An efficient matching solution for publish and subscribe systems

João Dias Pereira, Helena Galhardas, Andreas Wichert
INESC ID Lisboa, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{jooao,hig,Andreas.Wichert}@inesc-id.pt

This revision: December 23, 2009
An efficient matching solution for publish and subscribe systems

João Dias Pereira, Helena Galhardas and Andreas Wichert

INESC ID Lisboa, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{j, hig, Andreas.Wichert}@inesc-id.pt

Abstract. Publish and Subscribe is the paradigm in which users express long-term interests (“subscriptions”) and some external agent (perhaps other users) “publishes” events (e.g., offers). The job of Publish and Subscribe (P/S) software is to send events to the owners of subscriptions satisfied by those events. Two important requirements that must be met by P/S systems deployed in a Web environment are scalability and expressiveness. P/S systems face a scale problem due to the sheer number of users and quantity of dynamic information that they may handle. At the same time, these systems must be expressive so that subscribers are able to specify exactly their interests. The gain in the expressiveness of P/S systems results in an increase in the complexity of the matching process: the more sophisticated the constructs available to define subscriptions, the more complex the matching process. This paper presents three very efficient matching algorithms that support a high rate of events against a large number of subscriptions consisting of conjunctions of (attribute, comparison operator, value) predicates.

Keywords: Publish and Subscribe system, matching algorithm, event, subscription

1 Introduction

Much of human information is now available on the Web. The Web is particularly well-suited to time-varying information (e.g. Google or Yahoo are a better source of current world events than newspapers). For this reason, and as pointed out in [Bernstein et al., 1998], there is a need for systems to capture this changing information by notifying users of interesting events. The rapid growth of the amount of available time-varying (i.e., dynamic) information and of the number of users interested in that kind of information has led to an explosion in the number and variety of new data dissemination applications. We can consider that there are two types of entities in this kind of application: the entities that produce the information to be disseminated, called the producers of information, and the entities that are interested in the information, called the consumers of information. The main goal of such applications is to deliver the right dynamic information at the right time to the right consumers, i.e., to timely forward the new information to those consumers that are interested in receiving it.

There are two distinct approaches to disseminate information to interested consumers: the pull-way or the push-way. In a pull-based information dissemination approach, consumers control the dissemination of information. They query the producers in order to get new interesting information. In a push-based information dissemination approach, the producers control the dissemination of information. Push technology stems from a very simple idea. Instead of requiring consumers to explicitly ask for the information they need (i.e., use the pull technology), the producers of information can directly send the desired information to the interested consumers without having them to explicitly require it.

Publish and subscribe (P/S) systems uses the push-way approach to disseminate the information. In a publish and subscribe context, information is usually designated as events. An event constitutes a state transition of interest in the context of an application. Examples include updating columns in a database table, a football team scoring a goal, a stock reaching a specified value, a network link going down, and a mobile user moving around (that is, a change in its (x,y) location coordinates) or reaching a specific cell in the network. An example of a P/S system can be a system that supports a stock trading application written for a particular stock exchange, say the New York Stock Exchange (NYSE). In this application, all the dynamic information produced by the NYSE, e.g., stock trades, bids or sales, are published as events by the stock trading application. Brokers affiliated with the NYSE subscribe to interesting trades defining the required subscriptions for this stock trading application. Events can be represented as sets of attribute-value
pairs. This format is very common in P/S systems mainly due to its simplicity and flexibility in representing a large range of different types of information. In this example, we would have at least the following event attributes: the NYSE ticker symbol, the share price, the share volume and the broker id.

P/S systems effectively decouple the entities that generate events from those that receive them [Eugster et al., 2001]. Thus, event producers and consumers may be fully unknown to each other. The benefits of this decoupling has allowed the implementation of large distributed systems that can evolve constantly (i.e., the system participants can come up and go down periodically and new participants can be added or removed to/from the system at any instant). This paradigm has found wide-spread applications, ranging from application integration [Julienne and Holtz, 1994, Hall et al., 1997, ibi, Eugster et al., 2000], network and distributed system management [Mansouri-Samani and Sloman, 1997, Wolfson et al., 1991], and sensor networks [Souto et al., 2005, Hunkeler et al., 2008] to wide area network-based selective information dissemination systems [Altnel and Franklin, 2000, Yan and Garcia-Molina, 2000, Gruber et al., 1999, Terpstra et al., 2003, Baldoni et al., 2005, Baldoni et al., 2007].

Web Information dissemination applications have usually the following characteristics: (i) a very large number of users potentially interested in receiving the information (several millions); (ii) there is an overlap among the interests of the user population; (iii) users are mainly interested in new data or changes to existing data; (iv) the amount of data that must be sent to interested users is usually small; (v) the potential production rate of new data can be very high; and (vi) a high volatility of web user demands (new subscriptions, new users and cancellations)\(^1\).

Two important requirements that must be met by P/S systems deployed in a Web environment are scalability and expressiveness. P/S systems face a scale problem due to the sheer number of users and the quantity of dynamic information that they may handle. The matching algorithm applied by the P/S system must be able to match a high rate of events against a large number of subscriptions. Moreover, it should have a low maintenance cost in order to support the high volatility of web user demands. At the same time, these systems must be expressive so that subscribers are able to specify exactly their interests. In fact, subscribers should only receive events they are really interested in. However, a gain in the expressiveness of P/S systems results in an increase in the complexity of the matching process: the more sophisticated the constructs available to define subscriptions, the complexer the matching process. This complexity combined with a large number of subscriptions may severely degrade the matching efficiency. So, systems devoted to support a large number of subscriptions and a high rate of events have to face a trade-off between the subscription language’s sophistication and the matching efficiency. Designing a subscription language that is sufficiently expressive together with matching algorithms for efficient event processing is a major challenge for such systems.

A lot of matching algorithms have been proposed in the literature in the last years [Hanson et al., 1990, Aguilera et al., 1999, Gough and Smith, 1995, Yan and Garcia-Molina, 2000, Pereira et al., 2000, Fabret et al., 2001, Piskiel et al., 1999, Pereira et al., 2001, Ashayer et al., 2002, Segal and Arnold, 1997, Campailla et al., 2001, Bittner and Hinze, 2007]. Most of these algorithms ([Hanson et al., 1990, Aguilera et al., 1999, Gough and Smith, 1995, Yan and Garcia-Molina, 2000, Pereira et al., 2000, Fabret et al., 2001, Piskiel et al., 1999, Ashayer et al., 2002]) support a simple subscription language where the interests of the users are specified as a conjunction of conditions of event attributes. Some support a more expressive language, like [Altnel and Franklin, 2000, Pereira et al., 2001], where events and subscriptions are specified as XML documents and XPATH expressions, respectively, or [Segal and Arnold, 1997, Campailla et al., 2001, Bittner and Hinze, 2007], where subscriptions can be specified as arbitrary boolean expressions involving the event attributes. Some work has also been made to support the specification of subscriptions that are matched by composite events, i.e. a subscription is matched by a certain temporal combination of events [Courtenage, 2002, Heinze, 2003, Wu et al., 2006, Zhao et al., 2008].

In this paper, we present three very efficient matching algorithms with different characteristics that support a subscription language with a sufficient level of expressiveness [Pereira et al., 2000, Fabret et al., 2001]. These main memory algorithms were designed to optimize the number of processor cache misses which is an important issue in main memory algorithms.

\(^1\) For example, a user may want to go from New York to California in the next 24 hours but only if he can get a flight for under $400. Such a subscription would be short-lived.
1.1 Performance issues in main memory algorithms

With the emergence of cheap computers having very large random access memory, more and more algorithms will run in main memory without any access to secondary memory [Rao and Ross, 1999]. However, PC processors still have small cache memories: Processor cache memories are static RAM memories which hold data that were recently referenced by running programs. Inside a cache memory, memory references can be processed at processor speed. References that are not found in the cache, called misses, require the fetch of the corresponding cache block from the main memory at a much higher cost (tens of CPU cycles). When a cache miss occurs the processor is (normally) idle until the fetch is performed. So cache misses severely impede program performance. For this reason, main memory algorithm performance is not only sensitive to the number of instructions they perform, but also to cache behavior. Moreover, the main trends are: (1) RAM size and processor speed grow exponentially within the next years; (2) Processor cache size does not increase more than linearly. Thus, main memory algorithms will become more and more sensitive to processor cache behavior.

Processor cache management policies are very simple (for evident processing cost reasons). However, modern processors provide now the PREFETCH command that permits a running program to force the fetch of a cache block from a specified position in the RAM. This command is executed in parallel with program instructions. Thus, if the program can predict in advance which cache block it will need to read, it can avoid a cache miss by prefetching the cache block a few instructions before. Another way to limit cache misses is to design algorithms that are aware of temporal and spatial locality. Spatial locality is achieved when data that are used consecutively by the algorithm are placed in consecutive memory addresses. Temporal locality is achieved when the same data is manipulated in consecutive instructions.

The matching solutions that we present in this paper were specifically designed to be cache conscious. To our knowledge, these were the first matching solutions that are aware of the cache behavior.

1.2 Organization of the paper

This paper is organized as follows. Section 2 presents the matching problem. Section 3 gives a general description of our matching algorithms. Section 4 describes the matching algorithm that clusters the subscriptions using a single-dimensional index. Section 5 presents our solution to index the subscriptions using a multi-dimensional index. Section 6 presents our cost-based approach to compute optimal clustering. Section 7 presents an adaptive algorithm to deal with changes in subscription and event patterns. Section 8 presents performance studies. Finally, Section 9 concludes.

2 The Matching Problem

An important class of subscription language is called conjunctive subscription language with elementary expressions (CSLWE). In CSLWE, subscriptions are specified as conjunctions of predicates and each predicate defines a condition over a single attribute. The CSLWE class supports the minimal expressiveness that should be offered by content-based P/S systems. This means that subscribers can express their interests with a minimal level of exactness. In fact, most content-based P/S systems provide a subscription language that belongs to the CSLWE class or to a more expressive class.

In this context, the matching problem can be stated as follows: given an event $e$, specified in the attribute-value pair format, and a set $S$ of subscriptions, defined using a CSLWE, what are the subscriptions of $S$ that are matched by $e$? As we have seen before, the matching problem is very important because it has a strong impact on the scalability of the P/S system. We designate a solution to the matching problem as the matching algorithm.

There are two main approaches in the literature to solve the matching problem. The first one consists of individually matching each subscription against the incoming event to compute the matched subscriptions. We call this approach the scan approach. For a small number of subscriptions, the sequential scanning of subscriptions is acceptable. However, as the number of subscriptions increases, sequential scanning becomes prohibitively slow due to its linear scale-up. The second approach, called the index approach, uses indexing techniques to facilitate the computation of the matched subscriptions. In the index approach,
the subscriptions are placed into an indexing structure which is accessed using the attribute values of the event being processed.

The role of indexing in query optimization is well-understood in the database community. Indexes, or access methods, enable efficient access to a subset of the indexed data. There have been many studies about indexing in the context of relational database management systems (RDBMS) and the most popular indexing techniques are based on B-trees [Bayer and McCreight, 1972]. However, these indexing structures are not well-suited to solve the matching problem. Thus, P/S systems that have followed the index approach have defined their own indexing structures.

For instance, the matching problem is similar to the classical point query problem that arises in spatial databases. Spatial databases, which are databases whose stored data are regions or points in a multi-dimensional space, include typical search operations, like the point query, i.e., find all objects of the database that contain a given search point, and the range or region query, i.e., find all objects that overlap a given search region. A large number of multidimensional spatial access methods have primarily been designed to support such operations. The matching problem is similar to a point query. In fact, an event can be seen as the search point and subscriptions can be treated as regions in a \( k \)-dimensional space, where \( k \) is equal to the total number of attributes that can be referred to in events. For example, the following subscription \( s: \{(x = 2), (y \leq 6), (y \geq 4), (z \leq 10)\} \) and the event \( e: \{(x,2), (y,5), (z,7)\} \) can be represented as the region \((2,2), [4,6), [−∞, 10]\) in the \( xyz \) space, respectively. The matching problem is then analogous to the point query problem that consists of determining the regions, represented by the subscriptions, that contain the query point, defined by the matching event.

A large number of multi-dimensional indexing solutions, such as the R-tree [Guttman, 1984] the \( R^+ \)-tree [Sellis et al., 1987] and the \( R^* \)-tree [Beckmann et al., 1990] (see [Gaede and Günther, 1998] for a survey), have been proposed as spatial access methods in the literature, to solve the point query problem. These solutions could also be applied to solve the matching problem. However, the context of the matching problem in P/S systems makes these solutions impracticable in most cases.

The number of attributes that can be referred in a subscription or in an event can be very high. This is due to the fact that information is not usually described using only a few attributes but it can involve several tens, or even hundreds, of attributes. This means that the size of the global schema of the P/S system (i.e., the number of existing attributes) can be very large. Therefore, subscriptions and events can be high-dimensional data. We expect subscriptions to have a small number of predicates (between two and five predicates). In what concerns events, we expect that events will refer to most of the existing attributes in order to define the information they describe as precisely as possible. Therefore, the average size of attribute-value pairs in events is large. It has been observed that the performance of multi-dimensional indexes can deteriorate quickly with the increase of data dimensionality. This phenomenon is known as the curse of dimensionality. The deterioration can be so severe that the performance of the index is worse than the use of the scan approach. For example, the R-tree access method is prohibitively slow for dimensions higher than five.

3 Our approach to matching events

We present three matching solutions designed for the Web environment. These three matching solutions first cluster together the subscriptions according to some common predicates. Our algorithms are three-step algorithms. The first step computes the predicates satisfied by the event. Then, the second step computes the subscription clusters that may contain subscriptions verified by the event using the set of satisfied predicates determined in the first step. The third step accesses those subscription clusters and determines the subscriptions verified by the event. Depending on the matching solution, the subscription index can be single or multi-dimensional and its maintenance can be static or dynamic with both the subscription and event distributions.

This section first presents the event and subscription model supported by our matching solutions. Then, it describes the common data structures used by our matching algorithms to index the subscriptions and support the matching processing. Finally, it also presents the generic approach followed by our matching algorithms to compute the subscriptions matched by an event. The specific characteristics of each proposed matching algorithm are described in the next sections.
3.1 Event and subscription model

The event model we consider is as follows. Consider a finite set of attributes \( A = \{ att_1, att_2, ..., att_k \} \), designated as the global schema of the P/S system, where each attribute \( att_i \) is associated with a certain value domain \( D_i \). An event is a conjunction of pairs. Each pair consists of an attribute and a value, where the value belongs to the domain of the attribute. There are no two pairs in the same event referring to the same attribute. For example, the event \( e_1 \) that represents that the movie “Groundhog Day” can be seen at theater Odeon for a price of 8 dollars is defined as follows: \{ (movie, "Groundhog Day"), (price, 8$), (theater, "Odeon") \}. The set of attributes referred to in an event is called the event schema of the event. For example, the event schema of \( e_1 \) is formed by the attributes movie, price and theater. An event does not need to refer all attributes of the global schema of P/S system, i.e., the schema of an event can be a subset of the global schema.

A subscription \( s \) is a conjunction of predicates, each of them is a triple consisting of an attribute, a comparison operator \( (\leq, =, \geq) \), and a value. We define predicates that use the \( \leq \) or \( \geq \) comparison operators as non-equality predicates. An event pair \((a', v')\) matches a subscription predicate \((a, \text{comp}, v)\) if \( a = a' \) and \( \text{comp} v \) is true. For example, \( (\text{price}, 8\$, \leq) \) matches \( (\text{price}, 8\$, \leq) \) because they share the same attribute and \( 8$ \leq 10$. An event \( e \) satisfies a subscription \( s \) if every predicate in \( s \) is matched by some pair in \( e \). For example, the above event \( e_1 \) satisfies the following subscription \( s_1 \) \{ (movie, =, "Groundhog Day"), (price, \leq, 10$), (price, \geq, 5$) \}. The schema of a subscription is composed of the set of attributes referred to in the predicates of the subscription. The schema of a subscription can be a sub-schema of the global schema. For instance, the subscription schema of \( s_1 \) is given by the following set: \{ movie, price \}.

The set of comparison operators that we consider seems to be rather restrictive in expressiveness. However, we would like to point out that matching algorithms that support these comparison operators can usually be easily extended to support other types of operators, like the \( \neq \), for instance.

3.2 Data structures

The data structures used by our matching solution are the following: a set of indexes over the predicates; a predicate vector; a table containing the predicates of each subscription, called the subscription predicate table; a special index over the subscription cluster lists; a vector of references to subscription cluster lists, named cluster vector; and a set of subscription clusters (where the subscriptions are placed). Figure 1 shows these data structures.

The algorithm uses the set of indexes over the predicates to efficiently compute the set of predicates satisfied by an event. The indexes over the predicates are deployed as follows. The predicates are logically grouped by predicate family putting together all predicates having the same attribute and the same comparison operator. For example, consider the following three subscriptions \( s_1 = \{ (\text{price} = 20), (\text{quantity} \geq 10) \} \), \( s_2 = \{ (\text{price} \leq 15), (\text{quantity} = 10) \} \), and \( s_3 = \{ (\text{price} \leq 12) \} \). There are four predicate families: \( (\text{price} =), (\text{price} \leq), (\text{quantity} \geq), \) and \( (\text{quantity} =) \). Fast access to the predicates belonging to a given family is achieved by using an index where the key values are the values used in the predicates belonging to the family. The nature of the indexes depends on the family type. For example, hashing provides fast access to equality predicates, but it is not feasible for non-equality predicates.

