Predicate Matching and Subscription Matching in Publish/Subscribe Systems [*]

Ghazaleh Ashayer Hubert Ka Yau Leung
H.-Arno Jacobsen
Department of Electrical and Computer Engineering and
Department of Computer Science
University of Toronto
{gashayer — hubert — jacobsen}@eecg.toronto.edu

Abstract

An important class of publish/subscribe matching algorithms work in two stages. First, predicates are matched and then matching subscriptions are derived. We observe that in practice, the domain types over which predicates are defined are often of fixed enumerable cardinality. Based on this observation we propose a table-based look-up scheme for fast predicate evaluation that finds all matching predicates for each type with one table lookup. We compare this scheme to alternative general-purpose implementations. This observation may also suggests that matching in publish/subscribe systems could equally well be implemented with standard database technology. We propose two DBMS-based matching algorithms and compare the better one with a special purpose publish/subscribe matching algorithm implementation. We provide first evidence that for application scenarios that require large subscription workloads and process many events a DBMS-based solution is not a feasible alternative.

1 Introduction

In a publish/subscribe system subscriptions are matched against publications. A subscription expresses the subscriber's interest in the occurrence of specific events. A publication represents these events. If a match is established the subscriber is notified. In this interaction model publisher and subscriber are fully anonymous and decoupled, only connected through the publish/subscribe system. Subscriptions are formulated according to a subscription language and publications according to a publication type model. In many recent systems [11, 1, 9], the publication model can express typed content and the subscription language can expresses constraints on the publication content (a.k.a. content-based publish/subscribe). P/S systems have been been successfully applied to information dissemination applications [9], application integration tasks [4], and also to system management problems [13].

Our main interest in this area are applications that must manage several million subscriptions and are subjected to a very high event rate. These requirements apply, for instance, to information dissemination scenarios on the Web [3] or to location-based services in mobile wireless networks [8]. The maintenance and update of buddy-lists in centrally managed instant messaging systems, stock watch lists offered by financial information systems, network traffic filtering systems, and network management constitute good examples operating under these requirements.

A subscription "s `= ( (object = Palm)` and `(model = V)` and `(price <= 150)` and `(currency = $CAD) )`" consists of five predicates, which are conjunctively joint. This subscription may express the interest of a subscriber in a used Palm V at a resale shopping site, for example. We observe that in practice the types of the domains over which predicates are defined is often of fixed enumerable cardinality (e.g., the `'price'` will only vary between certain bounds; `'currency'` draws from a very limited set of possible values etc.). For instance, the objects for sale at a any particular site will be of limited finite number, often the categories would not be as coarse as in this example, but objects would be broken down into office products and home products and the like, even further delimiting the permissible domain type size. In this paper we exploit this observation for developing algorithms and data structures to efficiently process predicates over fixed enumerable domains. Our algorithms also support predicate inference to speed up evaluation of related predicates. If a predicate over one type is satisfied, the validity and non-validity of other predicates defined on this type can immediately be inferred.

While lots of effort has been spent on developing efficient matching algorithms [6, 2, 7, 5], it is not immediately obvious why a standard DBMS could not be used to solve the matching problem, especially with respect to the above ob-

[*]IEEE Computer Society Press in the proceedings of the 22nd International Conference on Distributed Computing Systems, 2nd-5th July 2002 in Vienna, Austria. Workshop on Distributed Event-based Systems. This research is supported by NSERC. We would also like to acknowledge Kx Systems and Arthur Whitney for providing us with the Kdb main memory database management system.

servation, in an equally efficient manner. We are not aware of any attempts to solve the matching problem with standard DBMS technology. The clear advantage of such an endeavor would be the features of a DBMS that are for free and don't have to be re-developed for a new special purpose solution, such as deletion and insertion of subscriptions, indexing of data (subscriptions), storage management, transaction support and the like. A secondary objective in this paper is to provide initial evidence that DBMSs, while fully appropriate for solving the matching problem, are not sufficiently performing in practice to stand up against high-performing special purpose publish/subscribe system implementations for applications operating under the requirements outlined above.

