Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems

Françoise Fabret Arno Jacobsen François Llirbat João Pereira Ken Ross
Dennis Shasha

Abstract

Publish/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/Subscribe software is to send events to the owners of subscriptions satisfied by those events. For example, a user subscription may consist of an interest in an airplane of a certain type, not to exceed a certain price. A published event may consist of an offer of an airplane with certain properties including price. A subscription closely resembles a trigger in that it is a long-lived conditional query associated with an action (usually, informing the subscriber). However, it is less general than a trigger so novel data structures and implementations may enable the creation of scalable, high performance publish-subscribe systems. This paper describes an attempt at the construction of such algorithms and its implementation. Using a combination of data structures, application-specific caching policies, and application-specific query processing our system can handle 600 events per second on 6 million subscriptions consisting of conjunctions of (attribute, comparison operator, value) predicates.

1 Motivation and Description of the Problem

Much of human information will be on the Web in ten years. The Web is particularly well-suited to changing information – Yahoo is a better source of current world events than newspapers. For this reason (and as pointed out in [3]) there is a need for systems to capture this changing information by notifying users of interesting events. For example, a bargain-hunter may search for something on the web, but decide it's too expensive. He may then want to be alerted when the item becomes cheaper. A food lover may wonder when certain cheeses are available in a convenient market. She too may want to be alerted. Such users would benefit from a publish/subscribe system in which they indicate their desires and they are alerted when items matching those desires are met. A tool that implements this functionality must be scalable and efficient. Indeed, it should manage millions of user demands for notifications (i.e. subscriptions). It should handle high rates of events (several million or more per day) and notify the interested users after only a short delay. In addition, it should provide a simple and expressive subscription interface and efficiently cope with the high volatility of web user demands (new subscriptions, new users and cancellations). 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.

We model a publish/subscribe system as a system managing a stream of incoming subscriptions and a stream of incoming data items (or events). Each subscription and each event is associated with a time interval during which it is considered valid. A publish/subscribe system stores both valid subscriptions and valid event and provides two complementary functionalities: First, when a new subscription comes in, the system evaluates the subscription against the valid events. Second, when a new event comes in, the system checks which are the subscriptions matched by the new event and sends the event to the interested users.

In this paper, we describe a publish/subscribe system that supports millions of subscriptions and a

high throughput of incoming events (hundreds of new events per second). We also consider the problem of supporting a high rate of subscription changes.

1.1 The Event Matching Problem

A subscription s in our system is a collection of predicates each of which is a triple consisting of an attribute, a value, and a relational operator ($<, <=, =, !=, >=, >$).

An event is a conjunction of pairs, where each pair consists of an attribute and a value. No two pairs have the same attribute. For example, (movie, groundhog day), (price, \$8), (theatre, odeon) is an event.

An event pair (a', v') *matches* a subscription predicate (a,v,relop) if a = a' and v' relop v. For example, (price, \$8) matches (price, \$10, <=) because they share the same attribute and \$8 <= \$10.

An event e *satisfies* a subscription s if every predicate in s is matched by some pair in e. For example, the event (movie, groundhog day), (price, \$8), (theatre, odeon) satisfies (movie, groundhog day, =), (price, \$10, $\leq$),(price, \$5, $\geq$).

The matching problem is: Given en event e and a set of subscription S find all subscriptions that are satisfied by e.

Notational Remark:

In the rest of the paper, we denote the set of equality predicates of s by $P(s)$. $A(s)$ represents the set of all the attributes occurring in the equality predicates of s. For example, for the subscription s =(movie, groundhog day, =), (price, \$10, $\leq$), (price, \$5, $\geq$) $P(s)$ = (movie, groundhog day, =) and $A(s)$ = movie.

1.2 Database solutions for subscription matching

In this section, we examine how database systems can be used to perform subscriptions matching directly. Until now, traditional database systems do not scale well to millions of subscriptions and very high throughput of incoming data, but research like this may change that state of affairs.

Database systems are designed for fast evaluation of queries against stored data sets. They also offer trigger functionality that can be used to check subscriptions when a new item comes in. First all valid data items might be stored in a single universal table of the form $D(A_1, ..., A_n)$ where $A_i, (i \in 1 \cdots n)$ are all possible attributes[1]. Subscriptions are defined as SQL triggers. For example, a subscription S $((A_1 = 3), (A_3 > 6))$ is implemented with the following SQL trigger :

```
CREATE TRIGGER T_S as
AFTER INSERT ON D
REFERENCING NEW ROW AS new
FOR EACH ROW
BEGIN
      IF  (new.A_1 = 3) AND  (new.A_3 < 6)
      THEN signal(S);
END
```

To manage millions of subscriptions the database system must support millions of triggers (one per subscription) and each single insertion of a data item may cause the execution of all millions of triggers. To make this solution scalable, database systems should implement optimization techniques for trigger executions. Projects TriggerMan [6] and NiagaraCQ [2] propose global optimization techniques for trigger executions. Our solution is close to the spirit of Triggerman in that it proposed main-memory data structures, though the exact nature of the data structures differ.

[1]Other schemas are possible but the essentials of ensuring the scalability of triggers are the same.

1.3 Contributions

This paper presents an efficient main memory matching algorithm for matching subscriptions which can handle a large number of volatile subscriptions (several millions) and support high rates of incoming data items (hundreds events per second). Our algorithm has the following nice properties:

1. It creates data structures that are tailored to the complexity of the subscription language.
2. Our algorithm is "processor cache conscious" in that it maximizes temporal and spatial locality. Moreover we use techniques that avoid cache misses by using the processor $PREFETCH$ command.
3. Our matching algorithm uses a schema based clustering strategy built on two main ideas: (1) group subscriptions based on their size and common conjunction of equality predicates, so many subscriptions can be (partly) evaluated using a single comparison (2) use multi-attribute hashing indexes so several subscription attributes can be evaluated using a single comparison.
4. We provide cost-based algorithms that given the knowledge of subscriptions and statistics on incoming data items are able to compute and incrementally adapt the optimal clustering to changes in subscription and data item patterns.

Our experiments using these algorithms show that we can support several millions of subscriptions, high rates of events (hundreds of events per second) and high rates of subscription changes.

Section 2 gives a general description of our matching algorithm. Section 3 presents our cost-based approach to compute optimal clustering. Section 4 presents an adaptive algorithm to deal with changes in subscription and event patterns. Section 5 presents related approaches and algorithms. Section 6 present performance studies. Finally, section 7 concludes.

2 Solution overview

2.1 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 [10]. 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 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.

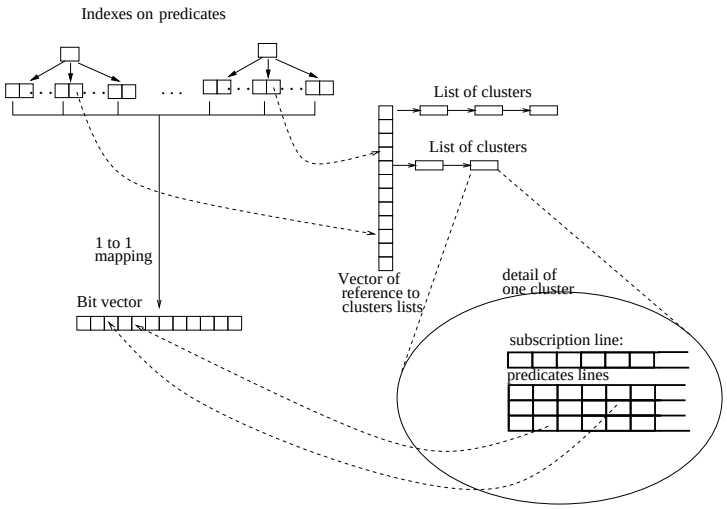


Figure 1: Algorithm Data Structures

In this paper we propose a matching algorithm which is specifically designed to be cache conscious. A lot of matching algorithms have been proposed in the literature [7, 1, 4, 12, 9]. Nevertheless, to our knowledge, none of them is aware of the cache behavior.

2.2 The matching algorithm

2.2.1 Data-structures

The algorithm data structures are depicted in Figure 1. Recall that a subscription s is defined by an identifier and a set of predicates of the form < attribute, comparison operator, value>. An event is a set of <attribute, value> pairs.