The set of predicates of each subscription is kept at the subscription predicate table. This table contains an entry for each subscription. Each of these entries stores the set of predicates of the corresponding subscription. Each subscription is associated with a unique identifier. An indexed predicate that occurs in one or more subscriptions is associated with a single entry in the predicate vector and is represented by a unique identifier. In Figure 1, \( P_0, ..., P_6 \) represent unique predicate identifiers. A predicate identifier is used as an index to access the entry associated with the predicate in the predicate vector. Each entry in the predicate vector represents the result of the predicate evaluation. It is set to one if the corresponding predicate is satisfied by the event, and zero otherwise.

Our matching solution factorizes the subscriptions using (some of) their common predicates. Subscriptions are clustered together according to a certain set of predicates that are common to all subscriptions in the same cluster. Moreover, all subscriptions within a cluster have the same size (i.e., the same number of predicates). Given a subscription cluster \( C \) and the set of predicates used to cluster the subscriptions of \( C \), we designate this set of predicates as the access predicate of \( C \) (and consequently of every subscription of
Fig. 1. The data structures used in our event matching solution. This figure also represents how three subscriptions $S_0$ (composed by predicates $P_4, P_2, P_1$ and $P_0$), $S_1$ (composed by predicates $P_4, P_2, P_1$ and $P_3$) and $S_2$ (composed by predicates $P_4, P_5$ and $P_6$) are stored in subscription clusters considering that they are clustered using a single predicate.

$C_i$. Two clusters can have the same access predicate, but differ on the number of predicates of their subscriptions. Clusters that have the same access predicate are put together into the same subscription cluster list.

Each cluster consists of a certain number of integer arrays that store the identifiers of the predicates of the subscriptions placed in the cluster that do not belong to their access predicate. The identifiers of those subscriptions are equally stored in the cluster. The information concerning each subscription (predicates and subscription identifier) is stored column-wise in the arrays of a cluster, i.e., the entries that correspond to position $l$ of each array of a cluster refer to the same subscription. The predicates of a subscription are stored in the arrays of a cluster according to their selectivity. For example, consider a subscription cluster $c$ with $n$ arrays. The most selective predicate of the subscriptions kept in $c$ is stored in the first array, while the least selective one is stored in $(n − 1)^{th}$ array of $c$. The last array of a cluster keeps the identifiers of the subscriptions placed in the cluster. This organization optimizes the average number of predicates per subscription that are accessed when checking the subscriptions of a cluster.

We would like to point out that an access predicate $ap$ is verified by an event if and only if every predicate of $ap$ is also verified by the event. This guarantees that subscriptions belonging to the subscription cluster list associated with an access predicate $ap$ only need to be checked if and only if $ap$ is satisfied. Therefore, our matching solution only needs to access those subscriptions placed in subscription groups whose access predicates are satisfied. To efficiently compute the access predicates satisfied by an event, we need to build an index over the existing access predicates that returns the corresponding satisfied access predicates given the set of satisfied predicates. The content of this index (i.e., the access predicates indexed in it) depends essentially on how the subscriptions are clustered, i.e., how the access predicates are defined.

Finally, the cluster vector maintains the association between each existing access predicate and the list of subscription clusters that share this access predicate. Each access predicate is associated with a unique identifier. This identifier represents the position in the cluster vector associated with the access predicate.

3.2.1 Space cost

The space cost of our matching solution is linear in the number of predicates. First, the size of the predicate vector is equal to the number of distinct predicates. Second, each subscription is stored into a single
cluster that also contains the identifiers of all subscription predicates (excluding those present in the access predicate). Thus, the total size of the subscription clusters is linear in the total number of subscription predicates.

Third, the cost of storing the predicates of each subscription in the subscription predicate table is also linear with the total number of subscription predicates. In what concerns the space required for the data structures used to access the subscription clusters (i.e., the index over the access predicates and the cluster vector), our solution of indexing the subscription cluster lists is linear in the number of distinct access predicates, which is at most equal to the number of subscriptions. This is the worst case, where each subscription cluster contains a single subscription.

Finally, an additional space is required for indexing the predicates in the predicate families. The type of indexes used by our solution for the predicates families, that are described in detail in [Pereira, 2002], guarantees a space cost for the predicate indexes that is linear in the number of distinct predicates.

3.3 The generic event matching algorithm

The event matching algorithm is executed each time a new event is processed, and it computes the subscriptions matched by the event. The generic event matching algorithm, shown in Figure 2, consists of three steps. The first step initializes the predicate vector, setting all entries to zero; computes the predicates satisfied by the incoming event, using the predicate matching algorithm, described in Section 3.3.1 that is represented by the compute_predicates function in Figure 2; and sets to one all the entries of the predicate vector that correspond to satisfied predicates. The second step computes the subscription clusters that must be accessed by the algorithm, i.e., determines those subscription clusters whose access predicates are verified by the event. We call these clusters the candidate clusters. This step is performed by the cluster access algorithm, described in Section 3.3.2, and is represented by the get_candidates function in Figure 2. Finally, the last step of the matching algorithm applies the cluster checking algorithm, shown in Section 3.3.3, to each candidate cluster in order to compute the subscriptions of each cluster matched by the event.

3.3.1 The predicate matching algorithm

The predicate matching algorithm, that implements the compute_predicates function in Figure 2, is represented in Figure 3. Given an input event \( e \), it computes the set of predicates satisfied by \( e \). This algorithm is very simple. For each attribute-value pair \( (att, val) \) present in \( e \), the algorithm accesses all predicate families that concern the attribute \( att \) (lines 8-10). For each of those predicate families, the algorithm searches for the predicates satisfied by the value \( val \) (line 9). This search is implemented by the get_satisfied_predicates function which depends on the type of predicate family: For instance, for the equality family type we can use hash tables, while for the less-than and greater-than family types we can order the predicates for each family and then use a binary search to compute the predicates satisfied by the event.

Finally, the algorithm returns all predicates verified by the event (line 12).

3.3.2 The generic cluster access algorithm

The cluster access algorithm that implements the get_candidates function in Figure 2 is the key point of our matching solutions. It is responsible for determining the subscription clusters that need to be accessed, i.e., the clusters whose access predicates are satisfied by the event being matched. It depends on the data indexing structure used to access the clusters.

Our matching solutions apply a subscription clustering strategy that consists in partitioning the subscriptions taking into account their predicates. Moreover, each subscription is placed into a single cluster. The clustering strategy depends on three fundamental issues: (i) how many predicates can compose an access predicate; (ii) how the access predicates are defined, i.e., how to choose the predicates that compose an access predicate; and (iii) how the existing access predicates are indexed. We propose three different strategies, one for each proposed matching solution, that address these three issues. The cluster access algorithm depends mainly on the solution adopted to the third issue of the clustering strategy, i.e., how the existing access predicates are indexed in order to efficiently compute the access predicates (and the corresponding clusters) satisfied by the event. This algorithm is described later in this paper for each proposed matching solution.
Fig. 2. The generic event matching algorithm.

3.3.3 The cluster checking algorithm

The cluster checking algorithm that corresponds to the cluster_checking function in Figure 2, is applied to each cluster whose access predicate is verified by the event being processed. The goal this algorithm is to determine the subscriptions of a cluster whose remaining predicates are all satisfied by the event. Basically, this algorithm iterates over each subscription of the cluster and applies a conditional subscription checking technique to verify whether the subscription is matched by the event. Note that this algorithm only needs to check those predicates of a subscription that are not part of the access predicate, since a subscription cluster is checked only if its access predicate is verified.

The subscriptions are clustered together according to their access predicate and number of predicates. The reason for clustering this way is twofold. First, when checking the subscriptions of a cluster, the cluster checking algorithm can predict the next subscriptions that need to be checked. Therefore, it can use prefetching techniques to decrease the number of cache misses that occur when the predicate arrays of the cluster are accessed. Second, since all subscriptions in a cluster have the same size, we can have different instances of the cluster checking algorithm, each one specialized for checking subscriptions with a different number of predicates. Thus, when checking the subscriptions of a cluster, it is not necessary to verify the number of predicates of the subscriptions. This way, the number of branching in the code of the cluster checking algorithm can be reduced.

We present two instances of the cluster checking algorithm in Figure 4. The first instance, represented by the check_3 function, is specialized for clusters with four integer arrays, i.e., clusters containing subscriptions that have exactly three predicates that do not make part of their access predicates. In our current implementation, we have a collection of similar functions specialized for small numbers of predicates, ten or fewer. There is one generic function to deal with clusters that have more than ten predicate arrays. This generic function is represented in Figure 4 by the generic_check function. The cluster checking generic function is more time consuming than the specialized functions because it needs an additional loop.
Fig. 3. The predicate matching algorithm.

However, most subscriptions usually have a small number of predicates, so the generic code will not be called often.

All cluster checking functions accept the following arguments: the number of subscriptions stored in the cluster, the integer arrays of the cluster to process, and an array where the identifiers of the matched subscriptions of the cluster are stored. The generic cluster checking function has an additional argument, n_preds, that contains the number of predicate arrays in the cluster to process. All these functions return the number of matched subscriptions of the cluster. These cluster checking functions apply a conditional subscription checking, i.e., predicate $i$ of a subscription of a cluster is verified if and only if predicate $i-1$ of the subscription is verified. In the examples of Figure 4, the preds variable refers to the predicate vector. Recall that this vector maintains the result of predicate checking. Notice that in order to check if a subscription predicate is satisfied, a cluster checking function only needs to verify whether the corresponding entry in the predicate vector is equal to one.

There are several important features of the cluster checking algorithm. First, notice that the subscriptions are stored column-wise, i.e., subscriptions have an entry in each array of the cluster where they are placed. The reason for this choice is to improve data locality as we explain in Section 3.3.5. Second, the loop over subscriptions is partitioned into two sub-loops, where the inner loop iterates over UNFOLD subscriptions of the cluster each time. The UNFOLD value is chosen so that UNFOLD array entries fit into a cache line. At the end of the inner loop, we call three times, with different arguments, a prefetch subroutine, prefetch(). This prefetch subroutine is implemented directly as assembly language prefetch instructions, telling the CPU to copy from RAM into the cache a cache line full of array entries, for processing in the near future. The LOOKAHEAD value is chosen so that the data arrives in the cache just before the CPU is ready to process that data. Such transfer is asynchronous, meaning that we can overlap computation and data transfer.

3.3.4 The generic maintenance algorithm

The maintenance algorithm is responsible for updating the data structures used by the event matching solution whenever a subscription is added or removed. This algorithm depends on two distinct factors: how the access predicate of a subscription is determined and how subscriptions are organized within a subscription cluster. The first factor determines the clustering strategy adopted by our matching solutions while the second factor determines the way subscriptions are stored in a cluster. This section does not detail how the access predicate of a subscription (or cluster) is determined. We will describe this aspect in the

\footnote{For simplicity of presentation, this code assumes that the number of subscriptions in the cluster is a multiple of UNFOLD. In practice, we need a small separate piece of code to deal with a remainder of up to UNFOLD-1 subscriptions.}
int check_3(int n_subs, int clu_arrays[][], int sub_matched[]) {
    int k, j, ansindex = 0;

    for(j = 0; j < n_subs; j += UNFOLD) {
        for(k = j; k < j + UNFOLD; k++) {
            if (preds[clu_arrays[0][k]] && preds[clu_arrays[1][k]] &&
                preds[clu_arrays[2][k]])
                sub_matched[ansindex++] = clu_arrays[3][k];
        }
        _prefetch(clu_arrays[0][j + LOOKAHEAD]);
        _prefetch(clu_arrays[1][j + LOOKAHEAD]);
        _prefetch(clu_arrays[2][j + LOOKAHEAD]);
    }
    return ansindex;
}

int check_generic(int n_subs, int clu_arrays[][], int sub_matched[],
    int n_preds) {
    int k, j, i, ansindex = 0;

    for(j = 0; j < n_subs; j += UNFOLD) {
        for(k = j; k < j + UNFOLD; k++) {
            for(i = 0; i < n_preds; i++) {
                if (!preds[clu_arrays[i][k]])
                    goto not_matched;
            }
            sub_matched[ansindex++] = clu_arrays[i][k];
        }
        _prefetch(clu_arrays[0][j + LOOKAHEAD]);
        _prefetch(clu_arrays[1][j + LOOKAHEAD]);
        _prefetch(clu_arrays[2][j + LOOKAHEAD]);
    }
    return ansindex;
}

Fig. 4. Two examples of functions that implement the cluster checking algorithm.

next sections when presenting the several versions of our matching solution. We also do not describe how subscriptions are physically added or removed to/from a cluster.

We divide this algorithm into two sub-algorithms: the subscription insertion and subscription deletion algorithms. The former one handles the insertion of subscriptions in the system and the latter one handles the deletion of subscriptions from the system. The first algorithm receives a textual representation of the subscription, e.g., ["price", =", 20], ["quantity", ≥", 10], adds the subscription to the system and returns the identifier assigned to the subscription. The second algorithm accepts the identifier of a subscription as input and removes this subscription from the system.

The subscription insertion algorithm, depicted in Figure 5, works as follows. A new subscription sub is submitted to the system as a set of predicates where each predicate is represented in a textual format.

Second, the algorithm assigns an identifier to sub (line 9), using the get_new_sid function. Then, it obtains the list of the identifiers of the predicates of sub through the parse_sub function (line 10). This function parses the textual representation of each predicate of sub and searches the parsed predicate in its predicate family (i.e., the predicate index associated with the attribute and comparison operator concerned by the predicate) to obtain the corresponding predicate identifier. If the predicate is not found in the predicate family (because it is the first time the predicate appears in a subscription), then a new predicate is
add_subscription(\text{sub})
1 \text{input:} \quad \text{a textual representation \text{sub} of a subscription}
2 \text{output:} \quad \text{the identifier of the new subscription}
3 \text{global variables:}
4 \text{sub\_preds; the subscription predicate table}
5 \text{local variables:}
6 \text{preds; a sub-set of the predicates of the new subscription; ap; an access predicate}
7 \text{sid; the assigned identifier of the subscription; c; a subscription cluster}
8 \text{Begin Body:}
9 \text{sid} \leftarrow \text{get\_new\_sid}()
10 \text{sub\_preds}[\text{sid}] \leftarrow \text{parse\_sub(\text{sub})}
11 (\text{ap, preds}) \leftarrow \text{assign\_cluster(\text{sid})}
12 \text{add\_sub\_cluster(\text{sid, ap, preds})}
13 \text{return sid}
\text{End Body}

Fig. 5. The subscription insertion algorithm.

created and added to the predicate family. The \text{parse\_sub} function returns the list \text{l} of the identifiers of the parsed predicates ordered by decreasing selectivity. Third, the algorithm uses the list \text{l} to update the entry in the subscription predicate table corresponding to \text{sub}, i.e., \text{sub\_preds[\text{sid}]}.

Fourth (line 11), the algorithm executes the \text{assign\_cluster} function to obtain the access predicate of the new subscription. Recall that the identifier associated with the access predicate of a subscription corresponds to the entry in the cluster vector that references the subscription cluster list containing the cluster where the subscription should be placed. The \text{assign\_cluster} function also returns the subset of predicates of \text{sub} that should be stored in the cluster (represented by \text{preds} in the figure), i.e., the predicates of \text{sub} that are not part of its access predicate. Finally, the algorithm calls the \text{add\_sub\_cluster} function that inserts the input subscription into its assigned cluster (line 12) and returns the identifier assigned to the subscription (line 13).

delete\_sub(\text{sid})
1 \text{input:} \quad \text{the identifier of the subscription to delete \text{sid}}
2 \text{global variables:} \quad \text{the subscription predicate table} \text{sub\_preds}
3 \text{local variables:}
4 \text{ap; an access predicate; size; the number of predicates of \text{sid} stored into the cluster}
5 \text{Begin Body:}
6 \text{recycle\_sid(\text{sid})}
7 (\text{ap, size}) \leftarrow \text{get\_cluster(\text{sid})}
8 \text{remove\_sub\_cluster(\text{sid, ap, size})}
9 \text{sub\_preds[\text{sid}]} \leftarrow \emptyset
\text{End Body}

Fig. 6. The subscription deletion algorithm.

The subscription deletion algorithm is detailed in Figure 6. The subscription \text{s} to delete is represented by its identifier \text{sid}. First (line 6), the algorithm recycles \text{sid}, so that a new subscription can later reuse this subscription identifier. Second (line 7), the \text{get\_cluster} function determines the access predicate of \text{s} and the number of predicates of \text{s} stored in its cluster. This information is used by the \text{remove\_sub\_cluster} function to remove the subscription from its cluster (line 8). Finally (line 9), the entry corresponding to the subscription deleted is reinitialized.
3.3.5 Cache behavior

Main-memory algorithms with a good cache behavior limit their number of cache misses having a good temporal and/or spatial locality. Spatial locality is achieved when data that are used consecutively by the algorithm are placed in consecutive memory addresses. Temporal locality is achieved when the same data is manipulated in consecutive instructions.