The contributions of this paper are as follows. We propose a new data structure for fast predicate evaluation of predicates with fixed enumerated domains and present a trade-off analysis of this data structure with alternative general-purpose implementations. We also propose two DBMS-based matching algorithms and compare their performance with several implementations of the well-known counting algorithm. We evaluate several implementations of the counting algorithm (a matrix, a bit-matrix, and a list-based one) and quantify their performance experimentally in comparison to the DBMS-based implementation. The matching algorithms have been implemented in our publish/subscribe system prototype, TOPSS (Toronto Publish/Subscribe System). TOPSS aims at investigating the application programming model of publish/subscribe systems and their functional requirements under various application domains and application requirements.

2 Type Model and Language

The publication type model and subscription language model of our system is very similar to existing proposals. Our objective was to define a very simple, sufficiently expressive, and flexible model that can be processed very efficiently. This constitutes our predicate evaluation kernel on which we layer and map extensions to the publication type model and the subscription language model. An *publication type* defines the type of permissible publications that the matching-engine can correctly process. More formally, the event types we currently support are defined by a list of attribute domain-type pairs, where each attribute is represented by an attribute name and is defined over an associated domain-type. A publication type, p_T, is defined as $p_T = (a_1 : D_1, ..., a_n : D_n)$, where a_i is the attribute name and D_i is the associated domain-type. For a given publication type, all attribute names must be unique. Our current model can represent integer, fixed-range integer intervals, string, and enumerated domain-types. A *publication* is an instance of such a type, p_T. It is not required that all attribute names, a_i, of a given type, p_T, be quantified in a publication of type p_T in order for the publication to be considered permissible according to this type model. However, for each, attribute name only values in the associated domain-type are allowed. A *subscription* is a conjunct of predicates. A predicate is defined by an attribute name, an operator, and a value. The value must be among the values of the domain-type over which the attribute is defined.

3 Algorithm and Data Structures

The class of matching algorithms we are targeting in this paper, operate in two stages. In the first stage all predicates are evaluated and in the second stage matching subscriptions are derived. In our system a predicate bit vector is used to convey the information of whether or not a predicate is satisfied. In our implementation we actually use a bit-vector (i.e., an array of n `unsigned char`s represents $8n$ predicates.) All predicates stored in the system are associated with a unique ID. The predicate bit vector is indexed by this predicate ID. IDs belonging to predicates that are deleted during the operation of the system are re-cycled. They are re-assigned before new IDs are created. This ensures that no holes of unassigned IDs are maintained in the predicate bit vector and other system data structures. The same predicate in different subscriptions is assigned the same ID, it is only stored once in the system. Similarly, subscriptions are identified with subscription IDs, managed analogously.

3.1 Predicate Matching

Predicates can be manged in several ways. Besides matching, the engine has to support predicate insertion, deletion, and update operations. (The same holds for subscription management.) A general way to manage predicates is to keep all predicates over the same attribute and same operator in an ordered linked list. Matching, insertion and deletion can then be supported by selecting the appropriate set of lists for an attribute name of a predicate (for insert and delete) or the attribute name of an attribute-value-pair (for matching) through a hash-table. Given the set of lists these operations amount to searching the corresponding list(s). All operations need to search through all selected lists incurring a cost of $O(n * nb_{op})$, where n is the number of elements in the list and nb_{op} is the number of operators for which lists are maintained. Other more sophisticated search and indexing structures (e.g., B-trees, skip lists etc.) could be used to improve insert, delete and match operations. As discussed above, however, our observation is that many predicate types used in practice are of a fixed enumerable size. For these types a more efficient data structure can be developed. This is described next.

3.2 Table-based Predicate Data Structure

Each attribute of a publication type is defined over a domain D (cf. Section 2). The type of this domain can be integer, string, or an enumerated type. We define domain, D, as $D : [l \cdots u]$ with l, u of type integer. This table-based look-up scheme has been designed to provide efficient predicate evaluation for predicates defined over finite domains, with potential cardinalities of several thousand values. It also supports predicate inference. If a predicate $a > 7$ is true then immediately all other predicates stored in the system over the same attribute name a, with $a > v$ and $v <= 7$ are also true. The predicate data structure offers an $O(1)$ predicate insertion, deletion, and look-up time for all predicates defined over the domain D. All matched and non-matched predicates can be identified in one table look-up. This does not count the traversal time to retrieve the predicate IDs of matching and non-matching predicates, which is required to set the predicate bit vector for further processing. Figure 1 exemplifies the logical organization of the predicate table. The predicate table, denoted by DS, for the domain D, extends from the lower bound l, to the upper bound u and can be indexed by the attribute–value and by the predicate operator. Implementation-wise it is a two dimensional array defined as: array$[op_1...op_n][l...u]$ of type. The predicate ID, p_{id}, of predicate $p - (a, op, v)$ defined over domain D – is stored in the table, DS.array$[op][v] = p_{id}$ (cf. Figure 1.) For subscription matching a publication, p, is broken down into its attribute value pairs. The attribute is used as hash key to retrieve the corresponding predicate table DS. The corresponding value v in the attribute value pair, (a, v), is used to index into the predicate table. Now, for each operator, different matching conclusions can be drawn. This is depicted in Figure 1.

