Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems

Françoise Fabret* H. Arno Jacobsen† François Llirbat* João Pereira*
Kenneth A. Ross‡ Dennis Shasha§

ABSTRACT

Publish/Subscribe is the paradigm in which users express long-term interests ("subscriptions") and some agent "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. Each subscription consists of a conjunction of (attribute, comparison operator, value) predicates. 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 more 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 for a typical workload containing 6 million subscriptions.

1. PROBLEM DESCRIPTION

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 printed newspapers. For this reason (and as pointed out in [5]) 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 identifies 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 $(<, \leq, =, !=, \geq, >)$.

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), (theater, 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,

*INRIA Rocquencourt, email: {fabret,llirbat,pereira}@caravel.inria.fr. The work of J. Pereira was partly supported by a JNICT fellowship of Program PRAXIS XXI (Portugal).
†University of Toronto, email: jacobsen@eecg.toronto.edu.
‡Columbia University, email: kar@cs.columbia.edu. The work of K. Ross was partly supported by grant 9812014 of the United States National Science Foundation.
§Courant Institute of Mathematical Sciences/New York University, email: shasha@cs.nyu.edu. The work of D. Shasha was partly supported by grant 9988345 of the United States National Science Foundation.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

\$8) matches (price, \$10, $\leq$) 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 event (movie, groundhog day), (price, \$8), (theater, 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 Trigger approach

In this section, we examine how database systems can be used to perform subscription matching directly. Until now, traditional database systems do not scale well to millions of subscriptions and very high event throughput.

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 millions of triggers. To make this solution scalable, database systems should implement optimization techniques for trigger executions. Projects TriggerMan [8] and NiagaraCQ [4] propose global optimization techniques for trigger executions. Our solution is close to the spirit of TriggerMan in that it proposes main-memory data structures, though the exact nature of the data structures is different.

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 of 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 presents performance studies. Finally, Section 7 concludes.

2. SOLUTION OVERVIEW

2.1 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 [13]. 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 performed, 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[15]. The actual prefetch happens asynchronously, with computation allowed to continue. 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

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

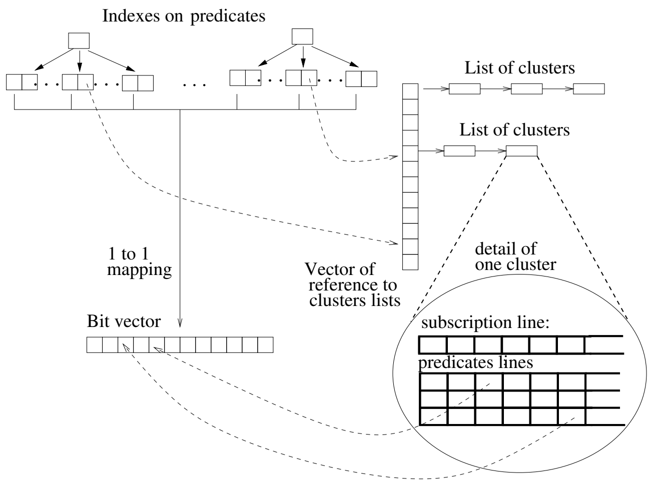


Figure 1: Algorithm Data Structures

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.

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 [9, 1, 6, 16, 12]. Nevertheless, to our knowledge, none of them is aware of the cache behavior.

2.2 The matching algorithm

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 subscription cluster 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 subscription 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

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, 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.

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 does not 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 index 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: In terms of temporal locality, only entries in the bit vector may be checked several times during the matching process. If the bit vector is small, such as when a small number of predicates appear in many subscriptions, it is resident in the processor cache. Spatial locality is achieved (1) by putting close together in the same cluster those subscriptions that are likely to be checked for the same event, (2) by using independent data-structures for predicate matching and subscription matching so that we can use optimized main memory data structures for predicate testing [13], and (3) by using size criteria (the number of predicates) to group subscriptions into clusters, we can organize clusters in integer arrays and use asynchronous prefetch operations in the cluster_matching algorithm in order to reduce the number of cache misses, which directly affects response time.