In terms of temporal locality, only entries in the predicate vector may be checked several times during the event matching process. All the other data structures used by our matching solutions are accessed once during the processing of an event. If the predicate vector is small, such as when a small number of predicates appears in many subscriptions, then it is resident in the processor cache, and our matching solutions have a good temporal locality.

Our matching solutions achieve a good spatial locality by: (i) putting together in the same cluster those subscriptions that are likely to be checked for the same event, (ii) using independent data-structures for predicate matching and subscription matching so that we can use optimized main memory data structures for predicate testing [Rao and Ross, 1999], and (iii) using size criteria (i.e., the number of predicates) to group subscriptions into clusters. Moreover, by organizing clusters in integer arrays, we can use asynchronous prefetch operations in the cluster checking algorithm in order to reduce the number of cache misses, which directly affects response time.

In the example of the cluster checking algorithm for subscriptions with three predicates stored in a cluster described in Figure 4 (the check_3 function), the columnar subscription storage means that every entry of clu_arrays[0] will be consulted. If the condition being tested is relatively selective, we may not consult every entry of clu_arrays[1] or clu_arrays[2]. In fact, we may in some cases avoid whole cache lines of these later arrays. If we had used a row-wise storage method we would have been forced to touch every cache line. Even though we are prefetching all cache lines from all three predicate arrays (clu_arrays[0], clu_arrays[1] and clu_arrays[2]), it may pay to avoid reading cache lines whenever it is possible for two reasons. First, the cache line may not have quite made it to the cache in time. Second and more important, some processors limit the number of simultaneous outstanding cache requests. Processors reserve the right to drop prefetch instructions when the limit has been reached, since prefetch instructions are not essential for correctness. Under such circumstances, we cannot be certain that a prefetched cache line will actually make it to the cache. If we access fewer cache lines, the effect of dropping prefetch instructions will be reduced. For larger numbers of predicates, we have found empirically that it is not worthwhile to prefetch all of the corresponding arrays. Prefetch instructions compete with one another according to the limit above. Therefore, it is better to avoid prefetching from arrays that are unlikely to be consulted, so that the frequently consulted arrays are prefetched more thoroughly.

3.3.6 Maintenance cost

The algorithm for adding a new subscription \( s \) to the system is very similar to the generic event matching algorithm. It also consists of three phases. First, the algorithm inserts the new predicates of \( s \) (if any) in the corresponding predicate families. Second, the algorithm determines the access predicate of \( s \). Third, the algorithm inserts the subscription into its assigned cluster. The cost of the insertion algorithm is then similar to the event matching cost. The deletion of subscriptions can be made even faster than the insertion if the algorithm is able to efficiently compute the subscription cluster of the subscription to remove. This can be achieved, for example, by maintaining the identifier of the subscription cluster list that contains the cluster of each subscription.

In the remaining sections, we describe the specific characteristics of each one of our event matching solutions. These solutions differ mainly on the clustering strategy and maintenance algorithm applied. Our matching solutions group subscriptions into subscription cluster lists according to their access predicates. As subscriptions in the same cluster list can only match events that verify the cluster list access predicate, only subscriptions whose access predicate is verified have to be checked. The performance challenge is to define access predicates so that each incoming event has to be matched only against a minimal number of subscriptions.
4 Single access predicate clustering algorithm

The index over the access predicates used by the propagation matching solution is very simple. In fact, this solution makes a direct mapping between the access predicate of a cluster (or subscription) and the single predicate that is associated with the access predicate. Moreover, the identifier of an access predicate is equal to the identifier of the single predicate that corresponds to the access predicate. This way, each predicate can be used as an index in the cluster vector to access the subscription cluster list associated with its access predicate. The size of the cluster vector is equal to the number of predicates that exist in the system. Predicates that are not access predicates of any subscription (i.e., they are never the most selective predicate of any subscription) reference an empty subscription cluster list. This algorithm selects the most selective predicate of each subscription as its access predicate since this is the predicate of the subscription with the lowest probability of being verified.

4.1 The cluster access algorithm

This section specifies the get_candidates function, that is applied in the context of the cluster access algorithm (see Section 3.3.2), used by the propagation matching solution. This function determines the subscription clusters whose access predicates are satisfied by the event being processed. The function is shown in Figure 7 and works as follows. For each predicate $p$ that is satisfied by the event (lines 10-14), this function simply checks the corresponding entry in the cluster vector (lines 11-13). If this entry references a subscription cluster list that is not empty, then the referenced cluster list is added to the set of clusters returned by this function. We would like to remark that the get_candidates function actually returns the set of subscription cluster lists whose access predicates are satisfied (line 15) and not the set of subscription clusters whose access predicates are satisfied.

```plaintext
get_candidates(satisfied_preds)
1 input: the list of satisfied predicates satisfied_preds
2 output: the set of subscription cluster lists whose access predicates are satisfied
3 global variables:
4 cluster_vector: the vector of references to subscription cluster lists
5 local variables:
6 candidate_C: a set of subscription cluster lists whose access predicates are satisfied
7 p: a satisfied predicate
8 Begin Body:
9 candidate_C ← ∅
10 foreach predicate p in satisfied_preds do
11   if cluster_vector[p] ≠ ∅ then
12     candidate_C ← candidate_C ∪ {cluster_vector[p]}
13 endif
14 endloop
15 return candidate_C
End Body
```

Fig. 7. The get_candidates function that returns the set of subscription clusters whose access predicates are verified by the event being processed.

4.2 The maintenance algorithm

We now describe the versions of the assign_cluster and get_cluster functions applied in the maintenance algorithm of the propagation matching solution. Both functions are very simple in this version of the matching solution. The assign_cluster function is used in the context of the subscription insertion algorithm and is responsible for determining the access predicate of a subscription, and its set of predicates that are not contained in the access predicate and must be stored in the cluster. In this version of the our matching
solution, this function returns the first predicate of the subscription, which is its most selective one, as its access predicate, and all the remaining predicates of the subscription.

The get_cluster function applied in the context of the subscription deletion algorithm is equally very simple. It accesses the subscription predicate table, using the identifier of the subscription $s$ to delete as an index, to obtain the list of predicates of $s$. It returns the first predicate of this list, which is the access predicate of the subscription, and the total number of predicates of $s$ minus one. This last number corresponds to the number of predicates of $s$ stored in the subscription cluster of $s$.

5 Schema-based subscription clustering

The previous section described a matching solution that clusters the subscriptions using a single predicate. A natural extension of this subscription clustering strategy is to factorize the subscriptions using several predicates instead of a single one. This way, we can reduce the number of subscriptions checked when an event is processed since the probability of two or more predicates are satisfied by an event can be lower than the probability of a single predicate. This section presents the approach followed by the other two matching solutions to index the multi-dimensional access predicates, i.e., it describes the data structures used by these two matching solutions to access the subscription clusters. We designate this approach as the schema-based clustering approach.

We would like to point out that the schema-based clustering approach only specifies the data structures that are needed to access the subscription clusters. It does not specify how the subscriptions are distributed among the clusters, i.e., it does not describe how the subscription clustering instance is chosen for a given set of subscriptions. The two matching solutions that follow this approach differ on the strategy used to cluster the subscriptions and also on the required knowledge. These specific aspects are described later in this section.

5.1 Principles

The schema-based clustering approach consists of: (i) grouping the subscriptions by several common predicates (that define the access predicate) and (ii) using multi-dimensional hashing to index the subscription clusters. More precisely, given a set of subscriptions $S$, and a clustering instance $C$ for $S$, clusters of $C$ are accessed through a set of multi-dimensional hash tables $H$. We designate $H$ as the indexing configuration for $C$. Each hash table of the configuration is associated with a set of attributes, called its schema, and supports the access to the clusters that have access predicates over this schema.

The event matching solutions that follow this approach are expected to be more efficient than the propagation matching solution since clustering subscriptions using two or more predicates reduces the average number of subscriptions that should be checked. In fact, the number of subscriptions sharing several predicates is usually smaller than those that share just one predicate. In addition, the probability of accessing a cluster is smaller in the former case (i.e., the clusters are defined using multiple predicates) than in the latter case. This means that the number of subscriptions that need to be checked in the third step of the event matching algorithm (see Section 3.3) is reduced and consequently, the average cost of the event matching algorithm decreases. However, to determine those subscription clusters that need to be checked, this new matching solution needs to access the index over the access predicates, which is much more complex than in the propagation matching solution. We conclude that there is a trade-off between the cost of accessing this new index and the saving of checking smaller clusters (and less subscriptions). The smaller the clusters, the lower the cost of checking them. But, at the same time, there are more clusters which may increase the cost of accessing the index in order to compute the clusters to check. To better illustrate these ideas consider the following example.

Example 1. Consider a collection $S$ of subscriptions and three independently distributed attributes $A$, $B$, and $C$ that are mentioned by some of the subscriptions. Suppose that each attribute’s domain has 200 values, and all values of each domain have the same probability of occurrence. Suppose that there are 7 million subscriptions in $S$, and that every subscription in $S$ has an equality condition on at least one of $A$, $B$, or $C$. There are seven nonempty subsets $X$ of \{A, B, C\}. For each such $X$, suppose there are exactly 1 million subscriptions from $S$ with equality predicates on exactly the attributes $X$. 

14
Consider two different ways of grouping subscriptions (i.e., defining the subscriptions clusters), $C_1$ and $C_2$. In $C_1$, subscriptions are clustered using single equality predicates on attributes $A$, $B$, or $C$. Subscriptions that have more than one equality predicate are placed in the corresponding cluster of one of the mentioned attributes in the equality predicates. If distributed uniformly, the population accessed through each hash table of the index over the access predicates would be 2.333 million subscriptions and each cluster would contain 11,665 subscriptions. Now, consider a second way of grouping the subscriptions, $C_2$. In $C_2$, subscriptions are clustered using conjunctions of two equality predicates on $AB$ or $BC$. When this is not possible (i.e., the subscription does not have two equality predicates concerning one of those two pairs of attributes), subscriptions are clustered using single equality predicates on attributes $A$, $B$ or $C$. Subscriptions with $AC$ are uniformly distributed between $A$ and $C$, and subscriptions with $ABC$ are uniformly distributed between $AB$ and $BC$. Therefore, the hash table populations would be $A$: 1.5 million; $B$: 1 million; $C$: 1.5 million; $AB$: 1.5 million; $BC$: 1.5 million. The average sizes of the corresponding clusters in each cluster index of $C_2$ would be $7,500 (A)$, $5,000 (B)$, $7,500 (C)$, $37.5 (AB)$ and $37.5 (BC)$.

We now consider the cost of matching an event that mentions $A$ and $B$ but not $C$. In both clustering instances, we first need to access all hash tables to determine those that have clusters that may contain subscriptions matched by the event, i.e., we need to compute all hash tables whose schema is contained in the schema of the event. This way, in $C_1$ we need to consult one of the $A$ clusters and one of the $B$ clusters, for a total cost of three hash table accesses, two hash table lookups and 23,330 subscription checks (on average). In $C_2$, we need to consult (on average) one of the $A$ clusters, one of the $B$ clusters, and one of the $AB$ clusters, for a total cost of five hash table accesses, three hash table lookups and 12,537.5 subscription checks. Based on this analysis, we expect the clustering instance $C_2$ to be preferred for this kind of event, unless the cost of two hash table access and one hash table lookup (the extra index cost) is higher than the cost of checking 10,792.5 subscriptions (the subscription checking gain).

The rest of this section is organized as follows. First, we define formally the notions of access predicate, indexing configuration and clustering instance. Second, we describe how the set of candidate subscription clusters can be computed from the set of satisfied predicates when subscriptions are clustered using multiple predicates. Third, we present the matching cost and space cost incurred when matching a set of subscriptions using a given clustering instance.

### 5.2 Definitions

Access predicates are defined as a conjunction of equality predicates. Given a set $S$ of subscriptions, we group them together in clusters using access predicates. More concretely:

- An access predicate consists of a pair $<id, preds>$ where $id$ is an identifier, and $preds$ is a set of equality predicates which are pairwise different over their attributes. The set of attributes occurring in $preds$ is called the schema of the access predicate.
- A subscription cluster is defined by a triple $<id, p, subs>$, where $id$ is the identifier of the cluster, $p$ is an access predicate, and $subs$ is a set of subscriptions having the same number of predicates such that each subscription contains all the predicates occurring in $p$.
- A clustering instance for $S$ is a set $C$ of clusters over the subscriptions of $S$ such that each subscription of $S$ appears in one and only one cluster of $C$. In what follows, we write $C(s)$ to denote the cluster containing subscription $s$, and $AP(C)$ to denote the set of all the access predicates to the clusters of $C$.
- Given an access predicate $p$ of $AP(C)$, $clusters(C, p)$ denotes the set of clusters in $C$ having $p$ as access predicate. Note that each of one these clusters differ from the others in the size of their subscriptions, i.e., two subscriptions with a different number of predicates, or size, can still have the same access predicate.

**Example 2.** Consider the following set of subscriptions $S$: $\{S_1 : (P_{A=1} \land P_{B \leq 5}), S_2 : (P_{A=1} \land P_{B=10}), S_3 : (P_{B=3}), S_4 : (P_{A=5} \land P_{B=2}), S_5 : (P_{C=2}), S_6 : (P_{A=8} \land P_{B=3} \land P_{C=5})\}$, where the index $i$ associated with each predicate $P_i$ represents the attribute, comparison operator and value of the predicate. The clustering instance $C_1$ described in Example 1 can be defined by the following set of subscription clusters $\{< 0, P_{A=1}, \{S_1, S_2\}>, < 1, P_{B=3}, \{S_3\}>, < 2, P_{A=5}, \{S_4\}> , < 3, P_{C=2}, \{S_5\}>$.
Remark that subscriptions $S_3$ and $S_6$ have the same access predicate but are placed into distinct clusters because they have a different number of predicates. The clustering instance $C_2$ of the same example can be defined as \( \{<0, P_{A=1}>, <1, P_{B=3}>, <2, P_{A=5}>, <3, P_{C=2}>\} \).

For instance, the schema of the access predicate $P_{A=5}P_{B=2}$ is $AB$.

In order to check the set of access predicates of a clustering instance $C$ against incoming events, we use one (or several) multi-dimensional hashing structures. These multi-dimensional hashing structures (or hash tables) are used to index $AP(C)$ and are designated as the indexing configuration of $C$. Each hashing structure is intended to index access predicates having a certain schema. More precisely:

- A multi-dimensional hashing structure over a set of access predicates is defined by a triplet $<T, A, h>$ where $T$ is the table where the indexed set of access predicates are kept, $A$ is a set of attributes called the schema of the hashing structure, and $h$ is a hash function which takes an event $e$, and returns the identifier of the access predicate (if it exists) which has $A$ as its schema, and is satisfied by $e$.

- An indexing configuration $\mathcal{H}$ for a clustering instance $C$ is the set of hashing structures given by $\mathcal{H} = \{<T_1, A_1, h_1>, ..., <T_n, A_n, h_n>\}$ where each access predicate $ap$ of $AP(C)$ is indexed in the hashing structure whose schema is equal to the schema of $ap$, and $\{A_1, ..., A_n\}$ denotes the set of distinct schemas of the access predicates in $AP(C)$. This set of schemas is called the schema of the indexing configuration.

Example 3. Using the same clustering instances defined in Example 2, the indexing configuration $\mathcal{H}$ for the clustering instance $C_1$ is defined as $\{<T_1, A, h_A>, <T_2, B, h_B>, <T_3, C, h_C>\}$. Table $T_1$ contains two entries referring to access predicates $P_{A=1}$ and $P_{A=5}$, $T_2$ contains a reference to access predicate $P_{B=3}$ and $T_3$ references access predicate $P_{C=2}$. In what concerns the clustering instance $C_2$, $\mathcal{H}$ is equal to $\{<T_1, A, h_A>, <T_2, B, h_B>, <T_3, C, h_C>, <T_4, AB, h_{AB}>, <T_5, BC, h_{BC}>\}$, where table $T_1$ indexes $P_{A=1}$, $T_2$ indexes $P_{B=3}$, $T_3$ indexes $P_{C=2}$, $T_4$ indexes $P_{A=1}P_{B=10}$ and $P_{A=5}P_{B=2}$, and finally $T_5$ indexes $P_{B=3}P_{C=5}$. The schema of the indexing configuration for the clustering instance $C_2$ is defined by the following set: $\{A, B, C, AB, BC\}$.

5.3 The cluster access algorithm

In this approach of our matching solution, subscriptions are clustered using a conjunction of predicates which are indexed in an indexing configuration $\mathcal{H}$. The hash tables of $\mathcal{H}$, managed by this approach of the matching solution, correspond to the special index over the subscription cluster lists (or access predicates) shown in Figure 1. We now detail the corresponding get_candidates function. Recall that this function determines the subscription clusters that must be checked when processing an event and is used to implement the cluster accessing algorithm, defined in Section 3.3.2.