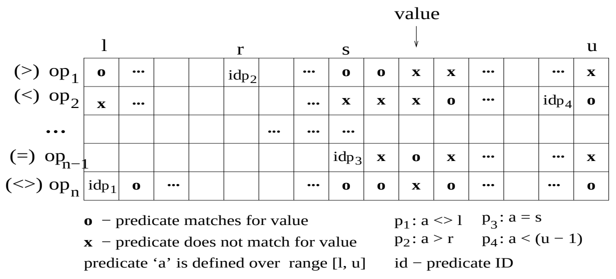

Figure 1. Domain data structure for predicates over finite ranges. (v corresponds to 'value' in the figure.)

For the greater-operator table slice, all the predicates saved in array$[>][l..v-1]$ are satisfied. So. by traversing DS.array[1] from $v - 1$ down to l, all the matched predicates of type greater-than can be retrieved, all other predicates in this slice are false. Similarly, for the less-then operator, traversing DS.array$[<]$ from $v + 1$ to u, we can find all the matched predicates defined with the less-than operator. Respectively, we see that for the equal-operator, we just need to obtain the predicate in DS.array$[=][v]$, while for the not-equal-operator all predicates, except the one in DS.array$[<>][v]$ should be recorded. These symmetries are depicted in Figure 1. The strength of this approach lies in its cache sensitive design. The table lends itself well to a memory layout that optimizes cache behavior (i.e., minimizes cache-misses). Moreover, this data structure directly supports inference to speed up predicate matching, i.e., during matching one table look-up returns all predicates satisfied on the associated attribute name in the system. This feature can be exploited to craft special purpose data structures for predicates defined over an ontology, to link different, otherwise unrelated, predicate types and thus increase predicate matching performance over these types.

3.3 Subscription Evaluation

Here, we briefly review the counting algorithm [14] and compare two alternative implementations. The great advantage of this algorithm is its simplicity. As has been shown by Fabret *et al.* [5] counting is not the most efficient algorithm, however, it does not require any additional information, such as statistics on predicates to determine their selectivity. Such statistics or more intricate schemes would be necessary for the better performing algorithms discussed by Fabret *et al.* [5] (e.g., to identify access predicates). The counting algorithm establishes through counting satisfied predicates whether the predicates satisfied in each subscription are equal to the overall number of predicates in this subscription (a match) or not equal (not a match). This scheme assumes subscriptions are conjuncts of predicates. The algorithm is based on the following data structures: a predicate bit vector, a hit vector, a subscription-predicate count vector, and a predicate subscription association table. The predicate bit vector is re-calculated in each matching phase, it records whether a predicate is true or false. In our implementation each predicate is represented by one bit in this vector. The hit vector is also re-computed for each stage of the algorithm. It records the number of satisfied predicates for each subscription in the current phase. The subscription-predicate count vector records for each subscription its true number of predicates. In our implementation we assume that a subscription may not have more than 256 predicates, represented by an `unsigned char`.[1] Hence, subscription-predicate-count-vector and hit-vector are represented as arrays of unsigned chars. The final stage of the algorithm compares hit vector and subscription-predicate vector for equality and returns the ID of the matching subscriptions. We evaluate two implementations of the association table that track which sub-

[1]This assumption may be easily relaxed.