The algorithm uses a set of indexes, a predicate bit vector and a vector of references to subscriptions clusters lists, called a cluster vector. The algorithm uses the indexes to compute the set of predicates satisfied by a given incoming event, and the set of clusters which are relevant for the event. Each indexed predicate that occurs in one or more subscriptions is associated with a single entry in the predicate bit vector. This entry serves to represent the result of the predicate evaluation. It is set to 1 if the predicate is satisfied by the event and 0 otherwise.

A predicate p may also be associated with a reference to a list of subscription clusters. In such case, we say that p is an *access predicate* for all subscriptions in the clusters list. A predicate p can be an access predicate for a subscription s only if s can only match events that verifies p. This guarantees that subscriptions in the cluster list associated to p need to be checked if and only if p is satisfied. Inside the cluster list, subscriptions are grouped in *subscription clusters* according to their size (number of predicates).

Figure 1 provides a detailed description of a subscription cluster for subscriptions having 3 predicates to check. A subscriptions cluster for subscriptions of size n is organized as follows: It consists of a collection of n-dimensional arrays called a *predicates array* containing references to bit vector entries and one 1-dimensional array called a *subscription line* that contains subscription identifiers. Entry $[i, j]$ of the predicates array contains a bit vector reference to the i^{th} predicate of the subscription whose identifier is stored at position j in the subscription line. This subscription will match an event if and only if all bit vector entries referenced at column j of the predicates array are equal to 1.

input:
an event instance e
global variables:
a set of index I, a bit vector B and a set of subscriptions clusters C
local variables:
$candidate\_C$: a set of clusters
S : a set of subscriptions
Body:
B= 0; $candidate\_C$=$\emptyset$; S=$\emptyset$;
1 Predicate testing:
for each index i in I do
for each predicate p reached by e through i do
if p has a reference b to the bit vector B
then $B[b] = 1$
if p is an access predicate for a clusters list lc
then $candidate\_C = candidate\_C \cup lc$
2 Subscriptions matching :
For each cluster c in $candidate\_C$
$S = S \cup cluster\_matching(c)$
return S;

Figure 2: The event matching algorithm

2.2.2 The event matching algorithm

The algorithm is depicted in Figure 2 The algorithm is executed each time a new event comes in. First, the predicate bit vector is initialized to 0, Then the algorithm consists of two steps. The first step uses the indexes to compute the set of verified predicates. Then, it sets to 1 all corresponding entries in the predicate bit vector and collects the lists of clusters having verified access predicates. The second step considers each candidate cluster and applies the *cluster_ matching* algorithm to compute matching subscriptions.

2.2.3 The cluster matching algorithm

An example of the Cluster_matching algorithm is given below. This particular example is specialized for a group of subscriptions that all have exactly three predicates. We have a collection of similar methods specialized for small numbers of predicates, in our current implementation, ten or fewer. There is one generic method to deal with subscriptions having more predicates. A generic method is more time consuming because it needs an additional loop. However, most subscriptions have a small number of predicates, so the generic code will not be called often.

```
ansindex=0;
for(j=0;j<number_of_subscriptions;j+=UNFOLD) {
    for(k=j;k<j+UNFOLD;k++){
      if (sub_array[0][j] && sub_array[1][j] && sub_array[2][j])
      {   answer[ansindex] = k; ansindex++;}}
    _prefetch(sub_array[0][j+LOOKAHEAD]);
    _prefetch(sub_array[1][j+LOOKAHEAD]);
    _prefetch(sub_array[2][j+LOOKAHEAD]);
}
```

There are several important features of this algorithm. First, notice that the subscriptions are stored *columnwise*. Subscription j has entries in three separate subscription arrays. The reason for this choice is to improve data locality.

The loop over subscriptions is partitioned into two loops. The value `UNFOLD` is chosen so that `UNFOLD` array entries fit into a cache line.[2] At the end of the inner loop, we execute some prefetch instructions. These prefetch subroutines are 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.

Cache Performance. The columnar storage means that every entry of `sub_array[0]` will be consulted. If the condition being tested is relatively selective, we may not consult every entry of `sub_array[1]` or `sub_array[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 arrays, it may pay to avoid reading cache lines when 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. (On a Pentium III, the limit is two.) 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 doesn't pay to prefetch all of the corresponding arrays. Prefetch instructions compete with one another according to the limit above, and so it is better to avoid prefetching from arrays that are unlikely to be consulted, so that the frequently consulted arrays are prefetched more thoroughly.

2.3 Algorithm Analysis

In this section we first analyze the properties of our approach in term of memory space, cache misses, matching time and subscription changes. We also discuss the problem of designing clusters and introduce the next sections.

Space cost: Space cost is linear with the number of predicates: The size of the bit vector is equal to the number of distinct predicates. Moreover, each subscription is stored in one single cluster that also contains all bit vector references to its predicates. Thus, the total size of the subscriptions clusters is linear with the total number of predicates. Finally, an additional space is used for indexes data-structures. By using hash indexes for equality predicates and simple B-Trees for inequalities we can guarantee a space cost for indexes that is linear with the number of distinct predicates.

Cache misses: Temporal locality is reached by avoiding repetitive operations on the same data items. In our algorithm predicates are checked no more than once. In the same way, a subscription is checked no more than once. Only entries in the bit vector may be checked several times. If subscriptions have redundant predicates the bit vector is kept small and is resident in the processor cache. Spatial locality is reached by using independent data-structures for predicate matching and subscription matching. Thus we can use optimized main memory data structures for predicate testing [10]. Moreover by putting closed together in the same cluster, subscriptions that are likely to be checked for the same event we clearly improve spatial locality. Moreover by using size criteria (number of predicates) to group subscriptions in clusters we can organize clusters in integer arrays. This permit us to use asynchronous prefetch operations in the cluster_matching algorithm in order to reduce the number of synchronous caches misses which directly affect response time.

[2]For simplicity of presentation, this code assumes that the number of subscriptions 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.

Matching time: The subscriptions are grouped in cluster lists according to their access predicates. Subscriptions in the same cluster list can match only events that verify the cluster list access predicate. This clustering permits the algorithm to check only those subscriptions whose access predicate is verified. The performance challenge is to define access predicates so that each incoming event has to be matched against only a minimal number of clusters. Equality predicates contained in subscriptions are good candidates for access predicates. First, checking these predicates incurs no additional cost since they are already checked in the first step of the algorithm when computing the bit vector. Second using these predicates as access predicates permits us to guarantee an exclusive access to clusters that are accessed using equality predicates on the same attribute. This reduces significantly the number of cluster to access per event making the processing time sub-linear with the number of subscriptions. For example, consider a simple workload that consists of n subscriptions containing an equality predicate on the same attribute A. If we assume a uniform distribution of equality predicates on A values, the average number of subscription checks per event having a value for A would be n/D_A where D_A is the number of distinct equality predicates on A contained in the workload. Besides using equality predicates, a natural idea to limit the number of subscription checks is to use access predicates that are conjunctions of equality predicates. However, using multi-dimensional access predicates incurs additional space and additional processing costs since additional index structures (e.g., hash table) are needed to check them. There is a clear trade-off between the additional cost of hashing and the number of subscription checks that are saved. In section 3 we propose a cost based approach to compute an optimal clustering using simple equality predicates and a conjunction of equality predicates as access predicates.

Insertion and deletion of subscriptions: The algorithm for adding a new subscription s in the system is very similar to the event matching algorithm. It consists of two phases. First the algorithm inserts predicates of s in the predicates indexes[3]. Then, the algorithm chooses an access predicate for s and inserts s in the corresponding cluster. The cost of insertion algorithm is close to the event matching cost. Indeed by using B_trees and/or hashing tables as indexes we can keep the maintenance cost of indexes close to the retrieving cost. Moreover, inserting s in a cluster simply requires to add a new entry at the end of each predicates array and the subscription line of the cluster. Deletions can be made fast by maintaining for each subscription the identifier of the cluster that contains it. Besides the cost of insertion or deletion, adding or deleting subscriptions can make obsolete and inefficient a previously optimal clustering. In the same way, changes in event patterns may degrade performance. In section 4 we present an adaptive algorithm that maintain an optimal clustering while supporting high rates of subscription changes and incoming data items.

