Symmetric Publish/Subscribe via Constraint Publication

Anthony Tomasic, Charles Garrod, and Kris Popendorf

June 2006
CMU-CS-06-129R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Current publish / subscribe systems offer a range of expressive subscription languages for constraints. However, classical systems restrict the publish operation to be a single published object that contains only constants and no constraints. We introduce symmetric publish / subscribe, a novel generalization of publish / subscribe where both publications and subscriptions contain constraints in addition to constants. Published objects are matched to subscriptions by computing the intersection of their constraints. This generalization improves the performance of classical publish / subscribe systems and introduces a new class of applications for publish / subscribe. This paper describes the core algorithms of our publish / subscribe implementation, evaluates the performance of these algorithms both analytically and empirically, and documents cases where the additional expressive power of symmetric publish / subscribe can be gained with minimal additional computational cost compared to the classical system.

Keywords: publish / subscribe, constraint databases, constraint publish / subscribe

1 Introduction

Current publish / subscribe systems support two key operations. The subscribe operation allows a client to register a subscription that contains a constraint. The publish operation allows a client to send a message to all clients whose constraints matches a published object. Constraint languages for the subscribe operation include atomic matching (many are listed in [11]), comparison predicates over sets of attribute/value pairs [19], XPath expressions over XML documents [1], and vector space matches on documents [20]. In all of these systems, clients may only publish objects with fixed constants.

In this paper we describe a system where both publications and subscriptions consist of constraints. The system computes the intersection of publication constraints with subscription constraints to determine the match between publications and subscriptions. This generalization leads to better expressive power than classical publish / subscribe systems where publications consists only of constants, not constraints and is analogous to constraint databases [16, 4, 3, 2] that provide query processing over constraints.

For example, in auction systems, a seller of merchandise typically offers a range of options to buyers. In particular, pricing discounts often vary depending on the size of an order (lot size). Buyers are interested in simultaneously expressing price upper bounds and lot size ranges.

In a typical classical publish / subscribe system such as the Java Messaging System (JMS), an example auction implementation might assign sellers as publishers and buyers as subscribers. The seller publishes a lot size and price on a topic (channel) that represents a product category. The publication is a set of attribute/value pairs. For example, the publication

```
topic = pencils
lotlower = 1000
lotupper = 10000
price = 1.00
```

represents an offer to sell a lot of pencils.

Symmetric publish / subscribe enables the seller to express such an offer directly as a publication constraint and matches buyers' and sellers' constraints directly. In this case the additional expressive power of symmetric / publish subscribe allows the seller to describe the offer more naturally as "$topic =$ 'pencils' AND $1000 \leq lot \leq 10000$ AND $price = 1.00$."

1.1 Contributions of this paper

This paper is the first known investigation of symmetric publish / subscribe systems. We present a preliminary investigation of such systems and report analytical and experimental results on several basic questions:

- What is the impact of the complexity of the constraint language?
- What is the performance penalty of symmetric publish / subscribe verses classical publish / subscribe?
- What is the impact on performance of the number of matches between publishers and subscribers?

- How well does the system scale with current technology?
- Does batching the processing of publications improve performance? If so, by how much?

In contrast to existing publish / subscribe systems, this system decouples matching of publications and subscriptions from the publish operation. Thus, the system supports four primary operations: client registration of subscriptions, client registration of publications, the matching of subscriptions and publications by computing the intersection of their constraints, and notification of the client pairs as indicated by the match operation. This decoupling allows us to define various transactional semantics as discussed in Section 7. The decoupling also means that batches of multiple publications may be easily simultaneously matched against the set of existing subscriptions, improving the amortized performance of the publication operation.

Section 2 describes symmetric publish / subscribe in more detail. Section 3 describes our design of a symmetric publish / subscribe system and the algorithms used to add publications and subscriptions to the system and to compute matches between constraints. Section 4 analytically evaluates the complexity of our algorithms. Section 5 describes the framework used to evaluate our symmetric publish / subscribe implementation in more detail, and section 6 presents the results of our experimental analysis. Section 7 describes a variety of transactional semantics that could be implemented using our design, as well as describes how symmetric publish / subscribe could be implemented on a larger scale in a distributed cluster environment. Section 8 surveys related work and section 9 concludes.