scriptions each predicate belongs to. The first implementation stores these associations in a bit matrix, that records – bit on – whether a subscription and a predicate are associated, or not associated – bit off. In our implementation this matrix is represented as a bit-level data structure. The second implementation tracks the association between predicates and subscriptions with linked lists. Each predicate maintains a list of the subscriptions it belongs to. In our implementation we use re-allocatable arrays for the lists. The re-allocation size is set to a constant amount, a dynamic scheme would be possible that chooses the size according to the popularity of the predicate, thus minimizing expensive re-allocation steps. The bit-matrix approach has a space cost of $n \times m$, for n subscriptions and m predicates registered with the system. This can be a considerable amount for n and m large. Depending on the distribution of predicates among the subscriptions, the matrix will be sparsely populated. The advantage of this scheme is the simplicity with which it can be processed. Due to the bit-level representation, several subscriptions are processed simultaneously, through appropriate comparison operations on the byte and word level. Similarly, subscription deletion operations can be performed very efficiently, by setting all column bits of the subscription to be deleted to zero. The same holds for subscription insertion operations. Matching time cost is $n_{p_{sat}} \times m + m$, where $n_{p_{sat}}$ signifies the number of satisfied predicates. The product represents the cost of counting and the m-term accounts for the final comparison. The list-based approach has a space cost of $n * m_{avg}$, where m_{avg} is the average number of predicates per subscription. While this approach is more space efficient for large number of subscriptions, it is less simple to manage. Subscription deletion operations require an additional data structure that record subscription predicate association, so that for a given subscription all predicate lists in the data structure can be updated. Its space cost should be considered as well ($n * m_{avg}$). Insertions are simple list update operations for each of the predicate lists. The matching time cost is $n_{p_{sat}} * m_{avg}$, with n and m as above. From a complexity point of view both alternatives have the same worst case space and time costs, however, we are interested in constant factors and behavior under different workloads. The trade-offs between both alternatives depend on the number of subscriptions registered, the number of distinct predicates, and the distribution of predicates among the subscriptions.

4 Matching with DBMS

The matching problem can also be solved by using a (relational) database management system. In this sense, subscriptions are stored and manipulated (i.e. inserted, deleted and updated) as relations in the database and SQL queries perform matching operations. An SQL query is applied to the database for each publication with the query result being the set of matching subscriptions. This approach is an alternative to using specialized data structures. It is much cheaper to implement, since it is based on a general purpose DBMS, with standard operations already available. The exact schema design, query formulation, and placing of indexes on the subscription relations will have great impact on the matching performance of the implementation. Our primary focus here is to provide a simple solution to the matching problem with database management systems, which is comparable to the specialized solution provided in the previous section. We discuss two algorithms to solve the matching problem with database management systems. Optimization and refinement of these approaches are deferred to future work.

The algorithms are implemented using a commercial database called Kdb developed by Kx Systems [12]. Kdb is a main memory database. In contrast to conventional databases, a main memory database loads the entire database into random access memory. It uses main memory data structures that are optimized for fast memory accesses. For example, Kdb stores columns of database tables in a C array occupying contiguous memory location [12]. This design can make aggregation function evaluation much faster than databases that store data of a database table row by row. A main memory database is chosen for this implementation because the matching algorithms require very fast access to a very large set of data.

4.1 Sequential Search

We represent subscriptions as a relation according to the following database schema: `subs ( sid, attr, op, val )`. Each row in the table represents a predicate. `sid` is a unique identifier for a subscription; `attr` is the attribute name; `op` is the relational operator; `val` is the value of the predicate. The matching algorithm given this schema is as follows:

1. group rows by `sid`
2. for each group of predicates of the same `sid`, do:
    (a) Scan through each predicate and compare the subscription predicate value to the corresponding value of the event (if one exist) using the subscription predicate operator.
    (b) Count the number of predicate that were evaluated to true.
    (c) If number of true predicates equals the number of predicates of this subscription, then this `sid` will be included in the result set.

The implementation expresses the algorithm in KSQL, which is an SQL-like query language supported by Kdb. The query statement is generated for each publication. For an event with `n` attributes $\text{attr}_1, \text{attr}_2, ..., \text{attr}_n$ and values $\text{val}_1, \text{val}_2, ..., \text{val}_n$ The generated KSQL statement is as follows:

```
t1: select distinct sid,count$
    by sid
    from subs
    where (attr = 'attr_1'
      and ( ((op='=')  and (val =  val_1)) or
          ((op='<>') and (val <> val_1)) or
          ((op='<')  and (val >  val_1)) or
          ((op='>')  and (val <  val_1)) )
     or attr = 'attr_2'
      and ( ((op='=')  and (val =  val_2)) or
          ((op='<>') and (val <> val_2)) or
          ((op='<')  and (val >  val_2)) or
          ((op='>')  and (val <  val_2)) )
      ...
     or attr = 'attr_n'
      and ( ((op='=')  and (val =  val_n)) or
          ((op='<>') and (val <> val_n)) or
          ((op='<')  and (val >  val_n)) or
          ((op='>')  and (val <  val_n)) )

select sid
from t1
where t1[sid].x =
      (select count$ by sid from sub)[sid].x
```