Matching time: The subscriptions are grouped in cluster lists according to their access predicates. As subscriptions in the same cluster list can match only 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 against only a minimal number of clusters. 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 the insertion algorithm is close to the event matching cost. 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 many 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 maintains a locally 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 set of subscriptions S, and a clustering instance C for S, clusters of C are accessed using a set of multi-attribute hash tables called a hashing configuration for C. Each table of the configuration is associated with a set of attributes, called its schema; it allows one to access clusters having predicates over this schema. A hashing configuration H for a clustering instance C is a set of tables such that for each cluster c of C there is a table in H having an entry referencing c. By using multi-attribute hashing, we can filter events according to their schema. Indeed, matching an incoming event e incurs a lookup per hash table of the configuration whose schema is included in the schema of e; the remaining tables are not accessed.

EXAMPLE 3.1. Consider a collection S of subscriptions and three independently distributed attributes A, B, and C

[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.

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

that are mentioned by some of the subscriptions. Suppose that each attribute has 100 values, and that all values for each attribute have the same probability. 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, 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 hash 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 hash 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: 1500; BC:1500.

Now consider the cost of matching an event that mentions A and B but not C. In C_1 we would need to consult one of the A clusters and one 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) one of the A clusters, one of the B clusters, and one of the AB clusters, for a total cost of three hash table lookups and 26,500 subscription checks. Based on this analysis, we would expect the clustering instance C_2 to be preferred for this kind of event.

The per event matching cost of the algorithm can be decomposed into 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 define the notions of access predicate, hashing configuration and clustering instance. 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 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 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 e, and returns the identifier of the access predicate (if it exists) which has A as schema, and is satisfied by e. We call a *hashing configuration* $\mathcal{H}$ for a set of access predicates AP the set of hashing structures given by $\mathcal{H} = \{< A_1, h_1 >, .., < A_n, h_n >\}$ where $\{A_1, ..., A_n\}$ denotes the set of the schemas of access predicates in AP. This set of schemas is called the *schema of the configuration*.

Clustering instance: Given a set S of subscriptions we group them using access predicates. A *subscription cluster* is defined by a triple $< id, p, subs >$ 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. 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 the following 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 $AP(C)$, we write $clusters(C, p)$ to denote the set of clusters in C having p as access predicate. (Note that these clusters differ from each others in the size of their subscriptions.) Finally, we define the *hashing configuration for C* to be the hashing configuration having an entry for each predicate of $AP(C)$.

Matching cost of a clustering instance: From now on we assume 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 against 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 cost is given by:

$$matching(S, C, \mathcal{H}) = idx\_retrieving(\mathcal{H}) + \sum_{H \in \mathcal{H}} \mu(H) hashing(H)$$

$$+ \sum_{p \in AP(C)} \nu(p) \left(\sum_{c \in cluster(C,p)} checking(p, c) \right) \quad (3.1)$$

where $idx\_retrieving(\mathcal{H})$ is the cost for retrieving the indexes, $\mu(H)$ is the probability that the incoming event schema 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, $checking(p, c)$ is the checking cost for cluster c, 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. 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.

In the following we assume that: (1) the cost for retrieving the relevant indexes is linear in the number of structures in the hashing configuration. (2) the hashing cost is independent of the size of the hashing structure but linear in the size of the schema of the hashing structure, (3) the cost of checking a set of subscriptions is linear in the number of subscriptions. All of 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.

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 the hashing structures, and (2) the cost for storing the clusters.

Thus the space cost is given by

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

where $i\_space(H)$ is the initial space necessary to create an empty hash table, $h\_space(H,p)$ is the space necessary to manage an entry for access predicate p in hashing structure H, and $c\_space(c.p,c)$ is the size of cluster c. Regarding the data structures for clusters (see Section 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 bounded 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 of 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 (Section 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.