2 Symmetric publish / subscribe

A symmetric publish / subscribe system consists of:

- A *schema* consisting of a set of attributes $A = \{a_1 \ldots a_{|A|}\}$.
- A set of comparison operators, $O = \{o_1 \ldots o_{|O|}\}$.
- A set of types $T = \{t_1 \ldots t_{|T|}\}$ and an associated domain of values $D_{t_i} = \{d_1 \ldots d_{|D|}\}$ for each type t_i.
- An assignment of a type to each attribute.
- A set of clients K.
- A set of constraints $C = \{c_1 \ldots c_{|C|}\}$. Each constraint is a pair (k, e) of a client $k \in K$ and a boolean expression e. e is composed of the conjunction, disjunction, and negation of predicates $p \in P$, where P is set of type-safe predicates of the form $a_i \theta_i v_i$ where $a_i \in A$, $\theta_i \in O$, and $v_i \in D$. We additionally require that each attribute must appear at most once in each disjunctive phrase within a constraint.
- A set of four functions:

  publish(c**,** k**,** z**)** is a client function that publishes the constraint c of client k with version z. Version management is discussed with respect to a distributed implementation of symmetric publish / subscribe in Section 7.2.

subscribe(c, k) is a client function that adds the subscription c for client k.

match(C_1, C_2) is a server function that computes the match of constraint sets (publishers) C_1 and (subscribers) C_2

notify($K \times K$) is a server function that notifies pairs of clients $(k_1, k_2) \in K \times K$ as determined by the match function.

The schema defines an underlying data model of attribute-value pairs. Each attribute has a single type of integer, float, date, string, point, region, etc. For example, {(x, integer), (y, float), (thing, string)} is a schema (that is, an instance of the data model) with three attributes and three different types. Publisher and subscriber constraints are type-checked against the corresponding type.

Publisher and subscriber constraints are conjunctions of comparison predicates where an attribute is compared to a constant. For example, a subscriber constraint may be "$x > 5$ AND $y < 5.5$ AND *thing* = 'squirrel' ". This constraint is also a legal publisher constraint. In fact, any constraint can be either a publisher or subscriber constraint; However, the interpretation of the constraint depends on its role as a published constraint or a subscription.

In addition, we require that each constraint may reference a variable only once per disjunctive clause. Thus, redundant constraints such as "$x < 1$ AND $x \leq 3$", tautologically false constraints such as "$x < 1$ AND $x \geq 3$", and complex constraints such as "$x < 1$ AND x $\neq$ 0" are disallowed. Range expressions such as "$x > 1$ AND x $\leq$ 3" are explicitly supported with the range operators for each type. For example, for integers there are four range operators covering the four cases of open or closed intervals: $(\cdot, \cdot)$, $(\cdot, \cdot]$, $[\cdot, \cdot)$, and $[\cdot, \cdot]$.

Constraints describe *point-sets* [16], the (possibly infinite) set of points that satisfy the constraint. Thus, the constraint $0 \leq x \leq 1$ AND $0 \leq y \leq 1$ where x and y are floats describes the set of all points of a unit square anchored at the origin. No type conversion is allowed; every value must be of the type of the attribute in a given predicate.

The match operation determines the set of publisher constraints that intersect subscriber constraints. A publisher constraint intersects a subscriber constraint if the point-set of a publication constraint intersects the projections of the point-set of the subscriber constraint. The constraint c_1 defined as "$x = 1$" contains the single value (1). The constraint c_2 defined as "$x = 1$ AND $y = 1$" has the point set containing the single point (1,1). The projection of c_2 onto c_1 is (1). The projection of c_1 onto c_2 is undefined. If c_1 is a publisher constraint and c_2 is a subscriber constraint, then the constraints match because the publisher point-set of (1) intersects projection subscriber point-set of (1,1). However, the opposite does not hold. If c_2 is a publisher constraint, then subscription c_1 does not match. Another consequence of the definition of matching is the treatment of half spaces. Consider constraints $c_1 = x < 1$ and $c_2 = x < 2$. These two constraints match regardless of the association with publisher or subscriber.