This approach essentially executes a sequential search on all subscriptions and matches each of them with the publication. This is an inefficient method because the subscription table is very big. The size of the subscription table equals to the number of subscriptions multiplied by the average number of predicates. In situations where number of subscriptions is in the order of millions, this approach is not practical. Moreover, this method has redundant comparison operations if multiple subscriptions contain the same predicate. This approach is inefficient and cannot scale up to support a large number of subscriptions.

4.2 Predicate Grouping

Since multiple subscriptions can include the same predicate, computation can be reduced by evaluating each unique predicate only once and share the result between subscriptions that contain this predicate. The predicate grouping approach stores the set of unique predicates in a table and uses an association table to relate subscriptions to their corresponding predicates. This is the same scheme as applied in the specialized solutions above. The schema for this approach is defined as follows: `subs ( sid, cnt ), preds ( pid, attr, op, val ), subs_preds ( sid, pid )`. The matching process involves preparing a KSQL statement. The result of this query is the set of subscription identifiers of the matched subscription. If an incoming event have n predicates, let the attribute names be $\text{attr}_1, \text{attr}_2, ..., \text{attr}_n$, and let the corresponding attributes values be $\text{val}_1, \text{val}_2, ..., \text{val}_n$. The generated KSQL statement is as follows:

```
select sid
from subs
where (cnt = (
  select distinct sid,count$
  by sid
  from subs_preds
  where pid in (
    select pid
    from preds
    where (attr = 'attr_1'
      and ( ((op='=')  and (val =  val_1)) or
            ((op='<>') and (val <> val_1)) or
            ((op='<')  and (val >  val_1)) or
            ((op='>')  and (val <  val_1)) )
       or attr = 'attr_2'
        and ( ((op='=')  and (val =  val_2)) or
              ((op='<>') and (val <> val_2)) or
              ((op='<')  and (val >  val_2)) or
              ((op='>')  and (val <  val_2)) )
              ...
       or attr = 'attr_n'
        and ( ((op='=')  and (val =  val_n)) or
              ((op='<>') and (val <> val_n)) or
              ((op='<')  and (val >  val_n)) or
              ((op='>')  and (val <  val_n)) ) )
   )[sid].x)
```

Although the predicate grouping approach can reduce the number of comparisons, if multiple subscribers subscribe to the same predicate, the bottleneck of the algorithm is in the `subs_preds` table, which has the same size as the `subs` table in the sequential approach. For large number of subscriptions, processing this table takes a long time.

5 Experiments

In this section we experimentally evaluate the performance of different algorithms. We aim to quantify the tradeoffs of data structures and algorithms. The algorithms are implemented in our publish/subscribe engine prototype. The engine is evaluated under various simulated workloads. We have implemented a workload generator that, given a workload specification, synthesizes subscription streams and publication streams, processed by the engine.

5.1 Methodology and Workload

We ran all experiments on a dual-CPU Pentium III workstation with an i686 CPU at 900MHz and 1.5GB RAM operating under Linux (RedHat 7.2). The publish/subscribe engine runs as a process on this workstation waiting for subscriptions and publications to process. Experiments evaluating predicate-matching and subscription-matching performance have been performed separately. A subscription workload is determined by n_t the total number of distinct attribute names, n_S the total number of subscriptions processed, n_{S_b} the number of subscriptions submitted to the system at once, n_P the number of predicates per subscription, and l_{P_i}, u_{P_i} lower and upper bounds of value type domain of predicate i. A publication workload is determined by n_E the overall number of publications to process, n_{E_b} the number of publications processed at once, n_A the number of attribute-value-pairs per publication, n_A the number of attribute-value-pairs per publication, and l_A, u_A bounds of value type domain of attributes. We assume that values are drawn uniformly from the value type domains. To evaluate the performance of different algorithms we use the following metrics: overall system throughput, memory use