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 shows that two clustering instances associated with a same hashing configuration schema only differ over the total checking cost (see line 2 of the formula). In the following, $best(S,\mathcal{A})$ denotes a 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.

Benefit 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 $\mathcal{A}$ 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. The benefit per unit of space may be infinite if the matching benefit is positive and DS is 0 (i.e., some space is saved).

The Greedy algorithm: The algorithm is described bellow. It takes as input a set S of subscriptions, and $Maxsize$ a space bound 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 bound.
  $GA = GA(S)$
  $\mathcal{A}_0 = \{\{A\} \mid A$ 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 wrt $\mathcal{A}$.
    if B does not exist then return($\mathcal{A}$,C)
    else $\mathcal{A} = \mathcal{A}\cup\{B\}$; $C = best(S,\mathcal{A})$
  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 of 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 patterns 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 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 two metrics called *cluster benefit margin*, and *hash table benefit*. A cluster is 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.

Cluster Benefit margin: The 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 $(\nu(p_c) - \nu(P(s)))$ 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 noted $BM(c)$ and is defined by the sum of all the benefit margins of its subscriptions and is equal to $\sum_{s \in c}(\nu(p_c) - \nu(P(s)))$. At first approximation we define $BM(c)$ as $\nu(p) \mid c \mid$ where $\mid c \mid$ is the size of the cluster.

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 $\mid H \mid - nbchecks$ where $\mid H \mid$is the number of subscriptions in the hash table and *nbchecks* is the average number of subscriptions accessed at each hash table access. At first approximation we define $B(H)$ as $\mid H \mid$.

4.1 Maintenance Algorithm

The maintenance algorithm is parameterized by three threshold values: $BMmax$, $Bdelete$ and $Bcreate$. A cluster is redistributed if its benefit margin is above $BMmax$. A hash table is created when its benefit is above $Bcreate$ and removed when its benefit is bellow $Bdelete$. The benefit margin of a cluster c may increase for two reasons: Either there is an insertion of a subscription in c, or 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 periodically after a certain number of subscription changes and/or a certain number of incoming events. The maintenance algorithm is called each time one of these metrics is updated.

The figure bellow depicts the part of the algorithm that reacts to a change of the benefit margin of a cluster c. It takes also as input the set of existing hash tables and the set of potential hash tables. It maintains for each potential table H a set of *candidate clusters* that contains subscriptions that could be moved to H. The algorithm acts only if benefit margin of c is above $BMmax$. At first step it calls the procedure **cluster_distribute** that tries to redistribute subscriptions in existing tables and updates all Benefit metrics of potential hash tables. Then the algorithm considers tables whose benefit has reached the $Bcreate$ threshold; it creates them and populates them by redistributing their candidate clusters. The function **cluster_distribute** works as follows: First it tries to redistribute each subscription s of c into another existing hash table that minimizes the probability of the subscription being checked. For example a subscription having equality predicates on attributes A, B, C and initially placed in a hash table of schema A could be moved to a hash table of schema BC if $\nu(BC, s) \leq \nu(A, s)$. If after such a redistribution the cluster margin is still excessive the function considers creating new hash tables. It considers each potential table that could receive subscriptions from c, add c to its set of $candidates$ and increments its benefits.