3 Architecture

In this section we show that the system can be efficiently implemented as an application program executing transactions on a relational database management system (DBMS). The publish, subscribe, and match functions are implemented as application code that executes transactions on the database. After matching is complete, the process of notifying pairs of clients is straightforward and

s_id	s_disjunct_id	field	val	op
1	0	x	6	<
1	0	y	3	=
2	0	x	4	<
2	0	y	2	<

Figure 1: Example instance of a subscription table, *integer_subs*

does not require database interaction. Thus, the notify function is not discussed in the remainder of the paper.

Using a relational DBMS provides all the usual advantages of relational technology: full power of indexing and query optimization; clustering, abstract data types, main memory databases; and 24x7 operation, monitoring, performance tuning, backups, archiving, etc. In particular, our implementation of symmetric publish / subscribe benefits from (a) the clear ACID semantics for publications, subscriptions, and matching operations, which enables a variety of transaction models as discussed in Section 7; (b) the persistent storage of client state over time that allows durable subscriptions, persistent messaging, etc.; and (c) the ability of a mature query optimizer to adapt the constraint-matching algorithm to the characteristics of the constraints of each particular workload.

3.1 Constraint Publication and Subscription

Data for each publication is stored in a publication relation, which for efficiency is partitioned by data type. For example, if the system supported constraints over integer, float and string types the database would contain the relations `integer_pubs`, `float_pubs`, and `string_pubs`.

The publish operation takes a constraint $\mathcal{C}$ as input and converts $\mathcal{C}$ into an equivalent constraint $\mathcal{C}'$ that is in disjunctive normal form. Negated atomic predicates are converted to equivalent predicates in a non-negated form if possible, and otherwise an error condition is returned to the publishing client (eg. "NOT integer x < 2" would be converted to "integer x $\geq$ 2"). For each disjunct in $\mathcal{C}'$ a tuple $(p\_id, p\_disjunct\_id, count)$ is inserted into a `pub_master` table which records the number of atomic predicates in the conjunctive clause of that disjunct. For each atomic predicate in $\mathcal{C}'$ a tuple $(p\_id, p\_disjunct\_id, field, value, operator)$ is inserted to the appropriate *`type`*`_pubs` table, where p_id is the unique identifier assigned to constraint $\mathcal{C}$ and $p\_disjunct\_id$ specifies in which disjunct the atomic predicate occurs. The subscribe operation similarly takes a constraint as input and inserts such tuples into `sub_master` and *`type`*`_subs` tables. Essentially, the constraint is encoded into the relation by reifying the variables and embedding the values, and operators.

For example, consider a symmetric publish / subscribe system that implements less-than and equality over integers. The subscription constraints "$x < 6$ AND $y = 3$" and "$x < 4$ AND $y < 2$" generate the relation shown in Figure 1.

Similarly, the publication constraint "$x < 5$ AND $y = 1$" generates the relation shown in Figure 2.

3.2 Constraint Matching

A summary of the match operation is as follows. To compute the constraint intersections a query is issued for each possible pair of operators and type, grouped by pair of publish and subscribe

p_id	p_disjunct_id	field	val	op
3	0	x	5	$<$
3	0	y	1	$=$

Figure 2: Example instance of a publication table, *integer_pubs*

constraint and disjunct identifiers. The query counts the number of matching conjuncts for the given pair of operators for each field and inserts the results into an intermediate answer table. An aggregate query is then executed over the intermediate answer table, determining the total number of conjuncts that matched for each disjunct. For each subscription disjunct this count is then compared to the total number of conjuncts for each disjunct in the `pub_master` table. If these counts are equal for a publication/subscription disjunct pair then the subscription and publication match, and the result is recorded for the notify function.

To improve the efficiency of matching, a multi-attribute index of `(id, disjunct_id)` is declared for the `pub_master` and `sub_master` tables. Similarly, a multi-attribute index of `(op, val)` is declared for each *type*_pubs and *type*_subs table where possible. For types where multi-attribute indexes are not supported, single attribute indexes on `op` and `val` are declared if possible.

Operators can be freely mixed between publication and subscription constraints for a particular field/type pair. Range and comparison operations on strings operate on the lexical ordering.

3.2.1 Generating the intermediate answer table

For each possible combination of operators and types, the system issues an explicit query that counts the satisfied intersections for a particular field and subscription. The results of this sequence of queries are inserted into a temporary table. Since operators can be freely mixed, a large number of queries may be generated. A rule-based query generator generates all possible combinations.