The get_candidates function, depicted in Figure 8, iterates over the existing set of hash tables and only considers those hash tables whose schema is included in the equality schema of the event being processed\(^3\), i.e., those hash tables that contain access predicates that can be verified by the event. For each of such hash tables (lines 12-19), this function hashes the event taking into account the schema of the hash table\(^4\) to obtain the access predicate $ap$ associated with the hashed value (line 14), i.e., the access predicate indexed in the hash table that is verified by the event. It may happen that no access predicate associated with this hashed value is found. This situation occurs when the event does not match any access predicate indexed in the hash table. If the hashed value corresponds to an access predicate (lines 15-17), then the reference to the subscription cluster list corresponding to the access predicate found is added (line 16) to the list of subscription clusters that constitute the result returned by this function (line 20).

\(^3\) By equality schema of an event, we mean the set of attributes of the event that concern the set of equality predicates satisfied by the event.

\(^4\) Actually, this function hashes the satisfied predicates corresponding to the schema of the hash table.
**get_candidates**(satisfied_preds, e)

1. **Input:** the list of satisfied predicates satisfied_preds; the event e being processed
2. **Output:** the set of subscription cluster lists whose access predicates are satisfied
3. **Global variables:**
   4. `cluster_vector`: the vector of references to subscription cluster lists
   5. `H`: the set of existing hash tables
4. **Local variables:**
   5. `candidate_C`: the set of subscription cluster lists whose access predicates are satisfied;
   6. `ap`: a satisfied access predicate;
   7. `event_schema`: the schema of e; and
   8. `H`: a hash table
5. **Begin Body:**
   6. `candidate_C ← ∅`
   7. `event_schema ← get equality schema of e`
   8. **foreach** hash table `H` in `H` **do**
      9. **if** `H.schema ⊆ event_schema** then**
         10. `ap ← H.hash(e, satisfied_preds)`
         11. **if** `ap` is valid **then**
             12. `candidate_C ← candidate_C ∪ {cluster_vector[ap]}`
         13. **endif**
     14. **endif**
   15. **endloop**
   16. **return** `candidate_C`
   17. **End Body**

**Fig. 8.** The `get_candidates` function for the schema based clustering approach. This function returns the set of subscription clusters whose access predicates are verified by the event being processed.

### 5.4 Matching cost of a clustering instance

This section describes the cost of matching an event for a given clustering instance.

From now on we assume that we have a set `S` of subscriptions, a clustering instance `C` for `S`, and `H` is the associated indexing configuration. The matching cost per event of the matching solution can be decomposed into three main parts: the cost needed for computing the value of the predicate vector, the cost of computing the references to the relevant clusters, and the cost of checking the set of accessed subscriptions. As it is generally possible to build several clustering instances for `S`, and the two later costs are sensitive to the way the subscriptions are clustered, the problem is to choose the most efficient clustering.

The cost of matching an event `e` against `S` using `C` having already determined the set of predicates satisfied by `e` includes: (i) the cost for retrieving the relevant multi-attribute indexes (i.e., hash tables) for the event, (ii) the hashing cost for each relevant hash table, and (iii) the cost for checking the clusters whose access predicates are satisfied by the event.

Thus, the general formula of the average matching cost per event of a clustering instance is given by:

\[
\text{matching}(S, C, H) = \text{index\_retrieving}(H) + \sum_{H \in H} \mu(H)\text{hashing}(H) + \sum_{p \in AP(C)} \nu(p) \left( \sum_{c \in \text{cluster}(C, p)} \text{checking}(p, c) \right)
\]

where \(\text{index\_retrieving}(H)\) is the cost for retrieving the indexes, \(\mu(H)\) is the probability that the schema of the incoming event includes the schema of the hash table `H`, \(\text{hashing}(H)\) is the cost of running the hash function of `H`, \(\nu(p)\) is the probability for an event to satisfy the access predicate `p`, \(\text{checking}(p, c)\) is the checking cost for a cluster `c` having `p` as access predicate (i.e., the cost of checking all subscriptions in the cluster `c` using a conditional subscription checking technique), and \(\sum_{c \in \text{cluster}(C, p)} \text{checking}(p, c)\) is the total cost for checking all the subscriptions in the set of clusters having `p` as their access predicate. This
cost takes into account the fact that the group of predicates in \( p \) is already checked, so only the remaining predicates have to be checked.

### 5.4.1 Index retrieving cost

Through the analysis of the \textit{get candidates} function described in Section 5.3, we can conclude that the cost for retrieving the relevant indexes, i.e., the hash tables whose schema is included in the event schema, is linear in the number of hash tables in the hashing configuration \( \mathcal{H} \). Therefore, the index retrieving cost present in the cost formula 1 is equal to \( C_r + K_r \times |\mathcal{H}| \), where \( C_r \) and \( K_r \) represent two constants and \( |\mathcal{H}| \) represent the number of indexes (hash tables) in \( \mathcal{H} \).

### 5.4.2 Simplified matching cost formula

In the following, we assume that: (i) the hashing cost is independent of the size of the hashing structure, but linear in the size of the schema of the hashing structure; and (ii) the cost of checking a cluster is linear in the number of subscriptions in the cluster\(^5\). According to these assumptions, the average matching cost per event of a clustering instance (formula 1) can then be simplified to:

\[
\text{matching}(S, C, \mathcal{H}) = C_r + K_r \times |\mathcal{H}| + \sum_{H \in \mathcal{H}} \mu(H)(C_h + K_h \times |H.A|) + \sum_{s \in S} \nu(C(s), p) \times \text{checking}(C(s), p, s)
\]

where \( |H.A| \) represents the size of the schema of the hash table \( H \), \( C_h \) and \( K_h \) represent two constants involved in the cost of running a hash function (i.e., \( \text{hashing}(H) = C_h + K_h \times |H.A| \)) and \( C(s) \) and \( C(s).p \) represent the cluster containing \( s \) and its access predicate, respectively.

### 5.4.3 Analysis of the matching cost

We now analyze the cost of our matching solution when using the schema-based clustering approach to cluster the subscriptions. We compare the matching cost of our schema-based clustering approach (represented by Formula 4.2) under some assumptions, with the cost of matching subscriptions using a scan approach. In the scan approach, \textit{all} subscriptions are checked by applying a conditional subscription checking technique. In this analysis, we do not take into account the cost of computing the predicates satisfied by an event since this cost is the same for both approaches.

In this analysis, we do the following assumptions:

- Subscriptions have a fixed number \( p \) of predicates.
- Subscriptions are uniformly distributed over all possible combinations of schemas with \( p \) attributes. The number of existing subscription schemas \( C_s \) is then equal to \( a!/(p!(a-p)!) \), where \( a \) represents the number of attributes.
- The total number of subscriptions is \( S_T \). Subscriptions are uniformly distributed through the \( C_S \) schemas. Therefore, the number of subscriptions per schema \( K_{sub} \) is equal to \( S_T/C_S \).
- The indexing configuration \( \mathcal{H} \) used for this distribution of subscriptions defines a hash table per existing subscription schema. Each subscription \( s \) is placed in the hash table whose schema is equal to its schema. In this case, \( |\mathcal{H}| = C_S \) and \( |H.A| = p \). Moreover, the cost of checking a subscription \( s \), i.e., \( \text{checking}(C(s), p, s) \), includes only the cost of accessing the identifier of \( s \) in its cluster. This happens because all predicates of \( s \) belong to its access predicates and thus it is not required to verify them. We represent this cost by \textit{access}.
- The schema of every event contains all existing attributes. This means that the schema of every hash table \( H \) is contained in the schema of the event, i.e., \( \mu(H) = 1 \). In addition, we also assume that the event attribute values are uniformly distributed and that each attribute domain has \( n \) values. This means

\(^5\) All of these assumptions are consistent with the implementation of the matching algorithm.
that clusters have a uniform probability of having their access predicates satisfied by an event, i.e., the probability that the access predicate \( ap \) of a cluster is satisfied is \( \nu(ap) = 1/n^p \).

Under these assumptions, our formula 4.2 for matching cost can be simplified to:

\[
\text{matching} = C_r + C_s \times (K_r + C_h + K_h \times p + K_{sub} \times \text{access}/n^p)
\]

The matching\_scan cost of matching the subscriptions using the scan approach is equal to \( S_T \times \text{check} \), where \( \text{check} \) represents the average cost of checking a subscription with \( p \) predicates. The \( \text{check} \) cost is higher than the access cost since \( \text{check} \) encloses the average cost of accessing and testing the \( p \) predicates of a subscription. We can say that \( \text{access} = \alpha \times \text{check} \), with \( 0 < \alpha < 1 \)

The gain of our solution with respect to the scan approach is then given by:

\[
gain = \frac{\text{matching}_{\text{scan}}}{\text{matching}} = \frac{K_{sub} \times \text{check}/(c_{\text{hash}} + K_{sub} \times \text{access}/n^p)}{1/(c_{\text{hash}}/(K_{sub} \times \text{check}) + \alpha/n^p) \quad (4.3)}
\]

where \( c_{\text{hash}} \) is equal to \( C_r/C_s + K_r + C_h + K_h \times p \) and represents the average cost per hash table of verifying that the schema of a hash table is contained in the schema of an event plus the cost of running a hash function.

The matching gain depends on two factors: \( c_{\text{hash}}/(K_{sub} \times \text{check}) \) and \( \alpha/n^p \). The smaller these two factors are, the better our solution is compared with the scan approach. When the number of subscriptions per schema is maximal, i.e., all possible combinations of predicate values are specified by the subscriptions, then \( n^p = K_{sub} \). The gain formula (4.3) can then be simplified to:

\[
gain = K_{sub} \times \text{check}/(c_{\text{hash}} + \alpha \times \text{check})
\]

In this situation, if the number of subscriptions per schema is large\(^6\) (e.g., tens of thousand), our matching solution is much better than the scan approach, and the gain depends on how large \( K_{sub} \) is. However, if the number of subscriptions per schema is small, then the scan approach can be better. We can then conclude that when \( K_{sub} \) is small, the strategy of creating a hash table for each existing subscription schemas may not be profitable. In this case, a better strategy consists in creating hash tables whose schema contains only \( k \) attributes instead of \( p \) attributes, with \( 1 \leq k \leq p \). In fact, by decreasing the number of attributes in the schema of each hash table, we increase the number of subscriptions per schema (i.e., \( K_{sub} \)), because the number of created hash tables is smaller. This way, the factor \( c_{\text{hash}}/(K_{sub} \times \text{check}) \) decreases as desired. The second factor \((\alpha/n^p)\) that influences the matching gain is now equal to \( \alpha'/n^k \), where \( \alpha' \) takes into account that the access cost represents the cost of checking \( p - k \) predicates of each subscription, since in this case the access predicate of each subscription only contains \( k \) of their \( p \) predicates.

Two distinct situations can now occur. The first one corresponds to \( c_{\text{hash}}/(K_{sub} \times \text{check}) > \alpha'/n^k \). In this case, the gain of our solution is at least equal to \( K_{sub} \times \text{check}/(2 \times c_{\text{hash}}) \). If the gain is still small, we can decrease \( k \) even more to increase the value of \( K_{sub} \) (unless \( k \) is already equal to 1). However, by reducing the \( k \) parameter, we can reach the second situation: \( c_{\text{hash}}/(K_{sub} \times \text{check}) < \alpha'/n^k \). The gain of our matching solution is now at least equal to \( n^k/(2 \times \alpha') \). If \( n \) is not very small and \( k \) is greater than 1, then the gain is high. But, if the gain is small, and \( k \) is still smaller than \( p \), then we can increase the number of predicates of subscriptions that are used to factorize them (i.e., increase \( k \)). The two parameters \( k \) and \( K_{sub} \) influence each other, since increasing one implies decreasing the other.

\(^6\) Recall that the number of subscriptions per schema depends on three factors: total number of subscriptions, number of attributes and number of predicates per subscription.
5.5 Space cost of a clustering instance

The space cost of a clustering instance $C$ on $S$ using the indexing configuration $\mathcal{H}$ includes: (i) the space required for storing the hashing structures, and (ii) the space required for storing the clusters. The space cost is then given by

$$\text{Space}(S, C, \mathcal{H}) = \sum_{H \in \mathcal{H}} \left( \text{hash}\_\text{space}(H) + \sum_{p \in \text{AP}(H,A)} \text{entry}\_\text{space}(H, p) \right) + \sum_{c \in \text{cluster}(C)} \text{cluster}\_\text{space}(c, p, c)$$

where $\text{hash}\_\text{space}(H)$ is the space necessary for a hashing structure $H$, $\text{entry}\_\text{space}(H, p)$ is the space required to manage an entry for access predicate $p$ in the hashing structure $H$, and $\text{cluster}\_\text{space}(c, p, c)$ is the space occupied by the cluster $c$. The space needed for storing a hashing structure $H$ is equal to $K_{h1} + K_{h2} \times \text{length}(| \text{AP}(H,A) |)$, where $K_{h1}$ and $K_{h2}$ represent two constants and $\text{length}(| \text{AP}(H,A) |)$ represents the length of the array required to hold the references to all entries stored in $H$. The size of an entry for access predicate $p$ is equal to $K_{e1} + K_{e2} \times | H.A |$ where $K_{e1}$ and $K_{e2}$ represent two constants. In what concerns the data structures for clusters (see Section 3.2), recall that for each subscription, the matching solution stores the predicates of the subscription that are not contained in its access predicate plus the identifier of the subscription. Therefore, the size required to hold a subscription cluster, $\text{cluster}\_\text{space}(c, p, c)$, is equal to $K_{space} \times | c\.subs | \times (( | s_i | + 1 \times | c\.p\_preds |)$, where $K_{space}$ represents a constant, $| c\.subs |$ is the numbers of subscriptions in cluster $c$, $s_i$ is a subscription in $c$, $| s_i |$ represents the number of predicates of the subscription $s_i$ stored in $c$ (and of all subscriptions in $c$) and $| c\.p\_preds |$ is the number of predicates in the access predicate of $c$.

6 The static clustering matching solution

This section presents our second matching solution, called the static clustering matching solution, that uses the schema-based clustering approach to index the clusters. The strategy used by this solution to define a clustering instance, called static clustering approach, consists of a cost-based approach that uses statistics about the subscriptions and the events to determine the best subscription clustering instance for a given set of subscriptions. The computation of the clustering instance requires information about all subscriptions and the computation of the probability of each possible access predicates is satisfied by an event. This last point means that it is necessary to maintain statistics about the events, namely, the distribution of their schemas and the distribution of values per attribute and their corresponding probabilities. In this solution, we do not address how these statistics are obtained and maintained.

Let $S$ be a set of subscriptions. The problem solved by this matching solution is to determine the clustering instance for $S$ that minimizes the cluster checking cost described in the Section 5.4, under the constraint that the total space occupied by the subscription clusters and the hashing structures is less than a given amount of (main memory) space. In this section, we pose the clustering problem in terms of minimization of the matching cost under a space constraint, we enumerate the search space, and we propose a greedy algorithm that produces a locally optimal clustering solution.

The static clustering approach specifies the subscription clustering instance based on the global knowledge of all subscriptions in the system and on statistics about the submitted events. However, with this approach, a previously optimal clustering instance can become obsolete and inefficient when a large number of subscriptions is added or deleted. In the same way, event patterns may change over time which degrades an initial optimal clustering instance.

To cope with the degradation of the clustering instance, the optimal clustering instance can be periodically recomputed from scratch in order to adapt it to the new situation. Due to the complexity of this reorganization, this solution is well suited only for applications where subscription and event distributions are relatively stable during large periods of time. Indeed, this static approach is clearly impracticable when event or subscription distributions are evolving continually. For this reason, we have developed a dynamic
clustering strategy that adapts the current subscription clustering instance to changes in the subscription and event distributions while supporting high rates of subscription changes and incoming events. This approach is used by the dynamic clustering matching solution shown in Section 7.

6.0.1 Computing the best clustering instance

An exhaustive algorithm would examine all possible clustering instances. In such approach, the algorithm builds each clustering instance by picking out one possible equality predicate group for each subscription as its access predicate and finds the associated matching cost and space. The number of clustering instances examined by an exhaustive algorithm is then \( \Pi_{s \in S}(2^{|P(s)|}) = 2^{|S|} \), where \( |P(s)| \) is the number of equality predicates of \( s \), \( |S| \) is the average number of equality predicates per subscription, and \( |S| \) represents the number of subscriptions. This complexity makes the exhaustive algorithm impracticable.

We propose a greedy algorithm whose worst case complexity is \( |S| \times (|GA(S)|)^2 \), where \( GA(S) \) is the set of attribute groups concerning equality predicates occurring in subscriptions of \( S \), and \( |GA(S)| \) represents the cardinality of \( GA(S) \). Notice that \( GA(S) \) is bounded by \( 2^{\lambda |A|} \) where \( \lambda \) denotes the set of attributes occurring in equality predicates of \( S \). Our algorithm starts from a “natural” clustering that consists of grouping the subscriptions using simple equality predicates as access predicates. In fact, using these equality predicates as access predicates incurs no additional hashing (and space) cost since the required hashing structures to index single-dimensional access predicates are already defined and used for the predicate testing phase of the global event matching algorithm. This natural clustering corresponds to the clustering applied by the propagation matching solution if we consider that the most selective predicate is always an equality predicate.