When the benefit of an existing hash table H drops under the $Bdelete$ threshold the maintenance algorithm simply redistributes all subscriptions in H to other existing tables (not shown below).

```
Maintenance Algorithm:
given :
c the current cluster and H_c its hash table.
𝓗 the set of hash tables that have been already created.
PH a set of potential hash table.
candidate_clusters a set of pairs (H, cls) where H belongs to PH
and cls is a set of clusters.
BODY :
 if BM(c) ≥ BMmax // Cluster c has an excessive benefit margin
   Cluster_distribute(c)
   While(Exists H in PH such that B(H) ≥ Bmin
     𝓗= 𝓗 ∪ {H}; PH= PH − {H}
     Foreach cluster c' in H.candidate do
        Cluster_distribute(c')
END BODY :
Cluster_distribute(c)
Foreach subscription s in c do
 let Hbest(s) be the hash table in 𝓗 such that ν(H, s) is minimal
 ifHbest(s) ≠ H do
   move s to H
   if s is marked do
     Foreach table H in PH ∩ GA(s) do
     B(H)-=1; delete mark from s od
 // The redistribution did not improve enough the Benefit margin
if BM(c) ≥ BMmax do
 Foreach subscription s in c such s is not marked do
   Foreach table H in PH ∩ GA(c) do
     B(H)+=1
     add c to H.candidate
     mark s od
```

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 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. The choice of threshold values clearly influences the number of moves. Indeed, $Bcreate$ influences the number of hash tables created. $BMmax$ influences the number of candidate clusters for new hash tables. 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, 10, 14, 3], and triggers [8]. (Such systems are sometimes described as "SDI systems," where SDI stands for "selective dissemination of information.") At the basis of these algorithms there are two main techniques.

The first one consists of 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 [16, 10, 12]. Neonet[10] uses a counting algorithm for the second step. The counting algorithm consists of "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 each corresponding subscription is incremented. SIFT[16] is an SDI system allowing users to subscribe to Text documents by specifying a set of weighted keywords. Matching documents are discovered based on similarity-based techniques that are very close to the counting approach. "XFilter"[2] is an SDI system that is able to handle subscriptions written in XPath for XML documents. XFilter enforces XML filtering by converting XPath queries into finite state machines (FSMs) which react to XML parsing events. Algorithm performance is improved by using indexing techniques to limit the number of FSMs to execute for a given document. Our algorithm is focusing on simpler subscriptions that consist of conjunctions of predicates. The matching algorithm proposed by Pereira et al. in [12, 11] uses a similar approach to our algorithm. Our work improves upon this approach by using prefetching, multi-attribute hash tables, and dynamic clustering.

The second technique consists of compiling subscription predicates into a test network ala A_TREAT[7]. (The network could be a tree structure.) Internal nodes represent tests (i.e., predicates), and 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. Events having successfully satisfied all the tests along a path reach a leaf and obtain by reference the matching subscriptions. In these algorithms, each subscription may appear in several leaves (as in [6]), or can appear in only one leaf (as proposed in [1]). In the first case an incoming event has to follow only one path in the tree. In the second case, it generally has to follow several paths. Therefore the first solution is more efficient but more space consuming. The algorithm proposed in [1] is used in the Gryphon system. When compared with the two phase approach, these algorithms suffer several drawbacks. They have poor temporal and spatial locality; they are space consuming; the test network data structures are complex and costly to maintain with respect to insertion and update of subscriptions, making these solutions not well suited for high rates of subscription changes.

Within the database community, "Continuous Queries" (CQ) and triggers have been developed to permit users to be notified about changes that occur in the database. They both evaluate conditions that combine predicates on incoming event with predicates on a current database state. This makes it impossible for these systems to scale over million of queries because they may have to check millions of complex conditions on the database state each time a new event modifies the database state. Scalability of XML Continuous queries is studied in [4]. Scalability of triggers is studied in [8]. In both cases the algorithm works in two steps: The first one is a filtering step over the content of incoming events in order to select the database conditions which are candidates for 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, only the most selective signature (usually a selection predicate with equality operator) is chosen for initial filtering, other selections are performed later. TriggerMan's filtering step is more sophisticated than in NiagaraCQ since it can consist of conjunctions of equality predicate 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 checks the database conditions. 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 of choosing simple equality predicates (as NiagraCQ does) nor the one consisting in systematically choosing the maximal conjunctions of equality predicates. We show that by 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 uses 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* uses. Both *static* and *dynamic* algorithms use a clustering strategy that takes advantage of conjunctions of equality predicates. With the static algorithm the clustering is built 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 Section 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.