Each query in the sequence has the same general form. For example, to generate matches between publications using integer less-than and subscriptions using integer greater-than the following query is issued:

```
INSERT INTO intermediate_answer
  SELECT p_id, p_disjunct_id,
         s_id, s_disjunct_id,
         p.field, COUNT(p.field)
    FROM integer_pubs p, integer_subs s
  WHERE p.op = ’<’ AND s.op = ’>=’
    AND p.field = s.field
    AND p.val > s.val;
  GROUP BY p_id, p_disjunct_id,
           s_id, s_disjunct_id
```

Figure 3 shows the intermediate answer table that would be generated for the example publication and subscription above after all such queries are issued. In this example, the first inserted row corresponds to the match of subscription 2's constraint “$y < 2$” with publication 3's constraint “$y = 1$.” The second row corresponds to the match of subscription 1's constraint “$x < 6$” with publication 3's constraint “$x < 5$,” while the third row corresponds to the match of subscription 2's constraint “$x < 4$”with publication 3's constraint “$x < 5$.”

Constraint information				Answer	
p_id	p_disjunct_id	s_id	s_disjunct_id	field	count
3	0	2	0	y	1
3	0	1	0	x	1
3	0	2	0	x	1

Figure 3: Example intermediate answer table

3.2.2 Counting the matches for each disjunct within the constraints

The intermediate answer table is then used to count the total number of matches for each disjunct within a publication/subscription combination. If the number of matches equals the number of conjuncts for some disjunct, then that publication/subscription combination is notified. This query is issued to compute the matching subscriptions from the intermediate answer table:

```
SELECT p_id , s_id
FROM intermediate_answer agg, pub_master pm
WHERE agg.p_id = pm.id
  AND agg.p_disjunct_id = pm.disjunct_id
GROUP BY agg.field, agg.p_id, agg.p_disjunct_id,
         agg.s_id, agg.s_disjunct_id, pm.count
HAVING sum(agg.count) = pm.count;
```

This query result is given to the notification system, which then notifies the appropriate clients.

3.3 Design Alternatives

The above section describes just one choice in a spectrum of design alternatives for the implementation of symmetric publish / subscribe, and in particular the encoding of constraints. Another option would be not to partition the subscription and publication tables by constraint data type and instead use single subscription and publication relations that could accommodate the variety of data types. Since database schema definitions are well-typed, such a relation would require a distinct attribute column for each type (i.e., integerval, floatval, etc.). The advantage of this design option is the simplification of the implementation; the disadvantage is the sparse nature of the encoding since each row of the table would contain mostly nulls.

Another option would be to avoid reification of variables and map each subscription variable into its own relation, e.g. "$x = 10$" would be encoded into either a relation `subscription_integer_x` or perhaps just `subscription_x`. Such a choice has the advantage that matching constraints for a particular variable would be highly optimized since each variable is encoded as part of the schema, but the disadvantage of greatly increasing the number of queries that must be executed to compute the match function. Additionally, such a design would require the schema to adapt any time a new variable were introduced to the workload, a frequent occurrence for many applications.

Our choice to partition the subscription and publication tables by type is a compromise between these two extremes. This design obtains dense storage of constraints without requiring the explosive number of queries required to perform matching and the frequently-evolving schema of the latter alternative.

4 Analysis

To process a set of publication and subscription constraints from scratch, the above method's response time cost is the sum of the times: (1) to store the constraints, (2) to construct the indexes, (3) to execute the match operation, and (4) to count the matches. In the following we use o for $|O|$ and use t for $|T|$ when the context is clear.

For step 1 above, the time to store a constraint and its conjuncts is $O(p + s)$ where p and s are the number of conjuncts in the publications and subscriptions. The incremental cost of an additional constraint is $O(1)$.

For step 2 above, the complexity of constructing and maintaining the indexes is $O(p \log p + s \log s)$. The incremental cost of indexing a new constraint is $O(\log p)$ or $O(\log s)$.

