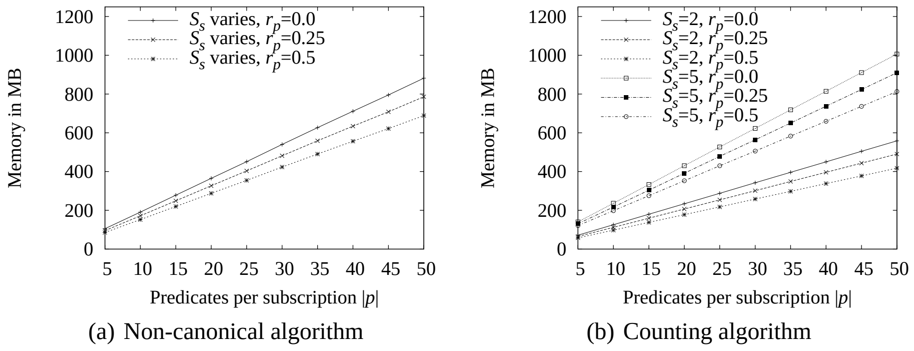


Fig. 11. Influence of predicate redundancy r_p on the algorithms in case of $|s| = 1,000,000$, $op^r = 0.5$ and $s^r = 0.3$

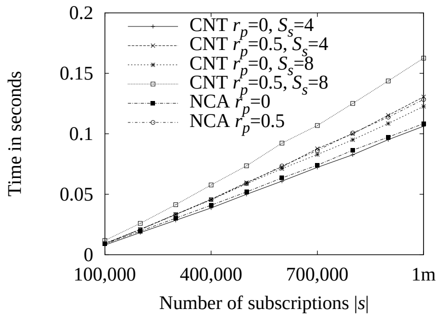
Practical Analysis of Influences of Redundancy In our theoretical analysis we have shown that predicate redundancy r_p does not influence the memory requirements of algorithms. However, in a practical realization this property does not hold.

The influence of predicate redundancy r_p on our implementation is illustrated in Fig. 11. The 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. 11(a), and the counting algorithm is presented in Fig. 11(b) for different numbers of disjunctively combined elements after conversion S_s and varying predicate redundancy r_p. In our figures, we diversify r_p between 0.0 and 0.5.

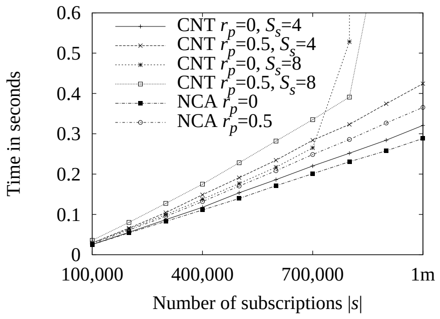
Both algorithms show decreasing memory requirements with increasing r_p. This behavior results out of the decreasing memory overhead in a practical implementation: Both algorithms utilize a predicate subscription association table, which requires a dynamic implementation causing more memory usage. Generally, for each registered unique predicate the number of subscriptions and a pointer has to be stored (cf. Sect. 5.3). If there are less unique predicates, which is caused by predicate redundancy, the amount of memory overhead decreases. Thus, the total memory requirements of both algorithms decrease as observable in Fig. 11.

6.2 Practical Analysis of Efficiency

We are aware of the correlation between memory usage and filter efficiency of filtering algorithms: The most space efficient algorithm cannot be utilized in practice if it shows poor time efficiency. Vice versa, time efficient solutions, e.g., [7], might become inapplicable in practice 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 the predicate matching step is the same in both algorithms. Time efficiency is



(a) $|p| = 10$



(b) $|p| = 30$

Fig. 12. Influence of the number of subscriptions $|s|$ on the counting algorithm (CNT) and the non-canonical approach (NCA) for varying predicate numbers $|p|$, predicate redundancy r_p, and disjunctively combined elements after conversion S_s

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

Figure 12 shows the influence of the number of subscriptions registered with the pub/sub system. In Fig. 12(a), we used 10 predicates per subscription ($|p| = 10$), Fig. 12(b) illustrates time efficiency in case of 30 predicates per subscription ($|p| = 30$). 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 $r_p = 0.5$, respectively. We also show the non-canonical approach assuming the worst case behavior, i.e., if a candidate subscription is evaluated, its whole Boolean expression is analyzed. Thus, always entire subscription trees are tested in our experiments. In this experiment we have increased the number of fulfilled predicates per event p_e with growing subscription numbers: $p_e = \frac{|s||p|}{50}$, i.e., $p_e = 20,000 \ldots 200,000$ and $p_e = 60,000 \ldots 600,000$ in Fig. 12(a) and Fig. 12(b), respectively. 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 12 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 $|p| = 30$ (cp. Fig. 12(b)), the counting algorithm requires more memory than available resources (resulting in 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. 6.1). Generally, increasing predicate redundancy r_p leads to growing filtering times for both algorithms in the evaluated setting. This is due to the fact of more candidate subscriptions required to be evaluated (non-canonical algorithm) and more counters to be increased in the hit vector (both algorithms). The counting algorithm in case of $S_s = 8$ shows always the worst time efficiency. According to the number of predicates $|p|$ either the non-canonical ap-