3 Schema Based Clustering

The schema based clustering consists of (1) grouping the subscriptions in terms of their size, and a common conjunction of equality predicates as access predicate and (2) using multi-attribute hashing to find the subscription clusters. More precisely, given a clustering instance C, clusters of C are accessed using a set of multi-attribute hash tables called a hashing configuration. Each table of the configuration is associated with a set of attributes, called its schema;, it allows one to access clusters having predicates access built over this schema. A hashing configuration H for a clustering instance C is such that for each cluster c of C there is a table in H having an entry referencing c.

Example 3.1 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 has 100 values, and that all values for each attribute are equiprobable. Suppose that there are 7 million subscriptions in S,

[3]Indexes are updated only if s contains a new predicate that is not already in the system.

and that every subscription in S has an equality condition on at least one of A, B, and 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.

Consider a clustering instance C_1 involving access predicates that are simple equality predicates on A, B, or C. Subscriptions mentioning more than one attribute with equality would be placed in the cluster of one of them. If distributed uniformly, the population accessed by each hashing table would be 2.33 million subscriptions and each cluster would contain 23,300 subscriptions. Consider C_2 involving access predicates that are simple predicates on A, B, C, and conjunctions of two equality predicates on AB and BC. Subscriptions with AC might be uniformly distributed between A and C, and subscriptions with ABC might be uniformly distributed between AB and BC. Thus, the hashing table populations would be A: 1.5 million; B: 1 million; C: 1.5 million; AB: 1.5 million; BC: 1.5 million. Sizes of the corresponding clusters would be A: 15,000; B: 10,000; C: 15,000 ; AB: 150; BC:150.

Now consider the cost of matching an event that mentions A and B but not C. In C_1 we would need to consult 1 of the A clusters and 1 of the B clusters, for a total cost of two hash table lookups and 46,600 subscription checks. In C_2, we would need to consult (on average) 1 of the A clusters, 1 of the B clusters, and 1 of the AB clusters, for a total cost of three hash table lookups and 25,150 subscription checks. Based on this analysis, we would expect the clustering instance C_2 to be preferred for this kind of event. Note that when we get to clusters having many equality predicates as access predicate (say ABC) we expect just one subscription to check for an event mentioning A, B, and C. Further partitioning this cluster (say into $ABCD$ for those subscriptions with equality conditions on A, B, C, and D) would probably not be worthwhile because it would add an extra hash table lookup while reducing the number of subscription checks by at most 1.

The per event matching cost of the algorithm can be decomposed in three main parts: the cost needed for computing the value of the predicate bit vector, the cost of computing the references of the relevant clusters, and the cost of checking the set of accessed subscriptions. As it is generally possible to build several clustering instances for a given set of subscriptions, and the two later costs are sensitive to the way the subscriptions are clustered, the problem is to choose the most efficient clustering. In this section we describe a cost-based approach to compute optimal (schema based) clusterings for our matching algorithm. The choice of the clustering is based on a cost function using statistics over the subscriptions and the events.

The section is organized as follows. We first precise the notions of access predicate, hashing configuration and clustering instances. Then we give the matching cost and space cost incurred by matching a set of subscriptions using a given clustering. Finally we pose the clustering problem in term of minimization of the matching cost under space constraint, we enumerate the search space and we propose a greedy algorithm that produces a locally optimal solution.

3.1 Multi-attribute clustering

We consider access predicates defined as a conjunction of equality predicates. An*access predicate* is defined by a pair $< id, pred >$ where id is an identifier, and $pred$ is a set of equality predicates which are pairwise different over their attributes. The set of attributes occurring in $pred$ is called the *schema* of the the access predicate.

Hashing configuration: Let AP be a set of access predicates. In order to test these predicates against incoming events we use one (or several) multi-attribute hashing structures. Each hashing structure is intended to check predicates having a certain schema. More precisely: A multi-attribute *hashing structure* over a set of access predicates is defined by a pair $< A, h >$ where A is a set of attributes called the schema of the structure, and h is a hash function which takes an event, and returns the identifier of the access predicate (if it exists) having A as schema, and which is satisfied by e. We call a *hashing configuration* for

a set of access predicates AP the set of hashing structures $\mathcal{H} = \{< A_1, h_1 >, .., < A_n, h_n >\}$ that covers all schemas of access predicates in AP. We call *schema of the configuration* $\mathcal{H}$ the set $\{A_1, .., A-n\}$ of the schemas of the tables in $\mathcal{H}$.

Clustering instance: Given S a set of subscriptions we group the subscriptions using access predicates. A *subscription cluster* is defined by a triplet $< id, p, subs >$ where id is an identifier, p is an access predicate, and $subs$ is a set of subscriptions such that each subscription contains all the predicates occurring in p. We call *clustering instance* for S 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 the following we note $C(s)$ the cluster containing subscription s, and $AP(C)$ the set of all the access predicates to the clusters of C . Given an access predicate p of $AP(C)$, we note $clusters(C, p)$ the set of clusters having p as access predicates, (note that these clusters differ from each others by the size of their subscriptions). Finally, we call *hashing configuration for* C the hashing configuration covering $AP(C)$.

3.1.1 Matching cost of a clustering instance:

Assume from now that we have a set S of subscriptions, a clustering instance C for S and $\mathcal{H}$ the associated hashing configuration. The cost of matching an event on S using C includes (1) the cost for retrieving the relevant multi-attribute indexes for the event, (2) the hashing cost for each relevant table, and (3) the cost for checking the accessed subscriptions.

Thus the per event cluster cost matching is given by:

$$\begin{aligned} matching(S, C, \mathcal{H}) &= index\_retrieving(\mathcal{H}) + \sum_{H \in \mathcal{H}} \mu(H) hashing(H) \\ &\quad + \sum_{p \in AP(C)} \nu(p) (\sum_{c \in cluster(C,p)} checking(p, c)) \end{aligned}$$

where $index\_retrieving(\mathcal{H})$ is the cost for retrieving the indexes, $\mu(H)$ is the probability that the schema of the incoming event includes the schema of H, $hashing(H)$ is the cost of running the hashing function of H, $\nu(p)$ is the probability for an event to satisfy the access predicate p, and $\sum_{c \in cluster(C,p)}$ $checking(p, c)$ is the total cost for checking the subscriptions in the clusters set having p as access predicate. $checking(p, c)$ is the checking cost for one cluster. It takes into account the fact that the group of predicates in p is already checked, so only the remaining predicates have to be checked.

In the following we assume that : (1) the cost for retrieving the relevant indexes is linear with the number of structures in the hashing configuration. (2) the hashing cost is independent from the size of the hashing structure but linear with the size of the schema of the hashing structure, (3) the cost of checking a set of subscriptions is linear with the number of subscriptions. All these assumptions are consistent with our implementation. Using these assumptions leads to the following simplified cost formula:

$$matching(S, C, \mathcal{H}) = K_r * \mid \mathcal{H} \mid + \sum_{H \in \mathcal{H}} \mu(H)(C_h + K_h * \mid H.A \mid) + \sum_{s \in S} \nu(C(s).p) * checking(C(s).p, s)$$

Where $\mid \mathcal{H} \mid$, and $\mid H.A \mid$ represent the number of indexes and the size of the schema of H respectively, K_r, C_h and K_h represent three constants, $C(s)$ is the cluster containing s and $C(s).p$ is its access predicate.

3.1.2 Space cost of a clustering instance:

The space cost of a clustering instance C on S using the hashing configuration $\mathcal{H}$ includes (1) the cost for storing hashing structures to $AP(C)$ (2) the cost for storing clusters.

Thus the space cost is given by

$$Space(S, C, \mathcal{H}) = \sum_{H \in \mathcal{H}} (init\_space(H) + \sum_{p \in AP(H.A)} hash\_space(H, p)) + \sum_{c \in cluster(C)} cluster\_space(c.p, c)$$

where $init\_space(H)$ is the initial space necessary to create an empty hash table. $hash\_space(H, p)$ is the space necessary to manage an entry for access predicate p in hashing structure H. $cluster\_space(c.p, c)$ is the size of cluster c. Regarding the data structures for clusters (see 2) this size is equal to $K_{space} * \sum_{s \in c} size(s - p.preds)$ where K_{space} represents a constant.

3.2 Computing the best clustering instance

Goal: Let S be a set of subscriptions, the problem is to find the clustering instance for S, that minimizes the cluster checking cost depicted above under the constraint that the total space occupied by the subscriptions clusters and the hashing structures is less than a given amount of (main memory) space.