6.1 Experimental Setup

We ran all experiments on a single-CPU Pentium workstation with an i686 CPU at 500MHz and 1GB RAM operating under Linux. The publish/subscribe system runs as a process on this workstation waiting for subscriptions and events to process. 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.

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 in a distribution, 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 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 into $n_{P_{fix=}}$, $n_{P_{fix>}}$, 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 determines 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, 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. By setting different lower and upper bounds for each predicate value domain we can simulate *subscription predicate data skew* (in the following referred to as subscription skew).

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, 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 *event attribute data skew* (in the following referred to as event skew).

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

Timings are taken in milliseconds within the workload generating process, starting just before events or subscriptions have been submitted to the publish/subscribe system process and ending 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.

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
l_A, u_A	limits of value domain of attributes	5 - 100

Table 1: Parameter definitions and range values.

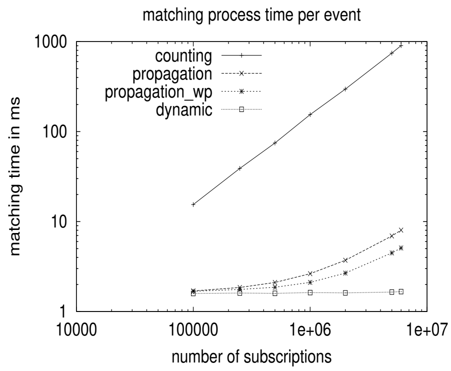
We ran several experiments multiple times and did not notice a significant difference in the results. We, therefore, do not report variances in our figures, which were lower than 0.1%, for the experimental runs repeated.

6.2 Experiments

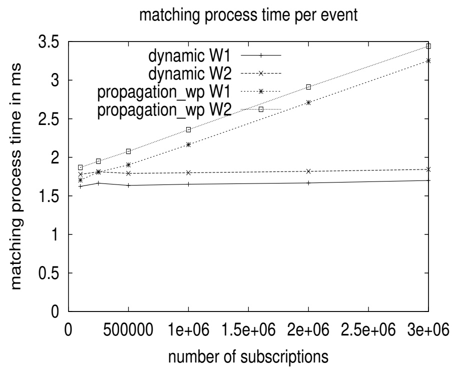
6.2.1 Total System Throughput and System Scalability

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. In these experiments we measured the event throughput, the memory size and the subscription loading time of our system using the different matching algorithms.

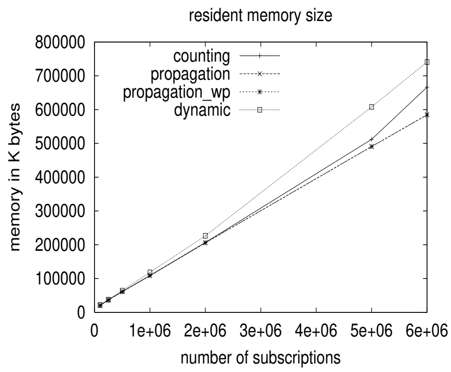
Figure 3(a) compares overall system throughput across all algorithms. The following workload specification was used: W0 = (n_t = 32, n_P = 5 (2 fixed, all equality), n_A = 32, value domain: (l = 1, u = 35) (no skews), n_{S_b} = 10,000, n_{E_b} = 100). The same workload is used in Figures 3(c) and 3(d). As expected, the dynamic algorithm shows the best performance, while the counting algorithm has the poorest 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, the event throughput of our system when loaded with 6,000,000 subscriptions is 1.1 events/s (counting), 124 events/s (propagation), 196 events/s (propagation with prefetching) and 602 events/s (dynamic). With this configuration, the time spent to compute the predicates verified by an event is 1.3 ms. This time is the same for all algorithms since they