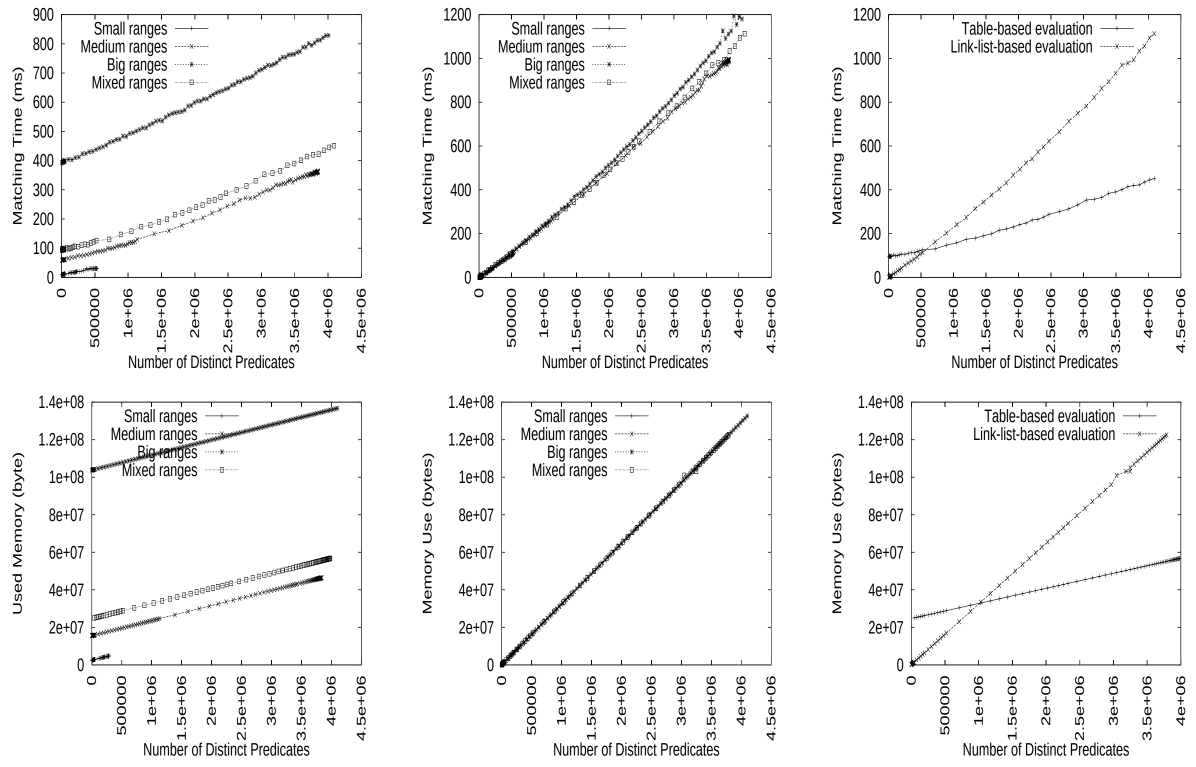

Figure 2. Matching performance (predicate schemes): (1) Table-based predicate evaluation. (2) List-based predicate evaluation. (3) Comparison of the former. Memory use: (4) Table-based predicate evaluation. (5) List-based predicate evaluation. (6) Comparison of the former.

and system update time (deletion & insertion). The overall system throughput measures the time to match publications against predicates and subscriptions stored in the system. The memory use captures the resident memory size of the publish/subscribe engine process. System update time measures the time it takes to submit updates (insertions and deletions) to the publish/subscribe system. Measurements where taken with standard Linux system calls with millisecond resolution, just before publications or subscriptions have been submitted to the publish/subscribe engine process and right after the system responds. In case of publication submissions the system responds with the notifications that contain the IDs of matched subscriptions, and in the case of subscription submission, the system responds with the IDs it assigned to the subscriptions submitted (required for latter identification). We ran the 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 repeated experimental runs.