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

Such complexity makes the exhaustive algorithm impracticable. We propose a greedy algorithm whose worst case complexity is $\mid S \mid \times(\mid GA(S) \mid)^2$ where $\mid S \mid$ represents the number of subscriptions, $GA(S)$ is the set of the attribute groups occurring in subscriptions of S and $\mid GA(S) \mid$ represents the cardinality of $GA(S)$; this number is bound by $2^{|\mathcal{A}|}$ where $\mathcal{A}$ denotes the set of attributes occurring in equality predicates of S. Our algorithm starts from a "natural" clustering that consists in grouping the subscriptions using simple equality predicates as access predicates. Indeed using these equality predicates as access predicates incurs no additional hashing (and space) cost since hashing structures are already defined and used for the predicate testing phase of the global matching algorithm 2. Then we improve this initial clustering by defining additional multi-attribute hash tables. The additional 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 a hashing configuration schema. We first explain this notion, then we give the benefit function and describe the algorithm. Our algorithm produces a local optimum. Experimental results in section 6 show the matching time improvements realized through this algorithm.

3.2.1 Best clustering instance for a hashing configuration schema.

Let S be a set of subscriptions, $\mathcal{A}$ a hashing configuration schema for S and $\mathcal{C}(\mathcal{A})$ the set of all the clustering instances having $\mathcal{A}$ as hashing configuration schema. We call *best clustering instance* for $\mathcal{A}$ a clustering instance that gives 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 $GP(s) \cap \mathcal{A}$ that minimizes $\nu(p)$ $checking(p, s)$. Indeed, matching cost formula 3.1.1 shows that two clustering instances associated with a same hashing configuration schema only differ over the total checking cost (see line 3 of the formula). In the following we note $best(S, \mathcal{A})$ a best clustering instance for $\mathcal{A}$, $bestcost(S, \mathcal{A})$ the cost of such best clustering instance and $Space(S, \mathcal{A})$ its space cost.

3.2.2 Benefit of a choice of an additional hashing structure

Let S be a set of subscriptions, $\mathcal{H}$ a hashing 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 $\mathcal{H}$ is denoted by $B(S, \mathcal{A}, A)$ and is defined as

$bestcost(S, \mathcal{A}) - bestcost(S, \mathcal{A} \cup \{A\})$. The space cost of adding H is denoted by $DS(S, \mathcal{A}, A)$ 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. The benefit per unit space of adding a hashing structure of schema A is 0 if $B(S, \mathcal{A}, A) \leq 0$ and $B(S, \mathcal{A}, A)/DS(S, \mathcal{A}, A)$ otherwise. Benefit per unit of space may be infinite if matching benefit is strictly positive and DS is 0 (i.e., some space is saved).

3.2.3 The Greedy algorithm

The algorithm is described bellow. It takes as input a set S of subscriptions, and $Maxsize$ a space constraint and returns a hashing configuration schema and the associated best clustering instance that fits into $Maxsize$.

given :S, a set of subscriptions, and $Maxsize$, the space constraint.
$GA = GA(S)$
$\mathcal{A}_0 = \{\{A\} \mid A$ is an attribute involved in some equality predicate in $S\}$
$\mathcal{A} = \mathcal{A}_0$
$C = best(S, \mathcal{A})$
while$(Space(S, \mathcal{A}) < Maxsize)$
  Among all schemas in $GA - \mathcal{A}$ let B be a schema which has
  the maximum positive benefit per unit space with respect to $\mathcal{A}$.
  if B does not exist then return($\mathcal{A}$,C)
  else $\mathcal{A} = \mathcal{A} \cup \{B\}$
    $C = best(S, \mathcal{A})$
  endif
end while
return ($\mathcal{A}$,C)

4 Dynamic Clustering

The goal of clustering is to minimize the number of subscription checks. In the static approach presented above, clustering decisions are taken given the global knowledge of all subscriptions in the system and the knowledge of statistics about incoming event streams. But subscription and event patterns may change over time degrading an initial optimal clustering. To cope with this problem a first solution consists in periodically recomputing from scratch a clustering instance that is adapted to the new situation. Due to the complexity of this reorganization, this solution is well suited for applications where subscriptions and event patterns are relatively stable during large time intervals. But this static approach is clearly impracticable when patterms are evolving continually.

In this section we describe a dynamic clustering algorithm that incrementally adapts clustering to changes in subscription and event patterns. Our algorithm dynamically decides (1) when to redistribute subscriptions from a given a cluster to other more profitable clusters, (2) when to delete a hash table and redistribute its subscriptions and, (3) when to create a new hash table and what table to create.

These decisions rely on three metrics called *benefit margin*, *absolute benefit* and, *reparation power*. A cluster is redistributed when its benefit margin becomes high. A hash table is removed when its absolute benefit is too small and a new table is created when its reparation power is sufficiently high. We first give definition of these metrics and show the use of these metrics to characterize the current state of a clustering instance. Then we describe the maintenance algorithm. This algorithm is parametrized by thresholds setting minimal values for absolute benefit and reparation power and maximal values for benefit margin. Finally

we discuss the maintenance cost and the impact of thresholds over the tradeoff between maintenance cost and matching cost.

Absolute benefit: Absolute benefit measures the average number of checks that are saved for a given clustering instance compared to the case where no access predicate is used. Let C be a clustering instance, c a cluster in C, and s a subscription in c. The absolute benefit of s in c is equal to $(1 - \nu(p_c))$ where p_c is the access predicate of c and $\nu(p_c)$ is the probability that an incoming event satisfies p_c. Indeed, when in cluster c, subscription s is checked with a probability $\nu(p_c)$ instead of being systematically checked if no access predicate were used for s. The absolute benefit of a cluster c is the sum of all the benefits of its subscriptions and is equal to $(1 - \nu(p_c)) * \mid c \mid$. The absolute benefit of a hash table H is the sum of the absolute benefits of its clusters and is equal to $\sum_{c \in H}(1 - \nu(p_c)) * \mid c \mid$.

Benefit margin: The benefit margin focuses on the number of checks that could be saved from a given clustering instances if all possible access predicates were used. Let C be a clustering instance c a cluster in C and s a subscription in c. The benefit margin of s in c is equal to $(\nu(p_c) - \nu(P(s)))$ 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 respectively the probability that an incoming event satisfies p_c and $P(s)$. The rationale for this is that $P(s)$ is a superset of p_c. The benefit margin of a cluster c is the sum of all the benefit margin of its subscriptions and is equal to $\sum_{s \in c}(\nu(p_c) - \nu(P(s))$.

Reparation power: Let C be a clustering instance and $\mathcal{H}$ its associated hashing configuration. The reparation power of hash table that is not in $\mathcal{H}$ and with respect to a set of clusters C' in C is the absolute benefit that could be obtained by moving subscriptions from clusters c to H. This benefit is equal to $\sum_{s \in Dmove(H,C')}(1 - \nu(p_s^H))$ where $Dmove(H, C')$ is the set of subscriptions which are in a cluster c of C' and such that $\nu(p_c) \geq \nu(p_s^H)$ where p_s^H is the new access predicate for s in H and p_c is the access predicate of c.

Algorithm Metrics: In order to use metrics that are not costly to compute we use as metrics an approximation of the parameters above. This approximation is based on the fact that selectivity of equality predicates is usually (very) low. Thus we characterize the current state of the clustering instance as follows:

- For each cluster c the approximate benefit margin of c is noted $BM(c)$ and is defined as $\nu(p) \mid c \mid$
- For each hash table H, its approximated benefit is noted $B(H)$ and is defined as $\mid H \mid$
- For any potential hash table H for a set of clusters C' its approximated reparation power is noted $RP(H, C')$ and is defined as $\mid Dmove(H, C') \mid$.

4.1 Maintenance Algorithm

Maintenance algorithm is parametrized by three threshold values: $BMmax$, $Bmin$ and $RPmin$. The maintenance algorithm will act in two situations: (1)The Cluster benefit margin of a cluster rises to $BMmax$ and (2)The benefit of an existing hashing table falls below $Bmin$. The benefit margin of a cluster c may increase for two reasons: There is an insertion of a subscription in c and, there is an increase of the selectivity $\nu(p_c)$ of the access predicate of c. The benefit of a hash table may decrease when subscriptions are deleted. In our implementation these metrics are updated at each insertion and deletion of a subscription. We also assume that an independent tool periodically provides statistics over events streams and their impact on access predicate selectivity.