For step 3 above, each individual query executes in $O(p + \log p + s + \log s)$ time since the join of each query is executed as the merge of two indexes. For sufficiently large `pubs` and `subs` relations, this merge is fed into a hash-aggregate for the group-by and count. Given o operators, t types, and m conjunct matches, $O(o^2t)$ queries must be executed, the answers of these queries are saved in $O(m)$ time.

For step 4 above, the cost to count answers is $O(m + p)$ since the query is executed as a nested loop join entirely in memory. In practice, this query could be replaced with application code that computes the same result.

The total worst-case time to compute all matches is then $O(o^2t(p + \log p + s + \log s) + p \log p + s \log s + m + p)$.

This analysis reveals one major difference between classical and symmetric publish / subscribe systems. In a classical system, additional types and operators can be added essentially without any computational cost. In our symmetric publish / subscribe implementation each additional data type imposes a high cost because of the dominant o^2t term.

Publish / subscribe systems typically operate on-line with a relatively static set of subscribers and a high volume of publishers and matching operations. In this case, (a) s is indexed once off-line, (b) each match operation is performed on a small number of publishers so $p \ll s$, and (c) the number of matches is small so $m \ll s$. The time to compute on-line matches reduces to $O(o^2t \log s)$ time. In practice, the system operates in $O(o^2tc + o^2t(c - 1) \log s)$ time where c is the fraction of combinations expressed by the o^2t combinations where the match query has an empty answer. In practice c is nearly 1 since the vast majority of operation combinations are empty.

If type conversion is allowed then the dominant $O(o^2t)$ term in the above analysis instead becomes $O(o^2t^2)$ since every combination of type and operator pair must be tested.

5 Experimental Framework

This section describes the framework used to evaluate our implementation of symmetric publish / subscribe. Our prototype is based on the PostgreSQL system and supports all pre-defined PostgreSQL data types. It includes all the standard comparison predicates supported by PostgreSQL ($=$, $<$, $\leq$, $>$, $\geq$, $\neq$) as well as a subset of its geometric predicates (*overlaps*, *contains*). Publisher and subscriber constraints may be composed of the conjunction, disjunction, or negation of strictly-typed comparison predicates. For instance, the constraint "int x < 7" would match "int x = 5" but not "double x = 5.0". Our implementation is easily extensible to include all pre-defined predicates on PostgreSQL types as well as type conversion within constraints, although the latter extension would increase the complexity of the match operation as mentioned in Section 4.

We implement symmetric publish / subscribe in Java 1.4.2 using PostgreSQL 8.0.1 and the standard PostgreSQL JDBC driver. All experiments were run on a 1.0 GHz Intel Pentium III with 256 KB cache and 512 MB of memory with a 60 GB 7200 RPM IDE disk with 1 MB cache, running RedHat Linux 7.1. The database was configured with sort memory, vacuum memory, and an effective cache size increased to match the physical memory of the machine. The default statistics target is 1000 buckets. All experiments are executed with a warm database cache, an assumption we expect is reasonable since a running publish / subscribe system would frequently execute the match function to incorporate new publications.

We additionally execute a small number of experiments on a main-memory DBMS to determine the effect of such an implementation on the performance of symmetric publish / subscribe. These experiments use FirstSQL, a Java-based main-memory DBMS as the underlying system, with all experiments executed on the same hardware configuration as those using PostgreSQL. FirstSQL was configured to use all the available physical memory of the machine, to pre-load all data to memory before each experiment's execution, and to suppress all writes to disk until after the completion of the experiment so that all experiment activity occurred in main memory.

We implement the architecture described in Section 3. Our implementation of the match function contains one significant optimization: rather than execute an explicit query for each valid combination of operators and types we track which operators and types are actually used by constraints in the database and execute queries only for those combinations. While this requires one additional query to determine which combinations are used, for most workloads it saves significant time by eliminating a large number of queries for which the answer is known to be empty. Our initial experiments demonstrated that this simple optimization results in up to a factor of 10 improvement in the performance of matching. In an early implementation we also chose to combine these many queries into a large single query with many conditions ORed together in the "WHERE" clause. In doing this we learned that the PostgreSQL optimizer handles all OR conditions using a sequential scan of the base relations, generating a query plan that does not use any indexes and leading to linear performance with respect to the size of the subscription table for each workload.