5.1.1 Predicate Matching Performance

In this experiments the P/S-matching engine is subjected to a large number of predicates and the average time to match the predicates against a large volume of events is measured (i.e., we stop processing after the first stage in the algorithm). We demonstrate the total system throughput and measure memory use for a given number of predicates across all algorithms. We use four workloads to capture different experimental conditions. The difference between these workload specifications is the different size of the predicate domain types. $W^0_{\text{small}} = (n_t = 120, n_P = [1, ..., 20], n_A = 120$, value domain: $(l = 1, u = 250), (n_{S_b} = 120, n_{E_b} = 100))$ and $W^1_{\text{medium}} = (n_t = 120, n_P = [1, ..., 20], n_A = 120$, value domain: $(l = 250, u = 10000), (n_{S_b} = 120, n_{E_b} = 100)))$ and $W^2_{\text{big}} = (n_t = 120, n_P = [1, ..., 20], n_A = 120$, value domain: $(l = 10000, u = 100000),(n_{S_b} = 120, n_{E_b} = 100))$ and W^3_{mix} is a unified mixture of W0, W1, and W2. Figure 2(1) compares the matching time of the table-based predicate evaluation scheme under W^0, W^1, W^2, and W^3. It can be seen that the matching time, depends on the predicate domain type size, it increases with increasing domain type size. Figure 2(2) shows the behavior of linked-list-based predicate matching algorithm on W^0, W^1, W^2, and W^3. It can be observed that the matching time is nearly independent of the domain type size. Finally, Figure 2(3) compares the matching time of each algorithm under W^3 (mix ranges). It can be see that for fewer number of predicates, the linked-list-based matching exhibits better performance, but for larger number of predicates the table-based predicate evaluation scheme is more efficient. Memory utilization under the four workloads is depicted in Figures 2(4-6). For table-based predicate evaluation, memory use, not only depends on the number of predicate types (number of tables), but is also significantly influenced by the domain type size, while for the linked-list-based evaluation memory use only depends on the number of predicates stored (cf. Figures 2(4 & 5)). In Figure 2(6) we see that for fewer predicates, it is less expensive to use the linked-list-based table, however, for more predicates it is of greater advantage to resort to the table-based evaluation scheme. For W^0 the number of distinct predicates is limited, data can therefore only be provided for the first few thousand predicates.

5.1.2 Subscription Matching Performance

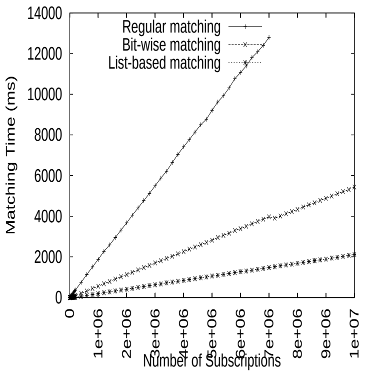

Figure 3. Subscription matching performance.

In this set of experiments the P/S-matching engine is subjected to a large number of subscriptions and the average time to match all subscriptions against new publications is measured (i.e., we measure subscription matching time only). We demonstrate the total system throughput for a given number of subscriptions across all algorithms. We also measure the memory utilization for each of the algorithms. The workload used to perform these experiments is: W^4 =(n_t = 10, n_P = 1 − 5, n_A = 10, value type domain: (l = 1, u = 10) n_{S_b} = 10, n_{E_b} = 10). We compare the bit-wise and the list-based subscription matching. In addition we substitute the bit-matrix with a byte-matrix (i.e., an unsigned char per association represented), referred to as

# subs	Kdb	bit-wise	list-based
1000	89.3	0.3	0.10
5000	206.4	1.9	0.27
10000	397.4	3.8	0.63
50000	1800.3	20.0	3.85
100000	3451.2	40.4	8.10
500000	17299.7	264.0	79.00
1000000	33473.1	562.4	190.70
5000000	338166.1	1555.7	1056.00

Table 1. Comparison of matching performance (wallclock time) for one event (ms).

the regular matching algorithm. The subscription matching time (cf. Figure 3) is smallest for the list-based counting algorithm, and highest for the regular matching algorithm. Bit-wise matching outperforms regular matching by about a factor of 3. Due to memory contention problems, the regular matching algorithm could not be run over the entire data set. In terms of memory use our analysis shows that pure list-based matching outperforms all other schemes. However, this is not the case, if we also count the data structure required to support efficient deletion for the list-based scheme (we need to track for each SID [2] its associated PIDs; in the matrix-based schemes, this comes for free.) In this case, bit-wise management outperforms the list-based scheme with respect to memory use. Figure 4 demonstrates the memory use of each algorithm. The DBMS-based implementation requires 4400KB to store 1000 subscriptions (bit-wise: about 122KB) and about 1 GB to store 5 million subscriptions (bit-wise: about 248MB).