The algorithm is described bellow. Actions undertaken by the algorithm to cope with situation (1) and (2) are twofold and are performed in two distinct phases. At a first phase the maintenance algorithm attempts a redistribution of subscriptions. When dealing with a cluster c with excessive benefit margin the algorithm tries to redistribute each subscription s of c into another existing table that maximizes the

absolute benefit of s. When dealing with a table H with an insufficient absolute benefit the algorithm removes H and redistributes its clusters. Redistribution of clusters is performed by *redistribute*() function. This function is recursive. Indeed, as redistribution induces insertions in other clusters it may recursively induce redistribution of other clusters. Redistribution terminates when there is no more subscription to move. Since a subscription is moved at first try toward its best table, subscriptions cannot be moved more than once. When no more subscriptions can be moved, it may happen that some clusters have still an excessive benefit margin. Function *redistribute* returns these clusters. These clusters are candidate to the second phase of the algorithm for reparation. The goal of the second phase is to find some additional hash tables able to reduce this remaining benefit margin.

This reparation phase takes as input the set of clusters to repair returned by first phase. New tables are chosen in terms of their *reparation power* w.r.t. the benefit margin to solve. The algorithm considers only candidate tables able to receive subscriptions from the input clusters. It first updates their reparation power. Then it selects and creates tables which cumulates a sufficient reparation power up to $RPmax$. It may happen that some cluster sent by first phase cannot be repaired immediately due to the fact that the tables that could repair it do not have cumulated a sufficient reparation power. Nevertheless the cluster contributes to increase their reparation power that could become sufficient after several iterations of phase one. As soon as one of these tables is created, cluster subscriptions are moved to it.

Maintenance Algorithm:
given :C the current clustering instance and $\mathcal{H}$ its associated hashing configuration
PH a set of candidate hash tables that are not in $\mathcal{H}$
PHASE 1: Redistribution
ON SITUATION_1(c) /* A cluster c has an excessive benefit margin*/
$candidate\_phase2$ = redistribute(c, C)
ON SITUATION_2(H) /* A table H has an insufficient benefit */
$\mathcal{H} = \mathcal{H}$ -H;
Foreach subscription s in H do
move s toward the cluster in $\mathcal{H} - H$ that maximizes benefits of s;
ENDForeach
Foreach cluster c that received subscriptions due to the deletion of H do
If SITUATION_1(c) then $candidate\_phase2$ += redistribute(c, C) ENDIf
ENDForeach
Foreach cluster c that was deleted due to the deletion of H do
If SITUATION_1(c) then add c to $deleted\_clusters$ ENDIf
ENDForeach
PHASE 2: Creation of new hash tables
Foreach c in $deleted\_clusters$ do
Foreach table H in $PH \cap GA(c)$ do
update reparation power of H w.r.t deletion of c
remove c from $candidate\_repair(H)$
ENDForeach
ENDForeach
Foreach c in $candidate\_phase2$ do
Foreach table H in $PH \cap GA(c)$ do
update reparation power of H w.r.t c
add c to $candidate\_repair(H)$
ENDForeach
ENDForeach

While(SITUATION_3) /* there exists a candidate table with a sufficient reparation power*/
Choose a table H that has a sufficient reparation power
$\mathcal{H}= \mathcal{H} \cup \{H\}$
move to H all subscriptions in $candidate\_repair(H)$ that have a better benefit in H
update all metrics
ENDwhile/* S3 */

Besides the cost of maintaining each hash table, the maintenance cost is proportional to the number of subscription moves. When a new subscription s arrives the insertion algorithm chooses always the hash table that gives the best absolute benefit for s. However s may move to another hash table during its lifetime if (1) deletion of other subscriptions make insufficient the benefit of its hash table or, (2) insertions or changes in event statistics increase the benefit margin of the cluster of s and triggers the creation of a hash table that is better for s. Choice of threshold values clearly impacts on the number of moves. Indeed, $Bmin$ impacts on the number of hash table deletions. $BMmax$ impacts the amount of clusters candidate for new hash tables. Finally, $RPmin$ impacts the number of hash table creations. In terms of matching cost, $BMmax$ quantifies the profitability of changing a cluster. More precisely it indicates an acceptable cluster checking cost under which no cluster reorganization is profitable. For example if a cluster c has a large size but is very rarely checked its benefit margin $\nu(p_c)* \mid c \mid$ may be small enough to decide that its average checking cost is acceptable and c will never candidate to reorganization. In the same spirit $RPmin$ quantifies the profitability to create a new hash table. $Bmin$ quantifies the profitability to maintain an existing hash table in the clustering configuration. In section 6 we study the performance of the maintenance algorithm both in terms of improvements of matching cost and maintenance cost.

5 Related work

A lot of main memory matching algorithms have been proposed in the context of content based publish/subscribe systems [1, 8, 11], and triggers [6]. At the basis of these algorithms there are two main techniques.

The former one consists in two phase algorithms which test the predicates during a first step, then compute the matching subscriptions using the results of the first step. Our proposal is a two phase algorithm. We can also cite [12, 8, 9]. Neonet[8] uses a version of counting algorithm for the second step. The counting algorithm consists in "counting" for each subscription its number of hits, i.e. its number of satisfied predicates. To achieve this, the algorithm maintains an association table giving for each predicate, the subscriptions where it occurs. Each time a predicate is satisfied, the count of the corresponding subscriptions is incremented. SIFT[12] is a SDI system allowing users to subscribe for documents by specifying a set of weighted keywords. Each keyword corresponds to a *predicate keyword* in the document. In a first step, the document is parsed for finding keywords, and then the best matching subscriptions are computed using a similar counting approach. Matching algorithm proposed by Pereira et al in [9] uses a similar approach to our algorithm. This algorithm groups subscriptions with respect to their number of predicates (as our algorithm does). But it doesn't use prefetching for optimizing the second step of the algorithm, and it only uses single predicates as grouping criterium. Our performance evaluation bellow shows the benefit of using prefetching and multi-attribute hashing tables.

The second technique consists in compiling subscription predicates in a test network ala A_TREAT[5] (that could be a tree structure). Internal nodes represent tests (i.e. predicates), the leaves of the network contain references to subscriptions. Events enter the network at the root of the network they are tested at internal nodes progressing from node to node if node test succeeds. Event having successfully satisfy all the tests along a path reaches a leaf and obtain by reference the matching subscriptions. In these algorithms, each subscription can appear in only one leaf (as proposed in Aguilera et al [1]), or may appear in several

leaves (as in Gough[4]). In the first case an incomming event only have to follow one path in the tree. While in the second case it generally have to follow several paths. Therefore the first solution is more efficient but it is very space consumming. The algorithm proposed by Aguilera et al is used in the Gryphon system. When compared with the two phase approach, these algorithms suffer of several drawbacks. First they have a bad temporal and spatial locality, second they are space consumming, third the test network data structures are complex and costly to maintain with respect to insertion and updates of subscriptions making these solutions not well suited for high rates of subscription changes.

Algorithms above are designated for conjunctions of (attribute, comparison operator, constant) predicates filtering event content. Triggerman and NiagaraCQ address respectively the problem of trigger condition and continous queries evaluation. They both optimize conditions that combine predicates on incomming event with predicates on a current database state. In both cases the algorithm works in two steps. the first step is a filtering step over the content of incomming events in order to select the database conditions which are candidate to a complete evaluation. During the second step, candidate conditions (resp. queries) are evaluated using global optimization techniques. However, the more discriminating the filtering step, the less the amount of computation of the evaluation step. In NiagaraCQ database queries are evaluated using a global multi-query plan including split operators where queries are grouped according to common predicate signatures [4]. Only the most selective signature (usually a selection predicate with equality operator) is chosen for initial filtering, other selections are performed further in the plan. TriggerMan uses a A_TREAT network to evaluate conditions. Its filtering step is more sophisticated than in NiagaraCQ since it can consist in conjunctions of equality predicates signatures. Both use index techniques to improve filtering through equality predicates. Our algorithm works on any conjunction of equality and inequality predicates over event content. It could be used to enhance the filtering phase of TriggerMan and NiagaraCQ by permitting more powerful event filtering that uses together equality and inequality predicates. Each subscription in our algorithm would be an entry point in the common query plans (network for TriggerMan) that would only consist in joins and splits operators. Even when filtering is limited to equality predicates our cost based algorithms can improve performance by choosing the best multi-key index configuration. Indeed the performance experiments in the next section show that the best index configuration is neither the one consisting in choosing simple equality predicates (as NiagraCQ does) nor the one consisting in systematically choosing the maximal conjunctions of equality predicates. We show that using cost based algorithms we can approach the best configuration.