In the static clustering matching solution, we can consider that we use only a single-dimensional hash table, called \( H_0 \), that hashes all access predicates that contain a single predicate. This special hash table accesses directly the cluster vector using the satisfied equality predicates as indexes in this vector (as the solution adopted in the propagation matching solution). Then, we improve this natural clustering by defining additional multi-dimensional hash tables. The additional hash tables are chosen incrementally step by step. At each step, we use a benefit function to decide which hash table to add. The benefit function is based on the notion of best clustering instance for an indexing configuration schema. We first explain this notion and then we present the benefit function and describe the greedy algorithm. This algorithm produces a local optimum.

6.0.1.1 Best clustering instance for a hashing configuration schema: Let \( S \) be a set of subscriptions, \( \mathcal{A} \) an indexing configuration schema for \( S \) and \( \mathcal{C}(\mathcal{A}) \) the set of all clustering instances having \( \mathcal{A} \) as the indexing configuration schema. We call best clustering instance for \( \mathcal{A} \) to the clustering instance that corresponds to the best matching cost among all clustering instances in \( \mathcal{C}(\mathcal{A}) \). Such clustering instance can be built by iterating over \( S \) and choosing, for each subscription \( s \) in \( S \), the predicate access \( p \) in \( GP(s) \) such that \( schema(p) \in \mathcal{A} \) and minimizes \( \nu(p) \times checking(p, s) \). Indeed, the matching cost formula 1 shows that two clustering instances associated with the same indexing configuration schema only differ on the total checking cost (see line 2 of the formula). In the rest of this section, \( best(S, \mathcal{A}) \) denotes the best clustering instance for \( \mathcal{A} \), \( bestcost(S, \mathcal{A}) \) denotes the cost of such a best clustering instance and \( space(S, \mathcal{A}) \) denotes its space cost.

6.0.1.2 Benefit of an additional hashing structure: Let \( S \) be a set of subscriptions, \( \mathcal{H} \) an indexing configuration for \( S \), and \( \mathcal{A} \) its schema. The matching benefit of adding a hashing structure \( H \) of schema \( A \) to \( \mathcal{H} \) with respect to \( \mathcal{A} \) is denoted by \( B(S, \mathcal{A}, \mathcal{A}) \) and is defined as \( bestcost(S, \mathcal{A}) - bestcost(S, \mathcal{A} \cup \{A\}) \). The space cost of adding \( H \) is denoted by \( DS(S, \mathcal{A}, \mathcal{A}) \) and is defined by \( space(S, \mathcal{A} \cup \{A\}) - space(S, \mathcal{A}) \) if \( space(S, \mathcal{A} \cup \{A\}) > space(S, \mathcal{A}) \) and 0 otherwise.

\footnote{The set of all possible equality predicate groups of \( s \) represents the set of access predicates that can be defined for \( s \) and is denoted by \( GP(s) \).}
**6.0.1.3 The greedy algorithm:** The greedy algorithm applied to compute the best clustering instance for $S$ is presented in Figure 9. This algorithm takes as input a set $S$ of subscriptions, and a space bound $Maxsize$. It returns a hashing configuration schema and the associated best clustering instance that fits into $Maxsize$ and minimizes the matching cost associated with $S$.

```plaintext
compute_best_clustering(S, Maxsize)
1 input: a set of subscriptions $s$, and the space bound $Maxsize$
2 output: an indexing configuration schema and the associated clustering instance
3 Begin Body
4 $A_0 \leftarrow \{A \mid A \text{ is an attribute involved in some equality predicate in } S\}$
5 $A \leftarrow A_0$
6 $C \leftarrow \text{best}(S, A)$
7 $GA \leftarrow GA(S) - A$
8 while (space$(S, A) < Maxsize$) do
9     Among all schemas in $GA$ let $B$ be the schema which has the
10     maximum positive benefit per unit space wrt $A$
11     such that space$(S, A \cup \{B\}) < Maxsize$.
12     if $B$ does not exist then
13         return $(A, C)$
14     else
15         $A \leftarrow A \cup \{B\}$
16         $C \leftarrow \text{best}(S, A)$
17         $GA \leftarrow GA - \{B\}$
18     endif
19 endloop
20 return $(A, C)$
End Body
```

Fig. 9. The greedy algorithm used to compute the best clustering instance.

The greedy algorithm used to compute the best clustering instance for the set of subscriptions $S$, starts by computing the best clustering instance $C$ for the initial indexing configuration schema $A$. $A$ is composed by the set of single-attribute schemas that concern the equality predicates occurring in $S$ (lines 4-6). Then, the algorithm determines the set of schemas $GA$ that concern conjunctions of equality predicates occurring in the existing subscriptions (line 7). This initial value of $GA$ does not include those single-dimensional schemas already present in $A$. $GA$ contains the set of possible schemas that can be added to $A$ to improve the initial indexing configuration schema.

Afterwards, the greedy algorithm executes a cycle (lines 8-19) that works as follows. For each schema present in $GA$, the algorithm computes the benefit of having an additional hash table with that schema and the corresponding cost in space. After having computed these two values for all schemas, the algorithm chooses the schema $B$ whose hash table has the best positive benefit per unit of space such that the space cost of the new indexing configuration is inferior to the bounding value $Maxsize$ (lines 9-11). This schema $B$ is added to the current indexing configuration $A$ (line 15). Then (lines 16-17), the best clustering instance for $A$ is updated, $B$ is removed from $GA$ and the algorithm restarts the iteration over the schemas of $GA$ (lines 8-19). This cycle ends when it is not possible to find a schema of $GA$ that has a positive benefit per unit of space and still verifies the space limit. When this situation occurs, the best clustering instance and the corresponding indexing configuration schema for the set of subscriptions $S$ are stored in variables $C$ and $A$, respectively. These two values are then returned by the algorithm.

**6.1 Maintenance of the best clustering instance**

This section describes the version of the maintenance algorithm used by the static clustering solution, called the static maintenance algorithm. This maintenance algorithm does not change the way subscrip-
tions are clustered. Each new subscription is placed into the best existing hash table. If there is not any multi-dimensional hash table where the subscription can be placed, then the subscription is placed in the special single-dimensional hash table $H_0$, and its access predicate contains only its most selective equality predicate. This maintenance algorithm requires an additional data structure, named hash table vector, that maintains the hash table where each subscription is indexed. This vector is indexed by the identifier of the subscription and is used to speed up the deletion of subscriptions.

### 6.1.1 Insertion of subscriptions

Before describing how subscriptions are inserted or removed when using the static clustering maintenance algorithm, we present some definitions that will be useful in this context. The **selectivity** of a hash table is defined as the average of the selectivities of all access predicates that can be indexed in the hash table. The selectivity can vary between one and zero, with one being the lowest selective value and zero the highest value. The **equality schema** of a subscription is the set of attributes concerned by the equality predicates of the subscription.

```plaintext
assign_cluster(sid)
1 input: the identifier sid of the subscription to add
2 output: the hash table where the new subscription should be placed and
3 the predicates of the subscription that do not make part of its access predicate
4 global variables:
5 sub_preds: the subscription predicate table
6 $H$: the set of hash tables; table_vector: the hash table vector
7 local variables:
8 $H_i$: a hash table; eq_schema: the equality schema of the subscription
9 $H_{best}$: the current best hash table; min_prob: the cluster average probability of $H_{best}$
1 Begin Body:
2 min_prob ← 2.0
3 eq_schema ← get equality schema of sub_preds[sid]
4 foreach hash table $H_i$ in $H$ do
5 if $H_i$.average_probability < min_prob AND $H_i$.schema ⊆ eq_schema then
6   $H_{best}$ ← $H_i$
7   min_prob ← $H_i$.average_probability
8 endif
9 endloop
10 if min_prob = 2.0 then // no multi-dimensional hash table found
11   $H_{best}$ ← $H_0$
12 endif
13 table_vector[sid] ← $H_{best}$
14 return $H_{best}$.add(sid)
1 End Body
```

**Fig. 10.** The `assign_cluster` function. This function determines the best subscription cluster list where a subscription should be stored.

Figure 10 shows the `assign_cluster` function, used in the context of the subscription insertion algorithm (see Section 3.3.4). This function starts by computing the best existing hash table for the new subscription $s$. The best hash table is the one for which the access predicate of $s$ has the highest selective value. To determine the best hash table (lines 14-19), this function iterates over the set of existing hash tables whose schema is contained in the equality schema of $s$, and searches the hash table where $s$ has the smallest probability of being accessed, i.e., where the access predicate of $s$ is more selective. Actually, in order to reduce the insertion cost of a subscription, the insertion algorithm considers the selectivity of each hash
table where \( s \) can be indexed, and does not consider the selectivity of the access predicate of \( s \). This way, the insertion algorithm does not need to compute the access predicate of \( s \) and the corresponding selectivity for each hash table where \( s \) can be indexed. Recall that the access predicate of a subscription depends on the schema of the hash table where the subscription is indexed. If no hash table is found, meaning that the equality schema of \( s \) does not include the schema of any existing hash table, then the insertion algorithm chooses the special single-dimensional hash table as the best hash table to index \( s \) (lines 20-22).

After having determined the hash table \( H_{\text{best}} \) where the access predicate of \( s \) should be indexed, the assign\_cluster function registers the association between the subscription and \( H_{\text{best}} \) (line 23) in the hash table vector. Then, the algorithm computes the access predicate of \( s \) and adds it to \( H_{\text{best}} \). This is done by the hash table \texttt{add} function. This function first hashes the predicates of \( s \) whose schema is equal to the schema of \( H_{\text{best}} \) (i.e., the access predicate of \( s \)) in order to obtain the identifier of the access predicate associated with the hashed value. If no identifier is found, meaning that \( s \) is the first subscription with such an access predicate, then the \texttt{add} function creates a new subscription cluster list with the corresponding identifier, and registers the association between these two values in the \texttt{cluster} vector data structure. Finally, the \texttt{add} function returns the identifier of the access predicate of \( s \) as well as the other predicates of \( s \) that are not included in its access predicate. These two values, the identifier and the predicates not included in the access predicate, constitute the result returned by the assign\_cluster function.

### 6.1.2 Deletion of subscriptions

The implementation of the get\_cluster function, used by the deletion algorithm to obtain the subscription cluster where the subscription is placed (shown in Section 3.3.4), is straightforward. First, this function uses the hash table vector to obtain the hash table where the subscription \( s \) to delete is indexed. Afterwards, and taking into account the schema of the hash table where \( s \) is indexed, the function determines the access predicate of \( s \) and hashes it to obtain its identifier stored in the hash table. Besides this identifier, the get\_cluster function must also return the number of predicates of \( s \) that are stored in its subscription cluster. This number is equal to the number of predicates of \( s \), which are stored in the corresponding entry of the subscription predicate table, minus the number of predicates in the access predicate of \( s \). Finally, this function checks if \( s \) is the last subscription placed in the corresponding subscription cluster list. If that is the case, the function removes the access predicate of \( s \) from the hash table, since the associated subscription cluster list becomes empty.

### 7 The dynamic clustering matching solution

This section describes the third version of our matching solution called the \textit{dynamic clustering} matching solution. This version is similar to the \textit{static clustering} solution in the sense that both solutions follow the schema-based clustering approach, i.e., both solutions use the same indexing structure over the access predicates. However, the \textit{dynamic clustering} solution defines a different clustering strategy to define and maintain the subscription clusters. The clustering strategy applied in this matching solution incrementally adapts clustering to changes in subscription and event patterns. This clustering strategy uses some heuristics to dynamically decide: (i) when to redistribute subscriptions from a given cluster to other more profitable clusters, (ii) when to create a new hash table, and (iii) when to delete a hash table and redistribute its subscriptions.

These decisions rely on two metrics called \textit{cluster benefit margin} and \textit{hash table benefit}. A cluster should be redistributed if its benefit margin is high. A hash table is created when its benefit is sufficiently high and removed when its benefit is too small. We first define these metrics and show how they can characterize the current state of a clustering instance. Then, we describe the algorithm that is responsible for the evolution of the clustering instance. This algorithm is parameterized by three thresholds that correspond to the maximal value of the \textit{cluster benefit margin} and the minimal and maximal value of the \textit{hash table benefit}. Moreover, this algorithm needs to maintain information about the probability of each possible access predicates is satisfied by an event and the number of subscriptions that can be better placed in \textit{potential} hash tables. Finally, we discuss the maintenance cost and the impact of the threshold values on the trade-off between the maintenance cost and the matching cost.
7.1 Cluster benefit margin

The cluster benefit margin focuses on the number of checks that could be saved from a given clustering instance if all possible access predicates were used. Let \( c \) be a cluster and \( s \) a subscription in \( c \). The benefit margin of \( s \) in \( c \) is equal to \( \left( \nu(p_c) - \nu(P(s)) \right) \) where \( p_c \) is the access predicate of \( c \), \( P(s) \) is the maximal group of equality predicates of \( s \), and \( \nu(p_c) \) and \( \nu(P(s)) \) are the probability that an incoming event satisfies \( p_c \) and \( P(s) \), respectively. The rationale for this is that \( P(s) \) is a superset of \( p_c \). The benefit margin of a cluster \( c \) is noted \( BM(c) \) and is defined as the sum of all the benefit margins of its subscriptions, and is equal to \( \sum_{s \in c} \left( \nu(p_c) - \nu(P(s)) \right) \).

If, on average, when a cluster is accessed there is a large number of subscriptions checked that are not matched, then the cluster is not optimal since we are checking unnecessarily a large number of subscriptions. But, if the cluster is almost never accessed then the checking cost of this non-optimal cluster per event is negligible. Moreover, if a cluster is never accessed it is always optimal. However, if the cluster is frequently accessed then the average checking cost of the cluster is not negligible. In this case, we say that the benefit of the cluster is high and it is better to redistribute (if possible) its subscriptions to other better clusters. For example, we can reorganize the subscriptions of a cluster using another equality predicate of its subscriptions\(^8\), adding it to the access predicate of each subscription, and redistributing the subscriptions by the clusters corresponding to the new access predicates. This way, we decrease the probability of checking these subscriptions.

7.2 Hash table benefit

The benefit of a hash table \( H \), noted \( B(H) \), is the average number of checks that are saved when using a given hash table. This benefit is equal to \( | H | - nbchecks \), where \( | H | \) is the number of subscriptions in the hash table and \( nbchecks \) is the average number of subscriptions indexed in \( H \) that are accessed per event.

A hash table has a high benefit if the average number of subscriptions per event that are indexed in the hash table and are not checked is high. If this number is low, two scenarios are possible: either almost all subscriptions indexed in the hash table are on average checked when an event is processed, or the number of subscriptions indexed in the hash table is small. In both cases, this means that the hash table is not beneficial. Since we are using conjunctions of equality predicates (which are expected to be relatively selective) as access predicates, we can consider that the probability of the former case to occur is negligible.

7.3 The algorithm metrics

We want to use metrics that are not expensive to compute, since they should be computed during the execution of the matching algorithm. For this reason, our metrics consist of an approximation of the parameters described above. This approximation is based on the fact that the selectivity of equality predicates is usually (very) high. Thus, we characterize the current state of the clustering instance as follows:

- For each cluster \( c \), the approximate benefit margin of \( c \) is denoted \( BM(c) \) and is equal to \( \nu(p_c) | c | \), where \( | c | \) is the size of the cluster.
- For each hash table \( H \), its approximate benefit is denoted \( B(H) \) and is equal to \( | H | \).

These metrics are used in the maintenance algorithm of our dynamic clustering matching solution to determine the heuristics that should be applied to decide when to redistribute the subscriptions of a cluster and when to create or delete a hash table. These heuristics are parameterized by three threshold values: \( BM_{\text{max}}, B_{\text{create}} \) and \( B_{\text{delete}} \). A cluster should be redistributed if its benefit margin is above \( BM_{\text{max}} \). The benefit margin of a cluster \( c \), \( BM(c) \), may increase for two reasons. First, a subscription may be inserted in \( c \). Second, the probability \( \nu(p_c) \) of the access predicate of \( c \) may increase, which implies that \( p_c \) has a higher probability of being verified by an event. A hash table is created when its benefit is above \( B_{\text{create}} \), and removed when its benefit is below \( B_{\text{delete}} \). The benefit of a hash table \( H \) may decrease when

\(^8\) If there is still one. Otherwise, the subscriptions cannot be redistributed, which, by the way, means that they are already in their best cluster.
subscriptions contained in clusters indexed in $H$ are deleted or are redistributed into other existing hash tables. For obvious reasons, the $B_{create}$ threshold must be greater than the $B_{delete}$ threshold.

These heuristics agree with our analysis of the matching cost made in Section 5.4.3. In fact, a hash table is created only if its benefit is higher than $BM_{max}$, i.e., the number of subscriptions that are better indexed in this hash table is high. Moreover, the maintenance algorithm only takes into account the subscriptions that are placed into large clusters and have a high probability of being accessed, i.e., clusters where the factor $n^k/(2 \times \alpha^j)$ is the dominant one. A high access probability means that $n^k$ is small, which usually implies that the subscriptions are being clustered using a small number of equality predicates, i.e., $k$ is small. Therefore, we can still use one or more equality predicates of the subscriptions to factorize them, i.e., increasing $k$, to improve their access probability. Finally, a hash table is deleted when its number of subscriptions is small.