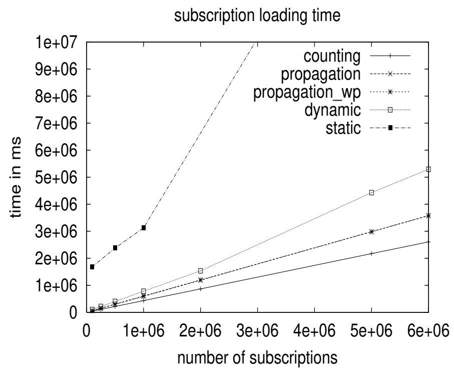
(a): n_P=5, $n_{P_{fix}} = 2$, n_S varies



(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.

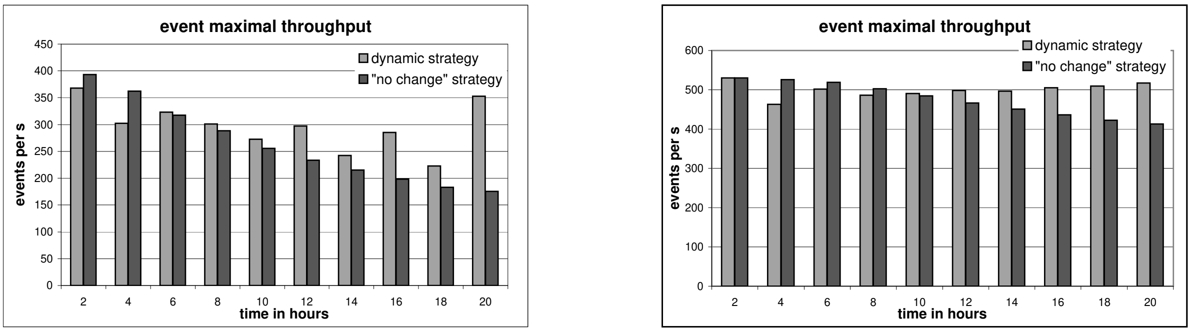
compute the satisfied predicates using the same method. Compute the matched subscriptions from the satisfied predicates takes 0.1 ms for the dynamic method and 3.53 ms for propagation with prefetching, for instance. Predicate matching performance could be further improved if highly optimized index structures on predicate domains were used. Our primary goal has been to highly optimize the subscription matching phase, as techniques for the former are well known. A notable feature of the dynamic algorithm is the fact that the matching time is kept independent of the number of subscriptions. This nice behavior is ensured by dynamically creating new hash tables when the size of clusters becomes too large. We also ran experiments to compare the dynamic and static algorithms. Static algorithm produced clustering instances that were very similar to those obtained by the dynamic algorithm (one or two additional hash tables) and did not significantly beat the dynamic algorithm. This shows that the metrics used in the dynamic algorithm provide a good approximation of clustering benefits.

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[4] were set as follows: $W1 = (n_S = 3,000,000,\ n_P = 4,\ n_{P_{fix=}} = 2,\ n_{P_{fix>}} = 1$ and one non-fixed predicate with equality operator, chosen freely among the n_t 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 non-fixed predicate with equality operator, chosen freely among the n_t 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. The performance gain of the dynamic algorithm is due to its improved handling of equality predicates via multi-attribute hash tables.

Figures 3(c) and 3(d) show memory utilization and subscription loading time, respectively, 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 use the most memory.

The subscription loading time (cf. Figure 3(d)) is smallest for the counting algorithm, which deploys very simple

[4]We only list values that differ from the above workload specification.



(a): Changing subscriptions schemas

(b): adding subscription and event skew

Figure 4: Evolution of event throughput under varying conditions

data structures, and highest for the static algorithm, that statically computes from scratch an optimal clustering configuration. Compared to the static algorithm, the dynamic algorithm improves the loading time significantly by reorganizing incrementally its internal data structures during processing to best suit the subscriptions encountered thus far. Additional experimental results (not shown) indicate that the matching performance obtained with incrementally computed clusterings is as good as that obtained by the static algorithm.