6 Performance Evaluation

In this section we evaluate the performance of our algorithm and compare the effect of our clustering strategies. We consider three versions of our algorithm: The simple *propagation* algorithm use only single equality predicates as access predicates. To evaluate the effects of the PREFETCH command (see section 2 we compare two implementations of the propagation algorithm: *propagation* does not use prefetching while *propagation_wp* does use prefetching. The *static* algorithm and the *dynamic* algorithms use a clustering strategy that takes advantage of conjunctions of equality predicates. With the static algorithm the clustering is build statically using the cost based algorithm depicted in section 3. In the dynamic Algorithm clustering is incrementally maintained using the maintenance algorithm depicted in section 4. Both algorithms are implemented with prefetching. Finally for comparison with (part of) related work we implemented the *counting* algorithm (see 5) since it is used in many publish/subscribe systems. All algorithms are implemented in our publish/subscribe system prototype. The system is evaluated under various simulated workloads, accounting for subscriptions and events emitted to the system. Our experimental results show that our algorithms are able to handle a large number of subscriptions (several millions) and a high rate of

[4] i.e, (attribute, comparison operator) for selection predicates and (attribute1, comparison operator, attribute2) for joins.

events (up to thousand events per second). A more detailed analysis of the characteristics of the various algorithms is presented below.

6.1 Experimental Setup and Workload Generation

We ran all experiments on a single-CPU Pentium workstation with an i686 CPU at 500MHz and 1GB RAM operating under Linux. The publish and subscribe system runs as a process on this workstation waiting for subscription and events to process. Subscriptions and events can be submitted to the system at any time. We implemented a workload generator that, according to a workload specification, emits subscriptions and events to the publish/subscribe system. The workload generation task ran as a separate process on the same workstation as the publish/subscribe system. Subscriptions and events are emitted to the system in fixed-size batches. The batch size may be set in the workload specification.

In order to evaluate the matching algorithms under a high number of subscriptions and high event rate we have developed the following evaluation framework.[5]

Subscriptions and events are drawn randomly according to a workload specification that determines subscriptions, predicates, events, and attribute names. A subscription workload specifies the total number of subscriptions to generate n_S, a batch size n_{S_b}, that determines the number of subscriptions to submit to the system at once, the number of predicates per subscription n_P, the number of predicates fixed per subscription $n_{P_{fix}}$ (broken down in $n_{P_{fix=}}$, $n_{P_{fix>}}$, and $n_{P_{fix<}}$, i.e., the number of predicates with the respective operators), and a predicate workload specification.

Predicates are determined by a name, an operator, a value domain, and the domain's cardinality. The value domain may be specified per predicate or once for all predicates. It determines the value of a predicate and is specified with a lower and upper bound, l_P and u_P, respectively. Values are drawn from this domain governed by a uniform distribution. 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.

Analogously, events are determined by the number of events to generate n_E, the batch size of events to submit to the system at once n_{E_b}, the number of attribute value pairs within the event n_A, the number of attributes fixed $n_{A_{fix}}$ (same breakdown as for subscriptions), 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.

The following determinants are used to influence the matching behavior of the algorithms in a probabilistic sense, i.e., to control the number of matched subscriptions for a given workload specification and to determine parameter settings for the workload generation specifications.

The *event attribute data skew* determines the distribution of attribute values of events. It may be specified differently for each attribute in the event, (in the following referred to as event skew.)

The *subscription predicate data skew* determines the distribution of predicate values. It may be specified differently for each predicate in the subscription, (in the following referred to as subscription skew.)

The correlation between subscription and event skews determine the overlap of predicate and attribute value domains. In modifying this correlation the number of events matched for a given workload specification can be influenced. This is necessary to evaluate the algorithmic behavior of the different matching algorithms at different points in their state space. It is also required to cross-validate the matching behavior of the algorithms. The workload parameters that fix a certain number of predicates (respectively attributes) serve

[5]It would be very resource intense to evaluate the algorithms with a pre–computed workload, where we know the number of events matched per subscription configuration. We resort to a simulated workload, where subscriptions and events are drawn randomly according to a workload specification. The specification allows us to influence the matching behavior of the algorithm in a probabilistic sense (a priori determine the number of matches for a given workload specification).

to determine the number of different subscription schemas that are generated on average[6].

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

Parameter	Description	Range
Global parameters		
n_t	total number of predicate / attribute names	32
Subscription and predicate determining parameters		
n_S	total number of subscriptions	100.000 - 6.000.000
n_{S_b}	number of subscriptions to submit to the system at once	10.000
n_P	number of predicates per subscription	3 - 16
$n_{P_{fix}}$	number of predicates fixed per subscription	2 - 8
l_{P_i}, u_{P_i}	limits of value domain of predicates (per predicate i)	5 - 100
Event determining Parameters		
n_E	number of events	...
n_{E_b}	number of events to submit to the system at once	100
n_A	number of attribute value pairs per event	32
$n_{A_{fix}}$	number of attributes fixed	32
l_A, u_A	limits of value domain of attributes	5 - 100

Table 1: Parameter definitions and range values.

To evaluate and compare the performance of the different algorithms we use the following metrics: overall system throughput, memory size, and system update time. The overall system throughput measures the number of events processed per unit of time for various configurations of the system. The memory size captures the resident memory size of the publish/subscribe system process, separately for the different algorithms, at different system states. System update time measures the time it takes to submit updates (insertions and deletions) to the publish/subscribe system.

Timings are taken in mill-seconds within the workload generating process, just before events or subscriptions have been submitted to the publish/subscribe system process and right after the system responds. The system responds to event submissions with the notifications that contain the IDs of matched subscriptions. The timings therefore include the interprocess communication times and individual timings account for the processing of an entire batch of subscriptions or events submitted.

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

6.2 Experiments

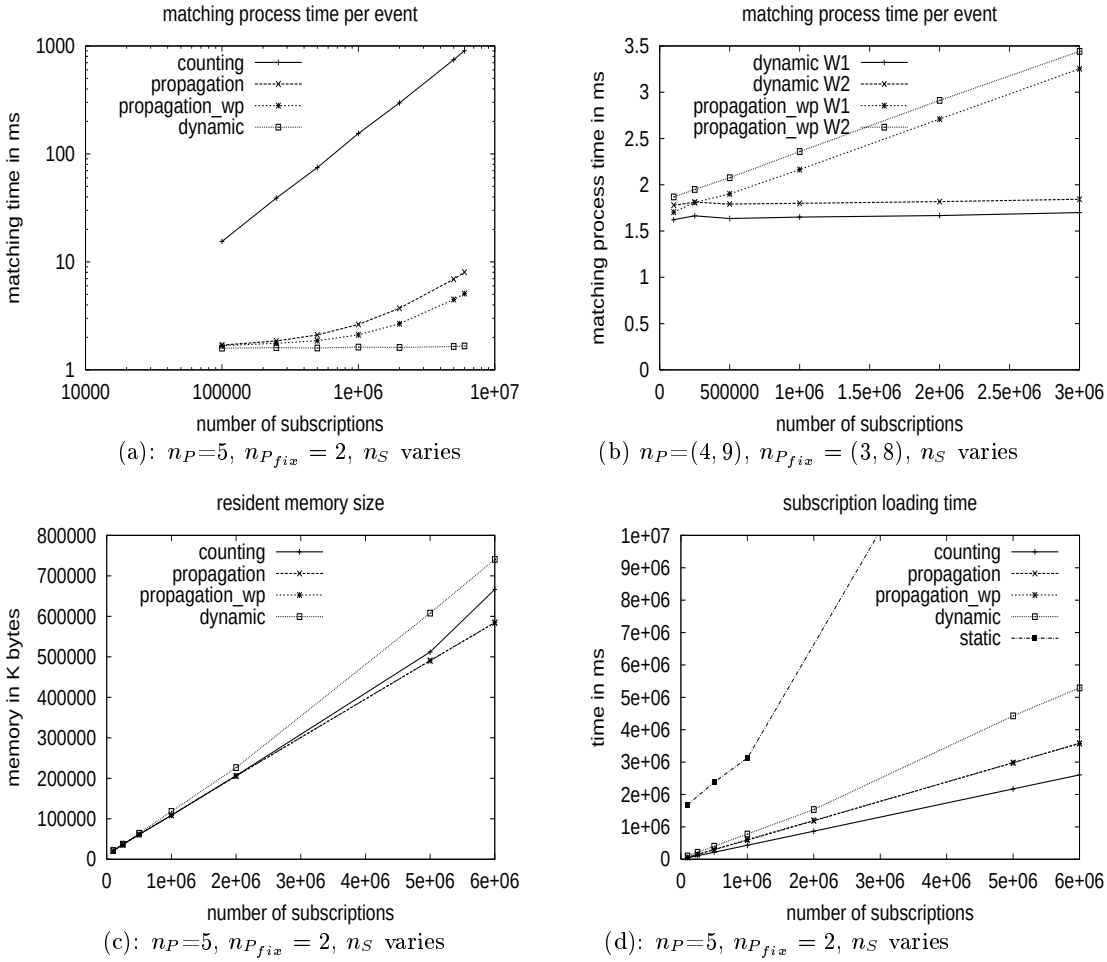
6.2.1 Total System Throughput and System Scalability