7.4 The maintenance algorithm

When a subscription is inserted into the system, the insertion algorithm places the subscription in the best existing hash table. However, and since new hash tables can be created at any moment, it may happen that after some time, a subscription is no longer placed in the best existing hash table. For example, a subscription that has equality predicates on attributes $A$, $B$, $C$ and $D$ and is initially placed in the hash table with schema $AB$, will be better placed in the table with schema $BCD$ if $\nu(BCD, s) < \nu(AB, s)$. Since a subscription is always placed into the currently best hash table, when $s$ was placed in $AB$, either $BCD$ did not exist yet or $\nu(BCD, s)$ was greater than $\nu(AB, s)$. Otherwise, the subscription would have been initially placed in the hash table with schema $BCD$.

The assign.cluster and get.cluster functions applied in the context of the dynamic clustering matching solution are equal to those defined for the static clustering matching solution. We now present the cluster maintenance algorithm of the current clustering instance. This maintenance algorithm is responsible for maintaining the clusters. It tries to redistribute the subscriptions of a cluster when it becomes too large. In our implementation, the benefits of clusters and hash tables are updated periodically after a certain number of subscription changes and/or a certain number of incoming events.

The cluster maintenance algorithm takes into account the set of hash tables already created and a set of potential hash tables. A potential hash table represents a hash table that can be created when its benefit becomes higher than the threshold value $B_{create}$. This algorithm maintains, for each potential hash table $H$, a set of clusters and the corresponding hash tables containing candidate subscriptions that could be moved to $H$. A subscription $s$ is said to be candidate of a potential hash table $H$ if and only if the checking cost of $s$ in $H$ is smaller than the checking cost of $s$ in its current hash table. For performance reasons (in order to reduce the maintenance cost), we limited the set of potential hash tables considered for a subscription $s$ indexed in a hash table $H$ to those hash tables whose schema is a superset of the schema of $H$ plus one of the attributes concerned by the equality predicates of $s$ that does not belong to the access predicate of $s$. We say that a subscription $s$ of a cluster is marked if the cluster maintenance has already updated the benefit for all the potential hash tables which can have $s$ as a candidate subscription. This way, when the algorithm updates the potential hash tables concerned by the subscriptions of a cluster, it only needs to take into account those subscriptions that have not been marked yet. This option reduces the cost of this maintenance algorithm.

The cluster maintenance algorithm is detailed in Figure 11. It is applied to a cluster $c$ and its hash table $H_c$ whenever the benefit margin of $c$, $BM(c)$, is updated (e.g., a subscription is inserted into $c$). The cluster maintenance algorithm works as follows. The algorithm acts only if $BM(c)$ is above $BM_{max}$, otherwise it does nothing. When $BM(c)$ is too excessive (lines 8-28), the cluster maintenance algorithm first calls the cluster.distribute function (line 9) that tries to redistribute the subscriptions of $c$ into existing hash tables where the checking cost of the subscriptions is smaller. This function is explained in detail below. If after this redistribution, the benefit margin of $c$ is still excessive (lines 10-18), the cluster maintenance algorithm accesses each subscription of $c$ that has not been marked yet. For each of these subscriptions, the algorithm first marks it and then considers each potential hash table where the subscription could be better placed. For each of those potential hash tables (lines 12-16), the algorithm updates its benefit and its candidate clusters. Afterwards, the algorithm iterates over the existing potential hash tables while there is at least one whose benefit is greater than the threshold value $B_{create}$. In each iteration (lines 20-24), this algorithm picks up
cluster maintenance\( (c, H_c) \)

1. **input:** \( c \) the current cluster and its hash table \( H_c \).
2. **global variables:**
   - \( \mathcal{H} \): the set of hash tables;
   - \( PH \): the set of potential hash tables;
   - thresholds \( B_{max}, B_{delete}, B_{create} \).
3. **local variables:**
   - \( s \): a subscription;
   - \( H \): a potential hash table.

Begin Body

8. **if** \( BM(c) \geq B_{max} \) **then** // cluster \( c \) has an excessive benefit margin
9. cluster distribute\( (c, H_c) \)
10. **if** \( BM(c) \geq B_{max} \) **then** // The redistribution did not improve enough the Benefit margin
11. **foreach** subscription \( s \) in \( c \) such that \( s \) is not marked **do**
12. mark \( s \)
13. **foreach** table \( H \) in \( PH \cap GA(s) \) **do**
14. \( B(H) \leftarrow B(H) + 1 \)
15. \( H.candidate\_clusters \leftarrow H.candidate\_clusters \cup \{(c, H_c)\} \)
16. **endloop**
17. **endloop**
18. **endif**
19. while (Exists \( H \) in \( PH \) such that \( B(H) \geq B_{create} \) and \( \forall h \in PH B(h) \leq B(H) \)) do
20. \( \mathcal{H} \leftarrow \mathcal{H} \cup \{H\} \)
21. \( PH \leftarrow PH - \{H\} \)
22. **foreach** pair \( (c', H') \) in \( H.candidate\_clusters \) **do**
23. cluster distribute\( (c', H') \)
24. **endloop**
25. **endwhile**
26. **if** \( B(H_c) \leq B_{delete} \) **then**
27. remove table\( (H_c) \)
28. **endif**

End Body

**Fig. 11.** The cluster maintenance algorithm of the dynamic clustering matching solution. It reacts to changes of the benefit margin of a cluster.

the potential hash table \( H \) with the highest benefit and creates it. Then it populates \( H \) by applying the cluster distribute function to each candidate cluster of \( H \). These candidate clusters contain subscriptions that are better placed in \( H \). We would like to note that when clusters are redistributed, the benefit of other potential hash tables can decrease since a subscription can be candidate to more than one potential hash table. Finally, if due to the redistribution of subscriptions, the benefit of the hash table \( H_c \) is below \( B_{delete} \), then the remove table function is executed. This function removes \( H_c \) from the system and redistributes all of its subscriptions into other hash tables.

7.4.1 The redistribution of clusters

The goal of the cluster distribute function (called in lines 8 and 22 of Figure 11) is to redistribute the subscriptions stored into a cluster \( c \) (indexed in a hash table \( H_c \), into better existing hash tables. This function is represented in Figure 12 and works as follows. For each subscription \( s \) of \( c \), it computes the existing hash table \( H_{best} \) that minimizes the probability of \( s \) being checked (line 8). If \( H_{best} \) is not the current hash table \( (H_c) \) that indexes \( s \) (lines 9-19), then the benefit of \( H_c \) is decremented by one (line 10). This happens because \( s \) is moved to its new hash table \( H_{best} \) (line 19). Moreover, if the subscription has been already marked by the cluster maintenance algorithm (lines 11-17), this function considers each potential hash table \( H \) where \( s \) is a candidate subscription and updates the benefit and the set of candidate subscriptions of \( H \).
cluster_distribute(c, H_c)
1 input: c: the current cluster and its hash table H_c.
2 global variables:
3 H: the set of hash tables; PH: the set of potential hash tables
4 local variables:
5 H_best and H: hash tables; s: a subscription of c
6 Begin Body
7 foreach subscription s in c do
8 let H_best be the hash table in H such that \( \nu(H, s) \) is minimal
9 if H_best \( \neq H_c \) then
10 \( B(H_c) \leftarrow B(H_c) - 1 \)
11 if s is marked then
12 foreach table H in PH \( \cap GA(s) \) do
13 \( B(H) \leftarrow B(H) - 1 \)
14 H.candidate_cluster \( \leftarrow H.candidate_cluster - \{(c,H_c)\} \)
15 endloop
16 delete mark from s
17 endif
18 move s to H_best
19 endif
20 endloop
21 End Body

Fig. 12. The cluster_distribute function used to redistributed the subscriptions of a cluster.

7.4.2 Deletion of hash tables

When the benefit of an existing hash table H drops under threshold \( B_{delete} \), the maintenance algorithm simply redistributes all subscriptions indexed in H into other existing tables, and removes H from the current indexing configuration. This situation can happen when a subscription placed into H is removed from the system or after a redistribution of the subscriptions of a cluster indexed in H that can occur when the cluster_distribute function is executed (line 26 of Figure 11). This part of the maintenance algorithm is executed by the function remove_table represented in Figure 13.

remove_table(H)
1 input: the hash table H to remove.
2 global variables:
3 H: the set of hash tables;
4 local variables:
5 c: a cluster of H
6 Begin Body
7 \( H \leftarrow H - \{H_c\} \)
8 foreach cluster c in H do
9 cluster_distribute(c, H)
10 endloop
11 End Body

Fig. 13. The remove_table function that removes a hash table from the system. This function is called when the benefit of a hash table is bellow the \( B_{delete} \) threshold value.

First, the remove_table function removes the hash table to delete H from the current set of existing hash tables. Second, it redistributes all subscriptions of each cluster indexed in H using the function clus-
ter_distribute. Since $H$ was already removed from the current indexing configuration, the cluster_distribute function always finds a hash table, other than $H$, where each subscription of $H$ should be placed.

7.4.3 Maintenance cost

Besides the cost of maintaining each hash table and potential hash table, the maintenance cost is proportional to the number of subscription moves. When a new subscription $s$ arrives, the insertion algorithm always chooses the hash table that gives the best absolute benefit for $s$. However, $s$ may be moved to another hash table during its lifetime if insertions or changes in event statistics increase the benefit margin of the cluster of $s$, and trigger the creation of a hash table that is better for $s$.

The choice of the threshold values clearly influences the number of subscription moves. Indeed, $B_{create}$ influences the number of hash tables created. $BM_{max}$ influences the number of subscriptions moved and also the number of candidate subscriptions for potential hash tables. Therefore, it also influences the number of hash tables created. If these two values are too low, we have a high maintenance cost because the algorithm considers those subscriptions that are placed in small clusters or in clusters that are rarely accessed. Moreover, we may have hash tables that are not worthwhile since their number of subscriptions is too small. However, if $BM_{max}$ and $B_{create}$ are too high, we have a low maintenance cost but we may have a higher matching cost. In this case, the dynamic clustering matching solution may not be able to create multi-dimensional hash tables that are profitable, because the number of subscription candidates is not enough to trigger their creation. This happens because $BM_{max}$ and/or $B_{create}$ are too high. We have made some experiments to tune the values for the three thresholds used in this matching solution.

8 Performance Evaluation

This section presents several experiments that evaluate the behavior of our matching algorithms. In each experiment, we isolate one parameter of a dimension and study its influence on the behavior of the matching algorithms. We are interested in studying the following kinds of behaviors of matching algorithms: (i) the average time to match an event against the current set of subscriptions, that is measured by the event maximal throughput supported by the matching algorithm; (ii) the maintenance time, that is measured by the time to load subscriptions in the system; and (iii) the space occupied by the data structures used by a matching algorithm to hold subscriptions. The parameters that we consider for each dimension of the data flow used in our experiments are: the number of subscriptions and the number of predicates per subscription (data volume), the number of subscription schemas (subscription distribution) and the number of values per attribute domain (cardinality of attribute domains). We also study the variation of the performance of the dynamic and static matching algorithms in a context where subscriptions are inserted and removed simultaneously and the characteristics of the data flow can evolve over time.

For comparison reasons, we also show the behavior of the counting algorithm which is used, with small variations, in several systems. A first version was presented in the SIFT system [Yan and Garcia-Molina, 2000], where it was applied in for matching text documents (i.e., the events) against keyword-based profiles (i.e., the subscriptions). A version of the counting algorithm applied in a context where events are defined as attribute-value pairs and subscriptions are conjunctions of predicates was presented in [Pereira et al., 2000]. The algorithm used by the SIENA distributed P/S system to forward events between brokers is based on this version of the counting algorithm [Carzaniga et al., 2001]. A less efficient version of the algorithm is used in the NEONet system [Piskiel et al., 1999]. The experiments were made using an implementation of the version of the counting algorithm presented in [Pereira et al., 2000]. This algorithms works as follows. First, it computes the predicates matched by an event. Then, it counts the number of predicates satisfied for each subscription. Finally, it computes the subscriptions verified by an event. This is achieved by comparing for each subscription its number of predicates with its number of matched predicates. If both numbers are equal then subscription is matched.

8.1 Experimental setup

Subscriptions and events are drawn randomly according to a workload specification that determines subscriptions, predicates, events, and attribute names.
If we require that certain attributes appear in all subscriptions generated according to a given workload specification, we call such attributes "common attributes" for the subscription set. A predicate is fixed in a set of subscriptions if its attribute is a common attribute. A subscription workload specifies the total number \( n_S \) of subscriptions to generate; a batch size \( n_{Sb} \), that determines the number of subscriptions to submit to the system at once; the number of predicates \( n_P \) per subscription; the number of predicates \( n_{Pfix} \) fixed per subscription (broken down into \( n_{Pfix1} \), \( n_{Pfix2} \), and \( n_{Pfix3} \), i.e., the number of predicates with the corresponding comparison operators), and a predicate workload specification. The number of subscriptions and the number of predicates per subscription are the parameters that we use to specify the characteristics of the data volume dimension of the data flow submitted to the matching algorithms. In our experiments, we submit the matching algorithms to a large number of subscriptions (several millions) and we vary the number of predicates per subscription from two to twelve.

Predicates are determined by a name, a comparison operator, a value domain, and the domain’s cardinality. Predicate names are drawn from the predefined set of attribute names. The same set of attribute names is used to draw attribute names for events. The total number of names available is determined by \( n_t \). The value domain determines all the possible values of a predicate and is specified with a lower and upper bound, \( l_P \) and \( u_P \), respectively. Values are drawn from this domain according to a uniform distribution. The number of values per attribute domain specifies the cardinality of the attribute domain dimension. By setting different lower and upper bounds for each predicate value domain we can simulate the subscription predicate data skew (in the following referred to as subscription skew) using several workload experiments for the same experiment.

The workload parameters that fix a certain number of predicates in all subscriptions, the number of predicates per subscription and the existing number of attributes determine the maximal number of distinct subscription schemas that are generated on average. These three parameters define the subscription distribution dimension. The maximal number of existing subscription schemas with a given number \( n_{atts} \) of attributes is given by:

\[
\frac{(n_t-n_{Pfix})!}{(n_t-n_{atts}-n_{Pfix})!(n_{atts})!}, \quad 1 \leq n_{atts} \leq n_P - n_{Pfix}.
\]

In most of the experiments we have run, we use a subscription distribution with a relatively high number of attributes (16 to 32), where the number of fixed predicates per subscription is small (one or two), and the number of predicates per subscription is usually four. With such a specification, subscriptions are scattered among a large number of schemas.

Analogously to subscriptions, events are determined by the number of events \( n_E \) to generate, the batch size \( n_{Eb} \) of events to submit to the system at once, the number of attribute-value pairs \( n_A \) within events, and the value domain, determined by a lower and an upper bound, \( l_A \), \( u_A \), respectively. Values are drawn uniformly distributed from this domain. For all experiments, we use intervals of positive integers as value domains. By setting different lower and upper bounds for each attribute domain we can simulate the event attribute data skew (in the following referred to as event skew). In most experiments, the generated events always define a value for every attribute considered in the experiment’s workload (i.e. \( n_t \) is equal to \( n_A \)), which from the matching algorithm point of view, is the worst case-scenario since an event can potentially match any subscription independently of the attributes referred in the subscription.

Table 1 summarizes the workload specification parameters and their values for our experiments.

### 8.1.1 Execution of the experiments

We ran all experiments on a PC, with a single CPU-kernel and 1GB RAM, operating under Linux. In all experiments, we used our P/S prototype to study the behavior of the four matching algorithms we are interested in. In our prototype, the matching algorithm is executed as a different process, called matching process. However, this process runs in the same machine as the P/S system, which is called the system process. We implemented a workload generator that emits subscriptions and events according to a workload specification. The subscriptions and events are then submitted to the P/S system. The workload generation task also ran as a separate process, called the workload generator process, on the same workstation as the other two processes. Subscriptions and events are emitted to the P/S system in fixed-size batches. The batch size is set in the workload specification. Timings are taken in milliseconds, starting just before the P/S system sends the generated subscriptions and events to the matching process, where they will be processed. The timings finish just after the system process receives the answer from the matching process. The timings include therefore the interprocess communication times and the time required for processing an entire batch
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_t$</td>
<td>total number of predicate/attribute names</td>
<td>$8-32$</td>
</tr>
<tr>
<td><strong>Global parameters</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_S$</td>
<td>total number of subscriptions</td>
<td>$100,000-7,000,000$</td>
</tr>
<tr>
<td>$n_{S_b}$</td>
<td>number of subscriptions to submit to the system at once</td>
<td>$10,000$</td>
</tr>
<tr>
<td>$n_P$</td>
<td>number of predicates per subscription</td>
<td>$4 - 12$</td>
</tr>
<tr>
<td>$n_{P_{fix}}$</td>
<td>number of predicates fixed per subscription</td>
<td>$0 - 2$</td>
</tr>
<tr>
<td>$l_{P_i}, u_{P_i}$</td>
<td>limits of value domain of predicates (per predicate $i$)</td>
<td>$5 - 1024$</td>
</tr>
<tr>
<td><strong>Subscription and predicate parameters</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_A$</td>
<td>number of attribute value pairs per event</td>
<td>$8-32$</td>
</tr>
<tr>
<td>$l_{A_i}, u_{A_i}$</td>
<td>limits of value domain of attributes</td>
<td>$5 - 1024$</td>
</tr>
<tr>
<td><strong>Event parameters</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Parameter definitions and range values.

of subscriptions or events submitted. The matching process responds to event and subscription submissions with the corresponding set of matched subscription identifiers and set of identifiers of the new subscriptions assigned by the matching algorithm, respectively.