6.2.2 Adaptability 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 to handle these conditions. In order to study its adaptive behavior in comparison to the other algorithms in such a context we simulate these conditions in a 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[5] 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 our algorithms at system equilibrium. 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 impact of subscription schema changes. This experiment models a situation where subscribers subjects of interest are changing over time. We start from a workload $W3$ = ($n_t = 16$, $n_S = 3,000,000$, $n_P = 5$, $n_{P_{fix=}} = 1$, $n_A = 32$, $l_{p_i} = l_A = 1$, $u_{p_i} = u_A = 35$) where all 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 equilibrium we use a clustering configuration that it is optimal for $W3$. During the first two hours subscriptions and events are following workload $W3$. Then we insert subscriptions according to a new workload $W4$ similar to $W3$ except it focuses on the 16 attributes that are not addressed in $W3$. After 16 hours the system reaches a new stable state where all subscriptions in the system are following $W4$. We continue to run the experiment during two hours inserting and deleting $W4$ subscriptions. Figure 4(a) shows the evolution of the average event throughput over 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 subscription changes by creating (and deleting) hash tables. The *no change* strategy does not change the initial (optimal) clustering configuration. Figure 4(a) shows that the *no change* strategy is vulnerable to performance degradation when subscriptions' schemas are changing. At the end of the test period the event throughput is half of what it was. 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 the *no change* strategy. However during the transition phase, the dynamic algorithm performance is quite irregular. This is due to the additional maintenance cost that occurs when new hash tables are created. This cost is quickly compensated by the matching benefit of the new tables. This makes the dynamic strategy better than *no change* strategy most of the time.

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 an area of interest is raised for both subscribers and publishers. Typical examples arise in news dissemination systems: A few days before 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

[5] $16 * 3600 * 50 sub/s = 2,880,000$.

subject. To model this phenomenon we designed the following experiment. We start from a workload $W5 = (n_t = 32, n_S = 3,000,000, n_P = 5, n_{P_{fix=}} = 2, n_A = 32, l_{p_i} = l_A = 1, u_{p_i} = u_A = 35)$ where equality predicates and attribute values are uniformly distributed among 35 values. During the first two hours, subscriptions and events follow workload $W5$. Then after two hours we create both event skew and subscription skew. New events and subscriptions are inserted according to a new workload $W6$. $W6$ is similar to $W5$ 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 $W5$. After 16 hours the system reaches a new stable state where all subscriptions in the system are following $W6$. We then run the system for a further two hours inserting $W6$ subscriptions. Figure 4(b) shows the evolution of the average event throughput over 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 degradation when more skewed subscriptions are coming into the system. By the end of the test period, the event throughput has been 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 as before.[6] 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 the processor `prefetch` command. (2) Our algorithm uses a schema based clustering strategy in order to minimizes the number of subscription checks. Subscription clusters are accessed through multi-attribute hash tables.(3) Its clustering strategy is based on a cost model for computing 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 hash tables dynamically when the set of subscriptions is modified (due to insertions and deletions) or when event patterns are changing. (5) Performance studies show that our algorithm can support several million subscriptions, high rates of events (600 hundred event per second for a workload containing 6 million subscriptions) and 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 disjunctive normal form 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.

[6]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.

8. 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] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*, 2000.

[3] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving expressiveness and scalability in an internet-scale event notification service. In *9th ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[4] 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.

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

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

[7] E. Hanson. Rule condition testing and action execution in ariel. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, 1992.

[8] E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the Int. Conf. on Data Engineering*, 1999.

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

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

[11] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed, demonstration overview. In *Proccedings of the 26th VLDB Conference*, 2000.

[12] João 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)*, 2000.

[13] 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*, 1999.

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

[15] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2), 2000.

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