In this series of experiments we assume that the publish/subscribe system is subject to a large number of subscriptions, that these subscriptions stay in the system for a long time, and that the system must handle a high rate of events. These are the basic assumptions upon which we designed the matching algorithms. This also represents the key requirements under which, we assume, our system will have to operate.

We demonstrate the total system throughput for a given number of subscriptions across all algorithms. We also evaluate the scalability characteristics of our system, i.e., its performance in terms of event throughput with an increasing number of subscriptions to process. We further measure the memory utilization and the time it takes to process a constant number of subscriptions by the system (i.e., system update time).

Figure 3 (a) compares overall system throughput across all algorithms. The following workload specification was used: W0 = ($n_t = 32$, $n_P = 5$ (2 fixed, all equality), $n_A = 32$ (all fixed), value domain: ($l = 1$,

[6]This number may be calculated combinatorially as follows: $\frac{(n_t - n_{P_{fix}})!}{(n_t - n_P)!(n_P - n_{P_{fix}})!}$.



(a): n_P=5, $n_{P_{fix}} = 2$, n_S varies (b) n_P=(4, 9), $n_{P_{fix}} = (3, 8)$, n_S varies

(c): n_P=5, $n_{P_{fix}} = 2$, n_S varies (d): n_P=5, $n_{P_{fix}} = 2$, n_S varies

Figure 3: event matching processing time, memory resident size and subscription loading time for the several algorithms.

$u = 35$) (no skews), $n_{S_b} = 10.000$, $n_{E_b} = 100$) (the same workload is used in Figures 3(c) and 3(d)). As expected, the dynamic algorithm shows the best performance, while the counting algorithm is the least performance. The performance of the propagation algorithms lies in between these two. The prefetching technique applied in the implementation of one of the propagation algorithms improves its performance additionally by a factor of 1.5 for large numbers of subscriptions. For instance, matching 100 events against 6.000.000 subscriptions takes on the average 90377 ms (counting), 804 ms (propagation), 509 ms (propagation with prefetching) and 166 ms (dynamic). A notable feature of the dynamic algorithm is the fact that the matching time is kept independent from the number of subscriptions. This nice behavior is ensured by dynamically creating new hashing tables when the size of clusters becomes too large. We also ran experiments to compare the dynamic algorithm with the static algorithm. Static algorithm produced clustering instances that were very similar to those obtained by the dynamic (one or two additional hashing tables) and did not significantly beat the dynamic algorithm (few ms per 100 events). This shows that the metrics used in the dynamic algorithm provide a good approximation of clustering benefits.

All algorithms we implemented work in two phases. The first phase consists in to determining all predicates matched by an event, the second phase consists in to determining all matched subscription based on the information gained in the first phase. Predicate matching is done by the same function for all algorithms. Propagation and dynamic algorithms are designed to optimize the second phase. In our experiments, we separately measured each phase. The 166 ms used by the dynamic algorithm for matching

100 events are spent as follows: All satisfied predicates are discovered in 130ms, this includes the time to process events (i.e., parse arriving events etc.). All matching subscriptions were found in 10 ms. The rest of the time was spent communicating the IDs of matching subscriptions back to the client process. The propagation (with prefetching) algorithm spends the same time for predicates checking and communication but its subscription matching time increases with the number of subscriptions (from 10ms with 100.000 subscriptions to 353ms with 6.000.000). Predicate matching performance may still be improved for all algorithms, if highly optimized index structure on predicate domains are used. Our primary goal has been to highly optimize the subscription matching phase, as techniques of the former are well known.

Figure 3 (b) compares overall system throughput of the dynamic algorithm and the propagation with prefetching algorithm for different kinds of operators in predicates. The workload specifications [7] were set as follows: $W1 = (n_S = 3.000.000, n_P = 4, n_{P_{fix=}} = 2, n_{P_{fix>}} = 1$ and one none fixed predicate with equality operator, chosen freely among the $n_t = 32$ unused predicate names) and $W2 = (n_S = 3.000.000, n_P = 9, n_{P_{fix=}} = 2, n_{P_{fix<}} = 5, n_{P_{fix>}} = 1$ and one none fixed predicate with equality operator, chosen freely among the $n_t = 32$ unused predicate names). The results show that both algorithms are sensitive to non-equality predicates. Their performance decreases by a constant factor as more non-equality predicates (i.e., $W2$ vs. $W1$) are being processed. The number of satisfied non-equality predicates computed in the first phase of the algorithms is greater in $W2$ as more non-equality predicates are being generated in the workload. The performance difference of both algorithms is equal. This is due to the fact that both algorithms use the same cluster propagation algorithm to handle non-equality predicates. In this algorithm bit vector entries associated to inequality predicates of a given subscription s are checked only if all equality predicates of s are verified. Since both algorithms are tested on similar subscription workloads the probability that such situation arises is the same for both of them. Performance gain of the dynamic algorithm, as shown in the left figure is due to its improved handling of equality predicates via multi-attribute hash tables.

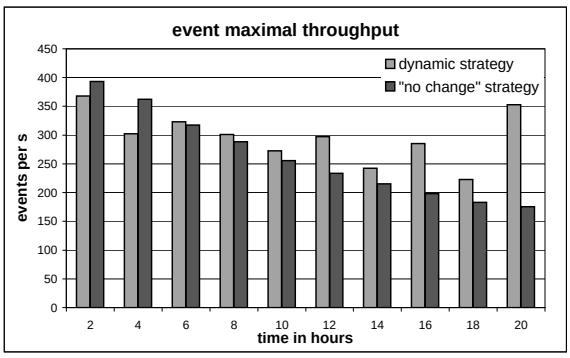
Figures 3(c) show memory utilization (first figure) and subscription loading time (second figure) across all algorithms. The individual graphs follow the natural intuition (increased processing time and memory use, due to increased data processing and storage needs). In terms of memory utilization, the propagation algorithms (both use the same internal data structures) require the least amount of memory, closely followed by the counting algorithm, while the dynamic algorithms requires the most. The multi-attribute hash-tables used in the dynamic algorithm let it use the most memory. The subscription load time (cf. Figure 3(d)) is smallest for the counting algorithm, which deploys very simple data structures, and highest for the static algorithm, that statically computes from scratch an optimal clustering configuration. Compared to Static algorithm, the dynamic algorithm improves significantly the loading time by reorganizing incrementally its internal data structures during processing to best suit the subscriptions encountered thus far. The experiments results depicted in Figure 3(a) show that the matching performance obtained with incrementally computed clusterings is as good as the ones obtained by the static algorithm.

6.2.2 Influence of Number of Predicates and Size of Value Domain

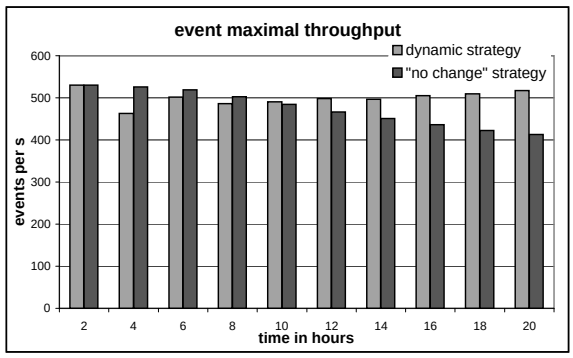
We performed a series of experiments to test the influence of the number of predicates, the size of the predicate value domains, and the kind of predicate operators used in subscriptions. Due to space limitations, we can only summarize our results here.