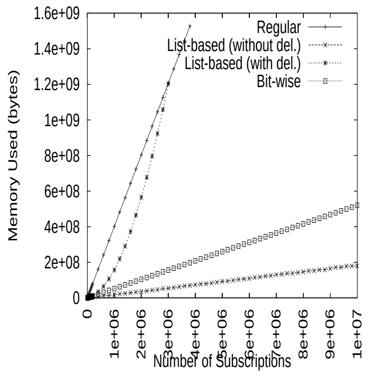

Figure 4. Memory use: different matching algorithm.

# subs	Kdb	bit-wise	list-based
1000	52	6	6
5000	84	26	28
10000	196	52	58
50000	706	261	353
100000	1699	523	862
500000	6976	2712	16093
1000000	15658	5428	97746
5000000	69811	26976	< 2678858

Table 2. Comparison of bulk loading (wall-clock) time (ms).

5.1.3 DBMS-based matching evaluation

In this set of experiments we evaluate the DBMS-based matching and compare it to the special purpose solution analyzed above. Evaluation is based on W^4. In these experiment we measure wall-clock time, since we could not adequately instrument the DBMS to reliably obtain user and system times separately. Measurements were taken in the application program (DBMS-client). The DBMS is loaded with a number of subscriptions. For each publication a query statement is generated and evaluated on the database. The measured time also accounts for the query generation time. Table 1 compares the matching time of the DBMS-based algorithms compared to the bit-wise and the list-based algorithm (both are based on the table-based predicate evaluation scheme). The Kdb solution clearly requires more time to process publications regardless of the number of subscriptions processed. Table 2 analyzes subscription bulk loading cost. Here, the DBMS-based solution outperforms the list-based matching algorithm for large number of subscriptions. Bulk loading the bit-wise data structures performs best. [3]

6 Conclusion

We have proposed a table-based lookup scheme for predicate evaluation, that applies to predicate domain types of fixed cardinality. Our predicate evaluation scheme supports inference over predicates defined on the same domain type. We have compared this scheme with a general purpose implementation. Our experiments have quantified the tradeoff in space and time between these approaches. In summary, predicate evaluation can significantly benefit from the proposed evaluation scheme, if a large number of predicates over the same domain type are to be processed. For subscription populations in the millions, table-based predicate evaluation combined with list-based subscription evaluation performed best. This combination also outperformed a DBMS-based implementation of the matching algorithm. We have thus provided initial evidence that matching algorithms based on DBMS are not as well-performing as specialized implementations. For applications that require to manage millions of subscriptions and process very high event-rates, a DBMS-based solution may therefore not be the appropriate choice. However, for other regimes, this must not be the case.

[2]subscription ID

[3]Bulk loading for Kdb was performed by reading subscriptions from a flat file, while for the special purpose solutions, bulk loading was done based on a workload generated on the fly.

References

[1] The Gryphon Project Home Page, IBM TJ Watson. http://www.research.ibm.com/gryphon/.

[2] 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.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Achieving scalability and expressiveness in an internet -scale event notification service. In *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000), Portland, OR, July 2000*, 2000.

[4] D.Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *17 th International Conferenceon Distributed Computing Systems*, Baltimore MD, U.S.A., May 1997.

[5] F. Fabret, H.-A. Jacobesen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish / subscribe systems. In *SIGMOD Conference*, 2001.

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

[7] E. N. Hanson, M. Chaabouni, C. Kim, and Y. Wang. A predicate matching algorithm for database rule systems. In *SIGMOD'90*, pages 271–280, 1990.

[8] H.-A. Jacobsen. Middleware services for selective and location-based information dissemination in mobile wireless networks. In *Workshop on Middleware for Mobile Computing, Middleware 2001*, Heidelberg, Germany, 12-16 November 2001.

[9] J. Pereira, F. Fabret, H.-A. Jacobesen, F. Llirbat, R. Preotiuc-Prieto, K. Ross, and D. Shasha. Le subscribe: Publish and subscribe on the web at extreme speed. In *SIGMOD digital library*, 2001. http://caravel.inria.fr/LeSubscribe/LeSubscribe.html.

[10] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proccedings of the 26th VLDB Conference*, 2000.

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

[12] A. Whitney. Kdb. Kx Systems. http://www.kx.com.

[13] O. Wolfson, S. Sengupta, and Y. Yemini. Managing communication networks by monitoring databases. *IEEE Transactions on Software Engineering*, 17:944–952, Sept 1991.

[14] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Trans. Database Syst.*, 19(2):332–334, 1994.