Besides subscriptions and events, the workload generator process can also generate lists of subscriptions previously added to the system that should be removed from the system. Each of these lists contains the identifiers of such subscriptions. This feature of the workload generator process allows us to submit a subscription change rate to the P/S system.

We ran the various experiments multiple times and did not notice a significant difference in the results. For this reason, we do not report variances in our figures, which were lower than 0.1%, for the experimental runs repeated.

### 8.2 Influence of the data volume dimension

This section presents several experiments that show how the behavior of the matching algorithms was affected by the following parameters of the data volume dimension: the number of subscriptions, the number of predicates per subscription and the type of predicates (equality vs non-equality).

#### 8.2.1 Number of subscriptions

The goal of this experiment is to study the impact of the number of subscriptions on the behavior of our matching algorithms. This experiment uses the following workload specification $W_0 : \{ n_t = 32; n_P = 5 \text{ (2 fixed, all equality)}; n_A = 32; \text{value domain: } (l = 1, u = 35) \text{ (no skews)}; n_{S_b} = 10,000; n_{E_b} = 100\}$. The number of subscriptions varies from $100,000$ to $7,000,000$.

This section is organized as follows. First, we show the event maximal throughput for each matching algorithm, i.e., the maximal number of events per second that is supported by each matching algorithm. Second, we present the memory utilization for each matching algorithm. Third, we show the loading time for each matching algorithm.

**Maximal event throughput** Figure 14 compares the system event throughput for all matching algorithms. As expected, the dynamic and static algorithms show the best performance, while the counting algorithm has the poorest one. The performance of the two propagation algorithms lies in between the counting and the dynamic algorithms. The propagation, static and dynamic algorithms are much better than the counting algorithm since they only consider subscriptions whose access predicates (single predicate in propagation and multiple predicate in static and dynamic) are satisfied while the counting algorithm considers all subscriptions which have at least one satisfied predicate. For instance, the event throughput
of our system when loaded with 7,000,000 subscriptions is 0.96 events/s (counting), 127.3 events/s (propagation), 224.7 events/s (propagation with prefetching), 1755.9 events/s (dynamic). With this configuration, 56% of the time needed to compute the subscriptions matched by a batch of events (100) in the dynamic algorithm is spent in communication between the two processes (i.e., the system and matching processes). We were not able to execute the static algorithm for values higher than four million subscriptions because it is not possible to compute the best clustering instance, as we explain later in this section.

Fig. 14. Event maximal throughput for the several matching algorithms when the number of subscriptions in the system varies. Note that the y-axis is in a logarithmic scale.

A notable feature of the dynamic and static algorithms is the fact that the throughput of the system has a very low sensitivity with the number of subscriptions. This nice behavior is ensured by using hash tables with a higher dimensionality when the number of subscriptions increases. The difference between the dynamic and static algorithms is small, less than five percent, and the static algorithm is always the best one. This small difference of performance is due to the fact that both algorithms produce similar indexing configurations. In this experiment, the indexing configurations produced by both algorithms differ at most in two hash tables. This shows that the metrics used by the dynamic algorithm to decide on the hash tables that should be created provide a good approximation of clustering benefits.

By comparing the two versions of the propagation matching algorithm, with and without the prefetching technique, we verify that the version that uses prefetching, called the propagation_wp algorithm, is always better than the one that does not use prefetching, called the propagation algorithm. The improvement increases with the number of subscriptions since the average size of the subscription clusters also increases with this factor. For 7,000,000 subscriptions, the prefetching technique improved the performance of the propagation algorithm by a factor of 1.76. This proves that the use of the prefetching technique can really improve the performance of the matching algorithm.

8.2.1 Memory utilization Figure 15 shows the memory utilization required by all algorithms to store the internal data structures used to index the subscriptions. All curves follow the natural intuition, that is memory use increases with storage requirements.
The two versions of the *propagation* matching algorithm only differ on the use of the prefetching technique. Both versions use the same internal data structures and organize the subscriptions in the same way. This means that the memory utilization in the two versions of the *propagation* matching algorithm is exactly the same. For this reason, we show only the memory utilization for the *propagation* algorithm in Figure 15 (and also in Figure 16 for the same reason). For all algorithms, the memory utilization increases linearly with the number of subscriptions in the system. The *propagation* algorithm requires the least amount of memory, closely followed by the *counting* algorithm, while the *dynamic* and *static* algorithms require the most. There is a memory overhead in the *dynamic* and *static* algorithms due to the use of multi-dimensional hash tables and the definition of more subscription clusters\(^9\). The *dynamic* and *static* algorithms have a similar memory utilization since they define subscription clustering instances that are very similar.

8.2.1.2 Subscription loading time

Figure 16 shows the time needed by each matching algorithm to load a given number of subscriptions in the system. The system initialization time is the smallest for the *counting* algorithm, which deploys very simple data structures, and the highest for the *static* algorithm, that statically computes from scratch an optimal clustering configuration.

The *static* algorithm exhibits a very high sensitivity to the number of subscriptions in what concerns the time needed to compute the optimal clustering and the amount of memory needed to compute it. We were not able to compute the optimal clustering for 5 million subscriptions due to the large amount of memory needed to compute the optimal clustering (more than 1 GB). Compared to the *static* algorithm, the *dynamic* algorithm significantly improves the loading time through the incremental reorganization of its internal data structures. This reorganization is performed to best suit the subscriptions encountered thus far. The *propagation* algorithm takes less time than the *dynamic* one due to the extra time spent by the *dynamic* algorithm for maintaining the information about the potential hash tables and for reorganizing the subscriptions whenever a potential hash table is promoted, i.e., a new hash table is created.

\(^9\) Notice that each cluster has a fixed cost in space.
In the rest of our experiments, we use a single version of the *propagation* algorithm, the one which uses prefetching. The other version was only used to show the gain in performance that could be obtained by using prefetching. The *static* algorithm is not shown either since its performance is very similar to the performance of the *dynamic* algorithm.

### 8.2.2 Number of predicates per subscription

This section presents two experiments that show how the number of predicates per subscriptions affects the performance of the matching algorithms. The first experiment varies the number of equality predicates per subscription while the second one varies the number of non-equality predicates per subscription.

#### 8.2.2.1 Varying the number of equality predicates per subscription

This experiment studies the impact of the number of equality predicates per subscription on the performance of each matching algorithm. It applies the following workload specification: \( n_t = 32; n_A = 32; \) value domain: \( l = 1, u = 5; \) \( n_S = 4,000,000; n_{P_{fix}} = 2; n_{S_b} = 10,000; n_{E_b} = 100 \). The experiment varied \( n_P \) (i.e., the number of predicates per subscription) from four to twelve. Figure 17 shows the result of this experiment. We remark that the *counting* algorithm is again the most affected algorithm. In fact, the performance of this algorithm decreases with \( n_P \). This happens because as the number of predicates per subscription increases, the list of satisfied predicates that must be processed by the matching algorithm also increases\(^{10}\).

The *propagation* algorithm is not sensitive to the number of equality predicates per subscription. The reason for this lies in the fact that in this configuration, the cost of processing an extra predicate array line is compensated by having a smaller number of matched subscriptions, as we explain next. On one hand, when the number of predicates per subscription increases, the number of *predicate* arrays of the subscription clusters also increases. Therefore, the number of cache misses that occur when checking the predicates of subscriptions increases with the number of predicates per subscription. On the other hand,

\(^{10}\) Recall that the *counting* algorithm must count the number of times each subscription appears in this list.
increasing the number of predicates decreases the average number of subscriptions matched by an event. The cost of storing the identifier of a matched subscription in the result returned by the matching algorithm is similar to a cache miss since the processor has always to write the new value to store in the RAM memory, and not in the cache. This way, the number of cache misses that occur when the identifiers of the matched subscriptions are stored in the result of the matching algorithm decreases. In the subscription configuration used in this experiment, the extra processing and number of cache misses involved when processing subscriptions with more predicates is compensated by the number of saved cache misses due to a lower number of matched subscriptions. Finally, we would like to remark that the performance of the propagation algorithm decreases when the number of predicates is greater than 10. This happens because the function that checks the subscription clusters is the generic cluster checking function (that works for any number of predicates per subscription), and not a specialized one as is the case for lower numbers of predicates.

The dynamic algorithm is the only one whose performance improves with $n_P$, because it is able to index the subscriptions in hash tables with a higher dimensionality when the parameter $n_P$ increases. Indeed, the number of subscriptions per schema increases with the number of equality predicates per subscription (i.e., $n_P$). This way, the dynamic algorithm is able to create multi-dimensional hash tables with a higher dimensionality when $n_P$ increases, since their benefit, i.e., the number of subscriptions better placed in the hash tables, increases with $n_P$. The cost of checking a subscription cluster is lower since the average number of subscriptions per cluster decreases with the dimensionality of the hash tables (since all attributes have the same cardinality).

8.2.2.2 Varying the number of non-equality predicates The next experiment compares the event throughput of the dynamic, propagation and counting algorithms for different kinds of comparison operators in predicates. This experiment uses three different specification workloads, $W_2$, $W_3$ and $W_4$, which only differ on parameters $n_{P_{\leq}}$ (number of fixed less than predicates) and $n_P$ (number of predicates per subscription). The $W_2$ workload specification was set as follows: $\{n_S = 3,000,000; n_t = 32; n_A = 32; n_P = 4; n_{P_{\leq}} = 2; n_{P_{\geq}} = 1\}$ and one non-fixed predicate with equality operator, chosen freely among
the \( n_t \) unused predicate names; value domain: \( \{ l = 1, u = 35 \} \). For workload specifications \( W_3 \) and \( W_4 \), the parameters \( n_{P_{fix}} \) and \( n_P \) were set to 2 and 6 and 5 and 9, respectively.

This experiment represents the performance of the matching algorithms through the average time to process an event which is the dual of the event throughput. Due to the great difference of values obtained for the counting algorithm and the other two matching algorithms, we present the results of this experiment in two different figures, Figures 19 and 18. The results show that all algorithms are sensitive to non-equality predicates. Again, the counting algorithm, represented in Figure 18, is the most sensitive one. This is mainly due to the low average selectivity (0.5) of non-equality predicates. Therefore, when the number of non-equality predicates per subscription increases, the average number of satisfied predicates per subscription also increases. Since the counting algorithm determines the number of satisfied predicates per subscription, the more non-equality predicates each subscription has, the more amount of processing the counting algorithm performs, and consequently the poorer is its performance.

**Fig. 18.** Event processing time for the counting algorithm under workload specifications \( W_2, W_3 \) and \( W_4 \). The counting\_W2, counting\_W3 and counting\_W4 curves represent the performance of the counting algorithm under the workload specifications \( W_2, W_3 \) and \( W_4 \), respectively.

Figure 19 shows the results of this experiment for propagation and dynamic algorithms. The time required by the propagation and dynamic algorithms to process an event decreases by a constant factor as more non-equality predicates (i.e., \( W_2 \) vs. \( W_3 \) and \( W_3 \) vs. \( W_4 \)) are being processed. In this experiment, when the number of non-equality predicates per subscription increases, both the number of predicate families and the average number of non-equality predicates satisfied by an event increase. Therefore, the processing cost of the first phase of our matching algorithms, i.e., the computation of the satisfied predicates, is higher in \( W_3 \) (\( W_4 \)) than in \( W_2 \) (\( W_3 \)) as more non-equality predicates (and predicate families) are being generated in the workload. The difference in time to process an event is independent of the number of subscriptions due to two reasons. First, the cost of the first phase of the matching algorithm does not depend on the number of subscriptions, and varies only with the number of existing predicate families and the number of existing predicates. Second, as we have seen in the previous experiment, varying the number of predicates per subscription may not affect the performance of the dynamic and propagation algorithms.
Notice that this experiment does not change the number of equality predicates per subscription, and that the dynamic algorithm only considers the equality predicates of subscriptions to dynamically create the hash tables. Therefore, increasing the number of non-equality predicates per subscription does not affect the hash tables dynamically created by this algorithm.

Finally, we would like to remark that the difference in time to process an event as we increase the number of non-equality predicates is equal in both matching algorithms. This is due to the fact that both matching algorithms use the same cluster checking algorithm to handle non-equality predicates. In the cluster checking algorithm, bit vector entries associated to non-equality predicates of a given subscription $s$ are checked only if all equality predicates of $s$ are verified. Since the two algorithms are tested under the same subscription workload, the probability that such situation arises is the same for both of them. The dynamic algorithm performs better because it handles the equality predicates via multi-dimensional hash tables.

8.3 Influence of the cardinality of attribute domains dimension

The next experiment studies how the cardinality of attribute domains dimension of the data flow influences the performance of our matching algorithms. We ran an experiment that increased, doubling each time, the cardinality of the domain of the first fixed predicate of each subscription. This cardinality ranges from 32 to 1024. The cardinality of the domains of the other attributes remained always equal to 32. The other parameters of the workload specification used in this experiment are: $n_t = 32, n_P = 4$ (2 fixed predicates, all predicates are equality predicate); $n_A = 32, n_S = 4,000,000, n_{S_0} = 10,000, n_{E_k} = 100$. We measured the performance of the counting, dynamic and propagation matching algorithms for this subscription workload and the results obtained are presented in Figure 20.
The performance of the *counting* matching algorithm improves very slightly with this factor. The improvement of its performance is only around 10% when the cardinality of the domain of one of the two *fixed* attributes varies from 32 to 1024. This happens because this algorithm depends mainly on the number of subscriptions in the system and as well as on the *average* selectivity of all predicates. This average selectivity only increases slightly when we vary the cardinality of the domain of a single (*fixed*) attribute.

The other two matching algorithms are both sensitive to this factor, by taking advantage of a more selective predicate to reduce the number of subscriptions that must be checked during the matching of an event. The *propagation* matching algorithm is more sensitive to the selectivity of predicates than the *dynamic* matching algorithm. In fact, the influence of a high selective predicate in each subscription is higher when the matching algorithm indexes subscriptions using a single predicate, as is the case of the *propagation* matching algorithm, than when the matching algorithm indexes subscriptions using multiple predicates, as is the case of the *dynamic* matching algorithm.

We now explain more precisely why the *propagation* matching algorithm is more sensitive to the cardinality of attribute domains than the remaining matching algorithms. The subscription cluster average size is greater in the *propagation* matching algorithm than in the *dynamic* matching algorithm, since the former one clusters subscriptions using a single predicate while the second one clusters subscriptions using multiple predicates. Considering the specification workload used in this experiment, the predicates that concern the attribute whose cardinality varies are always part of the access predicate of a cluster in both matching algorithms. This way, when we increase the cardinality of the fixed attribute, the cluster average size decreases by the same factor in both matching algorithms. However, the average number of subscriptions per cluster in the *propagation* matching algorithm is superior to that number in the *dynamic* matching algorithm. Therefore, when the cardinality of the domain of the fixed attribute increases, the variation of the number of subscriptions that are checked to determine those subscriptions matched by an event is higher in the *propagation* matching algorithm than in the *dynamic* matching algorithm. This fact explains

---

Fig. 20. Influence of the selectivity of the first subscription predicate in event throughput. Notice that the y-axis is in logarithmic scale.

11 Recall that the number of subscriptions checked by the matching algorithm influences the cost of event matching. The matching cost of the *propagation* and *dynamic* matching algorithms increases with this number.
why the difference of performance for the two matching algorithms is inferior to 20% when the domain cardinality of the fixed attribute is equal to 1024. In this case, the average number of subscriptions in a cluster in the propagation matching algorithm is already small and therefore the gain obtained by using multi-dimensional indexes is small.

8.4 Influence of the subscription distribution dimension

This section investigates the impact of the subscription distribution dimension of the data flow on the performance of the matching algorithms. The experiments shown in this section vary the number of subscription schemas and the number of subscriptions per schema, i.e., the distribution of the subscription schemas. Intuitively, only those matching algorithms that use the schema of subscriptions to decide on the organization of the subscriptions into the internal data structures (for example, the dynamic matching algorithm), should be affected by this dimension. Matching algorithms, like the counting and propagation, that index the subscriptions independently of their schemas should not be affected by this dimension. Therefore, the performance of such algorithms should not be affected when we only vary the distribution of the subscription schemas and maintain all the other properties of the data flow, e.g., the number of predicates per subscription or the cardinality of attribute domains.