Note that the design described in Section 3 works for either a single published constraint or a batch of publications. In the experiments below the same implementation is used for the batching and non-batching cases. The only difference is the number of tuples inserted into the `pubs` relation before the match operation is executed.

We examine the performance of our constraint-matching algorithm on a variety of workloads. In the first workload each subscription and each publication is composed of a single atomic predicate using integer equality. These constraints are selected uniformly such that irrespective of the number subscriptions and publications in the workload, the expected number of matches between subscriptions and publications is a small constant number (5); we call this the *fixed* workload. Specifically, the workload generator creates publication and subscription constraints of the form "integer x = R" where R is an integer chosen uniformly at random from the range $1..(\#subscriptions \cdot \#publications/5)$.

In the *proportional* workload each subscription and publication is similarly composed of a single predicate, but such that the expected number of matches is proportional to the number of subscriptions but independent of the number of publications ($0.01 \cdot \#subscriptions$). In this case each publication and subscription is of the form "integer x = R" where R is an integer chosen uniformly at random from the range $1..(100 \cdot \#publications)$.

In the *large-intermediate-results* workload each subscription and publication is composed of the conjugation of several terms in different variables, with each term in a publication expected to match the corresponding term in half the subscriptions. Note that although the probability of any term in a publication matching the corresponding term in a subscription is high, the probability of all such terms matching – and thus the probability of the publication and subscription matching – may be low. We choose the number of terms in each constraint to be such that regardless of the number of publications and subscriptions, the expected number of matches is again a small constant. Specifically, the workload generator creates constraints of the form "integer $\mathrm{x}_1 = R_1$ AND integer $\mathrm{x}_2 = R_2$ ... AND integer $\mathrm{x}_n = R_n$" where each R_i is an integer chosen uniformly at random from $\{0, 1\}$. The length of the expression, n, is chosen so that the expected number of matches ($\#subscriptions \cdot \#publications \cdot 2^{-n}$) is at least one but as close to five as possible.

Finally, the *fixed geometric* workload utilizes PostgreSQL geometric objects – a "box" – and the geometric "overlaps" operator so that a subscription box matches a publication box if the boxes overlap. These boxes are chosen such that irrespective of the number of subscriptions and publications the expected number of matches is again a small fixed constant (5). Specifically, each subscription constraint is of the form "box b overlaps (x, y, $x+\epsilon$, $y+\epsilon$)" where x and y are floating-point values chosen uniformly at random from $[0, 1]$ and ϵ is trivially small. Each publication is of the form "box b = (x, y, $x+w$, $y+w$)" where x and y are chosen uniformly at random from $[0, 1-w]$ and the width w is chosen so that the box's total area is 5 / ($\#subscriptions \cdot \#publications$). Since these publications and subscriptions are chosen to be contained within the unit square, the probability of a point-like subscription overlapping a publication box is just the area of the publication box. This yields an expected value of 5 matches regardless of the number of subscriptions and publications.

For all results in this paper, each data point is the mean performance of three executions of the experiment.

6 Results

Figure 4 lists the statistics gathered from the executions of the fixed workload using small (100,000) and large (1 million) numbers of subscribers. "Subscription generation" includes the generation of subscriber constraints from the distribution specified by the fixed workload, the parsing and processing of the constraints by the publish / subscribe system, and the output of the constraints into a flat file. "Subscription copy into db" is the time required to load that file into the subscriber table using the PostgreSQL bulk-loading mechanism. No flat file is used for the processing of the publication; in that case the "Publication generation" phase includes the generation and processing of the constraints, and the "Publication copy into db" phase consists of directly connecting to the database and updating the relations as necessary. "Index creation" and "histogram generation" are the time needed to create the indexes and generate the statistics after all subscriptions and publications have been entered. "Intermediate answer generation" and "final aggregation" are the separate stages of the matching algorithm as discussed in Section 3.2. Each of the statistics "totals" is the sum of the time of the relevant operations.