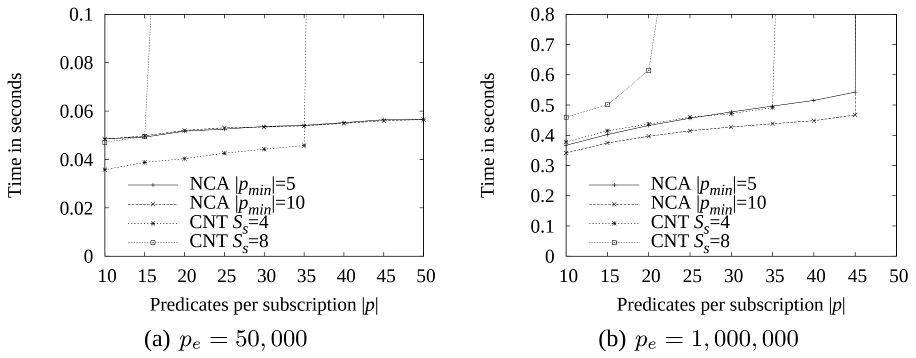


Fig. 13. Influence of the number of predicates $|p|$ on the counting algorithm (CNT) and the non-canonical approach (NCA) for varying numbers of fulfilled predicates per event p_e, disjunctively combined elements after conversion S_s, and fulfilled predicates required for matching $|p_{min}|$

proach (Fig. 12(b)) or the counting algorithm with $S_s = 4$ (Fig. 12(a)) are the most efficient filtering approaches (nearly on par with the respective other approach).

The influence of the number of predicates per subscription $|p|$ is shown in Fig. 13. We present two different settings: In Fig. 13(a), it holds $p_e = 50,000$ (fulfilled predicates per event); Fig. 13(b) shows the case of $p_e = 1,000,000$. For the non-canonical approach we analyzed two different settings: The minimum number of fulfilled predicates required for matching $|p_{min}|$ is chosen with 5 and 10, respectively. The counting approach is presented in two variants with $S_s = 4$ and $S_s = 8$. In this experiment, $1,000,000$ subscriptions are registered.

Again, sharp bends in the curves in Fig. 13 denote the point of exhausted main memory resources. The non-canonical approach shows the best scalability and is 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 a higher number of minimally required fulfilled predicates $|p_{min}|$. This effect becomes more apparent with a high value of p_e (Fig. 13(b)) due to the fact that more candidate subscriptions require evaluation. In case of a small number of fulfilled predicates per event p_e, the counting approach (in case of $S_s = 4$) is more efficient than the non-canonical approach; large numbers of p_e clearly favor the non-canonical approach. The reason for this behavior is the increased number of hits (incrementing the hit vector) in the counting approach due to canonical conversions.

Our efficiency analysis shows that the counting and the non-canonical approach perform similarly in case of increasing problem sizes (number of subscriptions and number of predicates). In some cases, the counting approach shows slightly better time efficiency. Other settings favor the non-canonical approach. In case of large values of 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. The measured values regarding the non-canonical approach differ from [3] due to our variations to the setting and the algorithm (cf. Sect. 2.2).

7 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 approach [2, 14], cluster approach [6, 8], and non-canonical approach [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: Only one disjunction in subscriptions might result in less memory requirements for non-canonical than for canonical solutions.

To show the applicability of our theoretical results in a practical implementation, we proposed an implementation for the required data structures to investigate 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., one disjunction per subscription is used), 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. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–61, Atlanta, USA, May 4–6 1999.
2. 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)*, pages 539–548, Vienna, Austria, July 2–5 2002.
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 Interna-*

tional Conference on Software Engineering (ICSE 2001), pages 443–452, Toronto, Canada, May 12–19 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)*, pages 163–174, Karlsruhe, Germany, March 24–26 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 International Conference on Management of Data (SIGMOD 2001)*, 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 (ACSC-18)*, Adelaide, Australia, February 1–3 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)*, pages 271–280, Atlantic City, USA, May 23–25 1990.
9. A. Hinze. *A-MEDIAS: Concept and Design of an Adaptive Integrating Event Notification Service*. PhD thesis, Freie Universität Berlin, Institute of Computer Science, July 2003.
10. G. Mühl and L. Fiege. Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.
11. G. Mühl, L. Fiege, and A. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS '02)*, pages 224–238, Karlsruhe, Germany, April 8–12 2002.
12. 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.
13. 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.
14. T. W. Yan and H. García-Molina. Index Structures for Selective Dissemination of Information Under the Boolean Model. *ACM Transactions on Database Systems (TODS)*, 19(2):332–364, 1994.