As stated above, varying the number of subscriptions per schema has a direct impact on the performance of the dynamic matching algorithm (as well as the static matching algorithm). In fact, in the dynamic matching algorithm, the decision to create a multi-dimensional hash table \( H \) depends on the number of subscriptions placed in beneficial clusters whose schema contains the schema of \( H \) and that would be better indexed in \( H \) than in their current hash table. The less the subscriptions are scattered, i.e., the higher is the average percentage of subscriptions per existing schema, the earlier the dynamic matching algorithm is able to create multi-dimensional hash tables. In other words, as the percentage of subscriptions per subscription schema increases, the number of subscriptions required by the dynamic matching algorithm to be able to create a multi-dimensional hash table with a given schema decreases. Moreover, multi-dimensional hash tables contain a higher number of subscriptions when subscriptions are less scattered than when subscriptions are more scattered. Indeed, the number of subscriptions that share a certain combination of attributes is higher in the former case (less scattered) than in the second case. For this reason, the dynamic matching algorithm is able to index the current set of subscriptions with a smaller number of multi-dimensional hash tables or/and use hash tables with a higher dimensionality when subscriptions are less scattered.

This experiment studies the influence of the number of attributes on the performance of the matching algorithms. It applies the following workload specification: \( \{ n_S = 3,000,000; n_A = n_t; n_P = 4; n_{P_{clean}} = 1; \text{value domain: } \{l = 1, u = 35\}\} \). The experiment varied the number of existing attributes from 8 to 32. Notice that by varying the number of existing attributes, the number of subscriptions per existing subscription schema also changes. This happens since each subscription has three predicates whose attributes are chosen among the \( n_t \) minus one existing attributes (one of the existing attributes is used in the fixed predicate of the subscriptions). Therefore, when we increase \( n_t \), the number of existing subscription schemas increases and the number of subscriptions per schema decreases. The results of this experiment, presented in Figure 21, show that the performance of the dynamic matching algorithm depends on \( n_t \), while the performance of the counting and propagation matching algorithms is only slightly affected by \( n_t \). We have not shown the counting matching algorithm in this figure for the sake of presentation. The variation of the performance of the counting matching algorithm obtained in this experiment was inferior to one percent. For \( n_t \) equal to 32, this algorithm was able to process 2.22 events per second.

The counting and propagation matching algorithms see their performance slightly worse with \( n_t \). This drop of performance is due to two distinct factors that are not directly associated with the subscription distribution dimension. First, the timings measured in our experiments include the interprocess communication time between the system process and the matching process\(^\text{12}\). The time spent to transmit the events from the matching process to the system process increases with \( n_t \), because the number of attribute-value pairs of an event is equal to \( n_t \). This way, when \( n_t \) increases, the size in bytes (and the transmission time) of the batch of events sent each time from the system process to the matching process also increases. Second,

\(^{12}\) Recall that these two processes execute our P/S system prototype and the matching algorithm, respectively.
Fig. 21. Performance of the propagation and dynamic matching algorithms when the number of existing attributes in the systems varies ($n_S = 3000000$).

the number of predicate families also increases with $n_t$, which implies that the cost of computing the predicates satisfied by an event increases with $n_t$. Since the predicate families in this experiment only concern equality predicates and we use hash tables to index them, the extra time required to compute the satisfied predicates due to a higher number of predicate families is small.

As expected, the dynamic matching algorithm is sensible to the number of existing attributes ($n_t$) since the subscription distribution dimension varies with this factor. When we increase the number of attributes, subscriptions become more scattered and the performance of the dynamic matching algorithm becomes worse. For instance, the number of existing subscription schemas with two attributes is 7 and 31 when $n_t$ is equal to 8 and 32, respectively. The average number of subscriptions per existing schema with two attributes is 1,286,000 in the first case, and 290,000 in the second case. If we consider the existing schemas with three attributes, the number of schemas and the respective average number of subscriptions are 21 and 428,571, and 465 and 19,354 when $n_t$ is equal to 8 and 32, respectively. In the first case ($n_t = 8$), we are able to index most subscriptions using three predicates, while in the second case it is worthwhile to index the subscriptions using only two predicates. This way, the dynamic matching algorithm indexes subscriptions with a lower number of hash tables and using hash tables with a higher dimensionality (which implies that the average size of the clusters is smaller) in the first case than in the second case. This fact explains the difference of performance between both cases.

We also made another experiment to investigate the impact of the number of fixed predicates on the performance of the matching algorithms\textsuperscript{13}. When the number of fixed equality predicates decreases the subscriptions are more scattered. The results were similar to the ones shown for the previous experiment. The performance of the propagation and counting matching algorithms is not affected by the number of fixed predicates (the number of predicate families does not vary in this experiment, which was not the case in the previous experiment). The dynamic algorithm is sensitive to the number of fixed predicates. When the number of fixed predicates decreases, its performance drops since the subscriptions become more scattered.

\textsuperscript{13} Recall that the number of fixed predicates specify the number of attributes that are referred in all subscriptions submitted to the system.
8.5 Adaptable to subscription and event skews

Under real world constraints, P/S systems deployed on the Web are likely to be subjected to a constant stream of subscription updates (e.g., modifications, insertions, and deletions) and events. The distribution of structure and content value of subscriptions and events is likely to change over time. Certain similarity patterns within neighboring elements in the streams may be observable. Subscriptions and events may, for instance, change in terms of their predicates’ domains. Our dynamic matching algorithm aims to handle these situations. In order to study its adaptive behavior in such a context and compare it with the static matching algorithm, we simulate these conditions in a set of experiments.

In these experiments, we consider situations where the P/S system has to handle concurrently incoming events and a high rate of incoming subscriptions. We assume subscriptions have a live time of about 16 hours. Given a subscription rate of 50 subscription insertions per second, the system will have to process roughly three million\(^{14}\) insertions before aging subscriptions begin to be deleted from the system. When the system reaches this point, where the number of deletions balances the number of insertions, we say the system reaches equilibrium. In the following experiments, we investigate the behavior of the static and dynamic matching algorithms when the system is at the equilibrium point. In the experiments shown in this section, the system is first populated with three million subscriptions according to a workload specification and the corresponding best clustering instance is computed. Both the static and dynamic matching algorithms are then initialized with this best clustering instance. At this state (i.e., equilibrium point), we remove 50 subscriptions (representing the 50 oldest ones, inserted 16 hours before) and insert 50 new subscriptions every second. If the system can manage these insertions and deletions in less than one second, we use the remaining time, before the following tick, to send events to the system. We measure the number of events the system can handle within the remaining time.

In the experiments reported in this section, we measure the system evolution according to two distinct application scenarios where subscription and event patterns are changing. In the first scenario, after the system has reached the equilibrium point, we vary the subscription schema of the new subscriptions submitted to the system. In the second scenario, we impose a subscription and event skew after the equilibrium point.

8.5.1 Schema subscription change scenario

The first experiment, whose results are presented in Figure 22, investigates the impact of subscription schema changes. This experiment models a situation where subscriber subjects of interest are changing over time. We start with the following workload specification \(W_S: \{n_t = 16; n_S = 3,000,000; n_P = 5; n_{P,i=1} = 2; n_A = 32; l_P = l_A = 1; u_P = u_A = 35\}\), where all of the 3,000,000 subscriptions focus on 16 of the 32 attributes available in the system and events provide uniform values for the 32 attributes. At the equilibrium point, we use a clustering configuration that it is optimal for \(W_S\). During the first hour, subscriptions and events are following \(W_S\) workload specification. Then, we insert subscriptions according to a new \(W_9\) workload specification similar to \(W_S\), except that it focuses on the 16 attributes that are not addressed in \(W_S\). After 16 hours, the system reaches the state where all subscriptions in the system are following \(W_9\). We continue to run the experiment during one hour, inserting and deleting \(W_9\) subscriptions. We call this interval of time the test period. Figure 22 shows the evolution of the average event throughput of the system over time (throughput is averaged every hour) and compares two opposite strategies for clustering maintenance: (i) the dynamic strategy used by the dynamic matching algorithm, that adapts the current clustering to subscription changes by creating (and deleting) hash tables on-the-fly; and (ii) the static strategy, used by the static matching algorithm, that does not change the initial (optimal) clustering configuration, i.e., the maintenance algorithm does not create (or remove) hash tables while subscriptions are being inserted and deleted.

This experiment shows that the static strategy is vulnerable to performance degradation when subscriptions’ schemas are changing. At the end of the test period, the event throughput is almost one fourth of what it was at the beginning. On the other hand, the dynamic strategy adapts the subscription clustering to the new situation. In the last hour, when subscription patterns are stable again, the system can handle 1685 events per second with the dynamic matching algorithm, instead of 461 events per second achieved with

\(^{14}\)16 * 3600 * 50 subscriptions/s = 2,880,000 subscriptions.
the static matching algorithm. During the transition phase, the performance of the dynamic matching algorithm is slightly inferior to the performance achieved by this algorithm when the subscription patterns are stable (the difference is never greater than 100 events per second). This difference is due to the additional maintenance cost incurred when new hash tables are created. The results obtained show that this cost is small and is more than compensated by the matching benefit of the new tables.

8.5.2 Event and subscription skew scenario

The last experiment investigates the impact of subscription skew when it is combined with event skew. This experiment models a situation where an area of interest is raised for both subscribers and publishers. Typical examples arise in news dissemination systems, for example a few days before the election of the US president, everybody may want to know about the candidates. At the same time, more and more information is published on this subject. To model this phenomenon we designed the following experiment. The subscriptions and events are initially generated according to the following workload specification $W_{10} : \{ n_t = 32; n_S = 3,000,000; n_P = 5; n_{P_{1ax}} = 2; n_A = 32; l_{p_{1}} = l_{A} = 1; u_{p_{1}} = u_{A} = 35 \}$, where equality predicates and event attribute values are uniformly distributed among 35 values. Once the system reaches the equilibrium point, the subscriptions and events submitted to the system follow the $W_{10}$ workload specification during the next hour. After this hour, we create both an event skew and a subscription skew. New events and subscriptions are submitted to the system according to a new $W_{11}$ workload specification. $W_{11}$ is similar to $W_{10}$ except that there is a skew (5 different values instead of 35) on event attribute and predicate values of one of the two fixed attributes used by subscriptions. After 16 hours, the system reaches the state where all subscriptions in the system are following $W_{11}$. We still run the system for one hour further inserting and removing $W_{11}$ subscriptions. Figure 23 shows the evolution of the average event throughput over time (averaged every hour) when using the dynamic and the static matching algorithms.

This experiment shows that the static matching algorithm does not prevent performance degradation in the presence of a skew in the distribution of event and subscription values. By the end of the test period, the event throughput of the static matching algorithm has been reduced by 20% compared to the performance achieved by the dynamic matching algorithm. The dynamic matching algorithm has a better performance
because it uses a maintenance strategy that adapts the current subscription clustering instance to the new situation. Nevertheless, at the end of the test period, even the dynamic matching algorithm cannot manage the same throughput as before. Its performance is reduced by 20%. This drop in performance occurs because there are more subscriptions matched under workload $W_{11}$ than under workload $W_{10}$ due to the subscription and event skew. This incurs in an additional cost (to store the identifiers of the matched subscriptions) that cannot be compensated by a clustering reorganization. At the beginning of the transition phase, and in what concerns the dynamic matching algorithm, the cost of maintaining the subscription clustering instance remains slightly preponderant compared to the matching benefit. But after five hours, the matching benefit obtained by clustering reorganization overcomes the maintenance cost. Notice that the drop in performance of the static matching algorithm is not so drastic as in the previous experiment, since the multi-dimensional hash tables created in the first part of the experiment can still be used to store the new subscriptions added during the second part of the experiment. Despite the fact that the initially computed hash tables do not correspond to the best indexing configuration, they provide nevertheless a great improvement compared to the situation where subscriptions are indexed using a single predicate.

However, we would like to remark that in this configuration, about half of the time needed to process a set of events is spent in interprocess communication. Moreover, about half of the time needed by the matching process to process the events is spent in the first step of the matching algorithm that computes the predicates satisfied by an event. Therefore, the static matching algorithm spends 80% more time than the dynamic matching algorithm to access the relevant subscription clusters and determine the subscriptions of these clusters that are matched by an event.

9 Conclusions

This paper presented three efficient main memory algorithms for filtering event contents with respect to conjunctions of (attribute, comparison operator, constant) predicates. These algorithms are three-step algorithms. The first step computes the predicates satisfied by the event. The second step uses this information to determine the subscription clusters that might have subscription verified by the event. Finally, the
third step checks each subscription of these clusters to obtain the subscriptions matched by the event. The \textit{propagation} algorithm indexes the subscriptions using a single-dimensional while the other two apply a multi-dimensional index. Both the \textit{static} and the \textit{dynamic} matching algorithms follow the schema-based clustering approach to index the subscriptions but differ on the clustering strategy applied to define and maintain the subscription clustering instance.

The \textit{static} algorithm computes the best clustering instance for a given set of subscriptions. However, the cost of this computation is high. Moreover, this matching solution is not able to adapt the clustering instance to changes that can occur in the subscription and event distributions. Therefore, the static matching solution cannot avoid the degradation of the clustering instance when event and subscription distributions can change over time. To cope with this drawback, the \textit{dynamic} algorithm is able to dynamically adapt the clustering instance to changes occurring in the subscription and/or event distributions. The \textit{dynamic clustering} solution may start from an optimal clustering instance defined for the initial set of subscriptions, or it may start from scratch, i.e., no subscriptions in the system.

Our matching solutions are able to handle a large number of subscriptions with a high rate of events, i.e., they exhibit a low matching cost. The experiments we have made show that usually the two schema-based clustering algorithms are the best ones in terms of event throughput, and the \textit{dynamic} algorithm has a throughput almost equal to the \textit{static} algorithm. This validates the heuristics used by the \textit{dynamic} matching algorithm to specify the subscription clusters. Our matching solutions significantly outperform the \textit{counting} matching algorithm. For a large number of subscriptions (several millions), the difference of performance between the \textit{counting} and the \textit{propagation} matching algorithm is more than two orders of magnitude, and is more than three orders of magnitude between the \textit{counting} and the \textit{dynamic} matching algorithm. This huge difference is due to the way the matching algorithms compute the matched subscriptions from the set of satisfied predicates. While the \textit{counting} matching algorithm considers all subscriptions in the system, our three matching algorithms just consider those subscriptions whose access predicates are satisfied.

Of course, there is a price to pay for having such an improvement in the performance. The schema-based clustering matching algorithms use more memory to hold their data structures and have a higher maintenance cost. Nevertheless, in terms of memory, the difference between the four algorithms is small. The schema-based clustering matching algorithms are the ones that demand more memory and the \textit{propagation} matching algorithm is the one that demands less. For 7 million of subscriptions this difference is inferior to 10\%. This difference in memory requirements is due to the fact that the schema-based clustering matching algorithms specify a larger number of clusters (and each cluster has a fixed cost in memory) and use multi-dimensional hash tables to index the access predicates.

In what concerns the maintainability of our matching algorithms, we have measured their subscription loading time. The \textit{counting} matching algorithm is the fastest to process subscriptions, since it uses simpler data structures to hold subscriptions. The worst algorithm is the \textit{static} matching algorithm, since it needs to consider all subscriptions in the system to compute the best clustering instance. For 4 million of subscriptions, this algorithm takes more than 10 times to load the subscriptions and to compute the best clustering instance than the \textit{counting} matching algorithm. This shows that the \textit{static} matching algorithm cannot be used in situations where subscription and event patterns are not stable, since the cost of recomputing the best clustering instance is very high. The difference between the \textit{counting} matching algorithm and the other two is much smaller. The \textit{propagation} and the \textit{dynamic} matching algorithms are about 50\% and 90\% slower than the \textit{counting} matching algorithm, respectively. Nevertheless, when comparing the \textit{dynamic} and \textit{counting} algorithms, the \textit{dynamic} algorithm is more than 1000 times faster to process events while it is just 2 times slower to process subscriptions.

Moreover, our experiments showed that the extra maintenance cost of the \textit{dynamic} matching algorithm needed to maintain statistics about the multi-dimensional hash tables that can be created and to move subscriptions from an old hash table to a better new hash table, is small. Moreover, this cost is more than compensated by the ability of the \textit{dynamic} matching algorithm to adapt the current subscription clustering instance to changes that can occur in the subscription and event distributions. This makes the \textit{dynamic} matching algorithm always better than the \textit{static} matching algorithm in environments where the subscription and/or event distributions change over time since the \textit{static} matching algorithm does not avoid the degradation of the clustering instance in such kind of environments.
Due to the distinct characteristics of our matching solutions, none of them is better than the other two for all cases. In environments where subscriptions are very scattered, it may not be worthwhile to index subscriptions using multiple predicates. In such a case, the three matching solutions index subscriptions using a single predicate and the best matching solution is the propagation matching algorithm since it has the lowest maintenance and space cost. When this is not the case, i.e., subscriptions are not that scattered, the best matching solution depends on the evolution rate of the subscription and event distributions. If the subscription and event distributions are stable, or evolve slowly, it is better to apply the static matching algorithm (if possible), since we need to compute the best clustering instance only once. However, if these distributions are evolving, the dynamic matching algorithm behaves better.

References