The dominant cost here is the creation of, indexing of, and histogram generation for the large subscriber relations. In an actual system these costs would be amortized across the 100000 or 1 million subscription operations. An actual system would also have to maintain indexes as subscriptions and publications are generated, rather than generate them as a single step after all subscriptions and publications have been inserted. This configuration would result in a slightly higher cost per

Experimental factors and metrics	small	large
Subscriptions	100000	1000000
Publications	1	1
Workload	fixed	fixed
Expected number of matches	5	5
Subscription generation (sec)	46.5	449.5
Subscription copy into db (sec)	4.1	74.1
Subscription total (sec)	50.6	523.6
Publication generation (sec)	0.67	0.67
Publication copy into db (sec)	0.07	0.07
Publication total (sec)	0.74	0.74
Index creation (sec)	234.7	438.8
Histogram generation (sec)	218.7	276.2
Preprocessing total (sec)	504.7	1239.3
Intermediate answer generation (sec)	0.008	0.008
Final aggregation (sec)	0.009	0.009
Matching total (sec)	0.017	0.017

Figure 4: Performance breakdown for small and large instances of the fixed workload.

operation than measured here. Given our primary choice of PostgreSQL as the underlying DBMS, we expect that the times reported are close to what could be obtained for an industrial-strength implementation of symmetric publish / subscribe.

A running system, however, also has to compute the matches between publications and subscriptions quite frequently. Depending on the transactional semantics implemented, the match function may need to be executed for each new publication and subscription. The overall performance of the system is critically dependent on the performance of the match function, and thus we focus on its performance below.

Figure 5 shows the cost of executing the match function for each workload as the number of subscriptions is scaled from 100000 to 1 million; these graphs present the same data with different vertical scales to emphasize differences in the workloads. The cost for both the fixed and fixed geometric workloads is negligible and scales well with the number of subscribers. These workloads result in small sets of intermediate results that are generated efficiently using the indexes, and the latency is dominated by the fixed costs of matching rather than the processing of the constraint data. The cost for both the proportional and large-intermediate-results workloads is much higher and scales no better than linearly with the number of subscriptions rather than sub-linearly. For the proportional workload one could not expect to do better since the output size itself is linearly dependent on the number of subscribers. For the large-intermediate-results workload the intermediate results are themselves linearly dependent on the number of subscribers even though the number of final matches is constant, so again the best performance our algorithm could be expected to obtain is a linear dependence.

Notably, the match function exhibits complex performance behavior even over such a simple selection of workloads and experimental factors. This seemingly non-linear behavior is not attributable to experimental uncertainty. It is the result of resource limits on the computer conducting the experiments as well as variations in the query execution plan selected by PostgreSQL

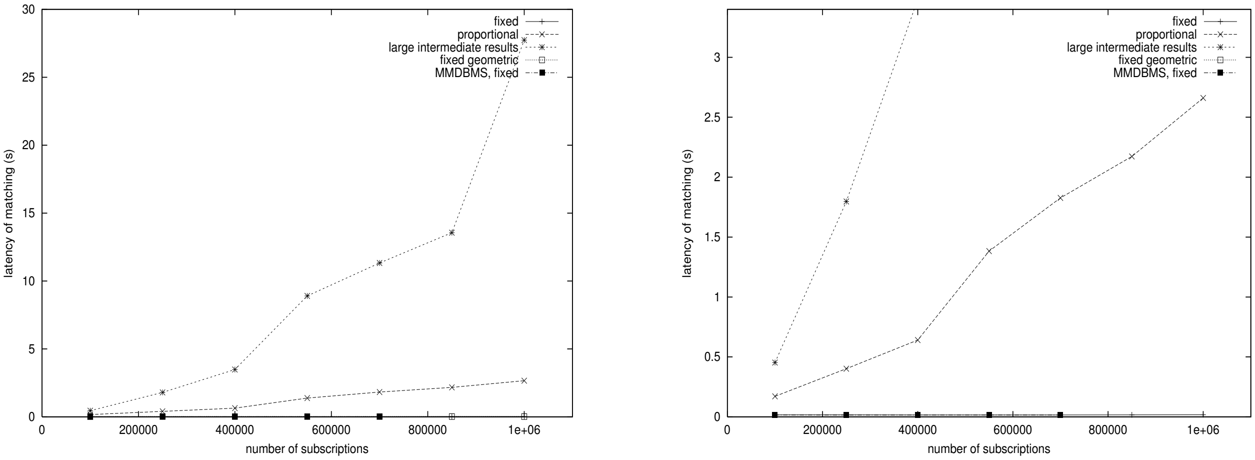

Figure 5: Impact of number of subscriptions on response time of the *match* function.

for the various experimental factors. For example, the cost of matching for the large-intermediate-results in Figure 5 scales linearly at first. However, as the workload size grows the size of the intermediate results eventually exceeds the memory available in the bufferpool and the DBMS must rely on paging data to disk for its computations. This limitation of resources and change of algorithm causes a dramatic spike the measured latency as the workload size increases.