We ran an experiment that increased, in steps of 20, the size of the value domain of predicates, from 10 to 100. We tested the performance of the dynamic algorithm and the propagation algorithm on this workload[8]. We conclude that the more values in the domain, the better the performance of the algorithms. This is due to the fact that the higher the selectivity of each domain the fewer subscription need to be

[7] We only list values that differ from the above workload specification.
[8] All other parameters of the workload being equal to the values above.



(a): Changing subscriptions schemas



(b): adding subscription and event skew

Figure 4: Evolution of event throughput under varying conditions

verified. Furthermore, the dynamic algorithm is less sensitive to this factor than the propagation algorithm. This is explained by the fact that the dynamic algorithm can take advantage of multi-attributes access predicates.

To test the influence of the number of predicates we ran a series of experiments that increased the number of predicates in steps of twos, from 4 to 14. The conclusion here is that the more predicates per subscription the better the performance of the dynamic algorithm. On encountering subscriptions with many predicates early on, the algorithm will build hash tables with more access patterns and therefore improve performance.

6.2.3 Adaptivity to Subscription Updates

Under real world constraints, publish/subscribe systems deployed on the Internet are likely to be subjected to a constant stream of subscription updates (e.g., modifications, insertions, and deletions) and events. Subscriptions and events are likely to change in structure and content value distributions 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 at handling these conditions. In order to study its adaptive behavior in comparison to the other algorithms in such a context we simulate these conditions in this set of experiments.

In these experiments we consider situations where the publish/subscribe system has to handle concurrently incoming events and a high rate of incoming subscriptions. We assume a subscription has 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 events after aging subscriptions are deleted from the system. We say the system reaches saturation.[9] In the following experiments we investigate the behavior of our algorithms at system saturation. In the experiments the system is first populated with three million subscriptions according to a workload specification. At this state we remove 50 subscriptions (representing the 50 oldest ones, inserted 16 hrs ago) 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 next second tick to send events to the system and we measure the number of events the system can handle within the remaining time. We measure system evolution according to various application scenarios where subscription and event patterns are changing.

The first experiment depicted in Figure 4(a), investigates the impacts of subscription schema changes. This experiment models a situation where subscribers subjects of interest are changing along the time. We start from a workload $W\_1 = (n_t = 16,\ n_S = 3.000.000,\ n_P = 5,\ n_{P_{fix=}} = 1,\ n_A = 32,\ l_{p_i} = l_A = 1,\ u_{p_i} = u_A = 35)$ where all the 3.000.000 subscriptions focus on 16 of the 32 attributes available in

[9] $16 * 3600 * 50 sub/s = 2.880.000$.

the system and events provide uniform values for the 32 attributes. At saturation we use a clustering configuration that it optimal for W_1. During the first two hours subscriptions and events are following workload W_1. Then we insert subscriptions according to a new workload W_2 similar to W_1 except it focuses on the 16 attributes that are not addressed in W_1. After 18 hours the system reaches a new stable state where all subscriptions in the system are following W_2. We continue to run the experiment during two hours inserting and deleting W_2 subscriptions. Figure 4(a) shows the evolution of the average event throughput along the time (throughput is averaged every two hours) and compares two opposite strategies for clustering maintenance: The *dynamic* strategy uses the Dynamic algorithm to adapt clustering to subscriptions changes by creating (deleting) hashing tables. The *No Change* strategy does not change the initial (optimal) clustering configuration. Figure 4(a) shows that the "No Change" strategy does not prevent performance to degrade when subscriptions schema are changing. At the end the event throughput is divided by two. On the other hand the dynamic strategy adapts the clustering to the new situation. In the last two hours when subscription patterns are stable again, the system can handle 350 events per second instead of 200 events per second with "No Change" strategy. However during the transition phase, Dynamic algorithm performance is quite irregular. This is due to the additional maintenance cost that occurs when new hashing tables are created. This cost is quickly compensated by the matching benefit of the new tables. This makes dynamic strategy most of the time better than "No Change" strategy.

The second experiment is depicted in Figure 4(b). It investigates the impact of subscription skew when it is combined with event skew. This experiment models a situation where a same interest raises for both subscribers and publishers. Typical examples arise in news Dissemination systems: Few days before election of 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 phenomena we built the following experiment. We start from a workload $W\_1 = (n_t = 32,\ n_S = 3.000.000,\ n_P = 5,\ n_{P_{fix=}} = 2,\ n_A = 32,\ l_{p_i} = l_A = 1,\ u_{p_i} = u_A = 35)$ where equality predicates and attributes values are uniformly distributed among 35 values. During the first two hours, subscriptions and events are following workload W_1. Then after two hours we create both event skew and subscription skew. All new events and new subscriptions are inserted according to a new workload W_2. W_2 is similar to W_1 except there is a skew (2 different values instead of 35) on attribute values and predicates of one of the two fixed attributes used by subscriptions in W_1. After 18 hours the system reaches a new stable state where all subscriptions in the system are following W_2. We then still run the system during two hours inserting W_2 subscriptions. Figure 4(b) shows the evolution of the average event throughput along time (every two hours) when using the *dynamic* and the *No Change* strategies. Figure 4(b) shows that the "No Change" strategy does not prevent performance to degrade when more skewed subscriptions are coming into the system. At the end, the event throughput has reduced by 20%. On the other hand the dynamic strategy adapts the clustering to the new situation. At the end of the experiment when subscription patterns are stable the system can manage almost the same throughput has before[10]. At the beginning of the transition phase the cost of maintaining clustering remains slightly preponderant compared to the matching benefit. But after 8 hours the matching benefit obtained by clustering reorganization overcomes the maintenance cost.

7 Conclusion

In this paper we propose a main memory algorithm for filtering event contents with respect to conjunctions of (attribute, comparison operator, constant) predicates. Our algorithm has the following nice properties: (1) our algorithm is "processor cache conscious" in that it maximizes temporal and spatial locality. Moreover we use techniques that avoid cache misses by using processor PREFETCH command. (2)

[10] Due to subscription and event skew, more subscriptions are matched at the end of the experiment. This incurs an additional cost that cannot be compensated by clustering reorganization.

Our algorithm uses a schema based clustering strategy in order to minimizes the number of subscription checks. Subscription clusters are accessed through multi-attribute hashing tables.(3) Its clustering strategy is based on a cost model to compute the optimal hashing configuration and the corresponding clusters given statistics on incoming events. (4) We also propose a dynamic algorithm to create and remove clusters and hashing tables dynamically when the set of subscription is modified (due to insertions and deletions) or when event patterns are changing. (5) Performance studies show that our algorithm can support several millions of subscriptions and (very) high rates of events (600 hundreds event per second for 6 Millions of subscriptions on a single-CPU Pentium workstation with an i686 CPU at 500MHz and 1GB RAM). (6) Performance studies also show that our algorithm can support high rates of subscription changes.

Our filtering algorithm is implemented in a publish/subscribe system and already provides an efficient support to a subscription language consisting of DNF conditions on events. We also think that our algorithm can be used as an efficient (pre-)filtering module in more powerful Publish/subscribe systems such as SQL triggers and continuous queries.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99)*, 1999.

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

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

[4] K. J. Gough and G. Smith. Efficient recognition of events in distributed systems. In *Proceedings of ACSC-18*, 1995.

[5] E. Hanson. Rule condition testing and action execution in ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, 1992.

[6] E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the International Conference on Data Engineering*, pages 266–275, 1999.

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

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

[9] Joao Pereira, Françoise Fabret, François Llirbat, and Dennis Shasha. Efficient matching for web-based publish/subscribe systems. In *Proc. of the Int. Conf. on Cooperative Information Systems (COOPIS)*, Eilat, Israel, 2000.

[10] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 78–89, 1999.

[11] B. Segal and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, 1997.

[12] T. Yan and H. Garcia-Molina. The sift information dissemination system. In *ACM TODS 2000*, 2000.