Using FirstSQL as the underlying database, performance is slightly better than PostgreSQL for the executions of the workloads that it supports. Our experiments with FirstSQL illuminate several key observations regarding main-memory databases and symmetric publish/subscribe. First, since the repeated nature of the match operation effectively maintains a warm cache when the total number of subscriptions and publications is small, the main-memory database does not significantly outperform PostgreSQL for those cases. Second, the size of the workload supported by a main-memory DBMS is limited by the available memory as one would expect, but this limitation is reached on smaller workload sizes than for which a conventional DBMS's performance starts to degrade. This is because the main-memory DBMS maintains all data in memory – including the base relations and all index structures – while the bufferpool manager of the conventional DBMS may selectively retain only the data structures needed to execute the actual query. Based on these observations, we conclude that there is little benefit from implementing symmetric publish / subscribe using a main-memory DBMS except in specific applications with small numbers of publications and subscriptions and predictable workloads. In the case of our experiments, the limitations of the main-memory DBMS prevented data collection for all but the smallest workloads.

6.1 Effect of Publication Batching

During run time the system may buffer the stream of incoming operations, only occasionally executing the match operation. This technique may increase system throughput by amortizing the fixed costs of processing those operations across the batch.

Figure 6 shows the effect of publication batching on matching time for our various workloads. In this case both the fixed and proportional workloads scale well. (Recall, the number of matches generated by the proportional workload is proportional to the number of subscriptions but constant with respect to the number of publications.) Both the large-intermediate-results and fixed geometric

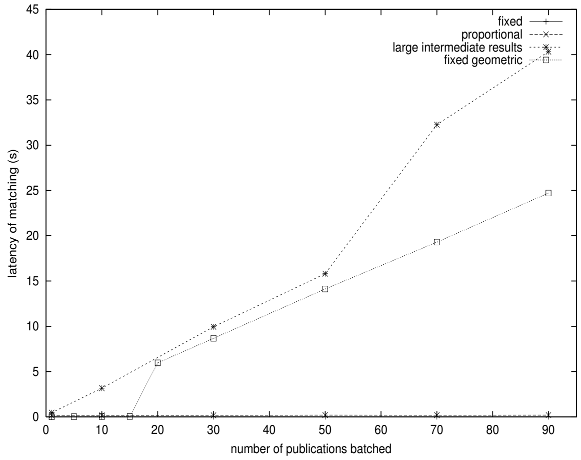

Figure 6: Cost of publication using batch processing of multiple publications.

workloads scale quite poorly when publications are batched. This behavior is not surprising for large-intermediate-results since the intermediate result size scales with the number of publications but is unexpected for the fixed geometric workload. For the geometric type it seems that the PostgreSQL optimizer is unable to accurately estimate the cost of various join methods and uses an index join only for very small numbers of publications. For larger publication batches it instead uses a more costly repeated sequential scan of the publication table. In this case better statistics collection or forcing PostgreSQL to use a particular query plan might result in better matching performance.

Figure 7 shows how matching throughput is dependent on the batching of publications. It is important to note that this indirect throughput measurement only indicates the performance of the matching function and does not necessarily reflect the overall throughput of the running system. (The overall system additionally needs to parse publications, record the constraint information to the base relations, and maintain the indexes.) As expected from the overall matching cost, batching is highly effective for the fixed and proportional workloads, but mostly ineffective for the large-intermediate-results and geometric workloads.

The effect of resource limitations and the query optimizer is quite significant when considering the effect of batching on publisher throughput. Because of the non-linear effect of publication batching on matching performance, overall publication throughput may decrease if the system attempts to process too many publications in a batch. This result is best observed in the throughput measured from the fixed geometric workload, in which the query optimizer selected an inferior execution plan as the size of the publication batch increased. For that workload peak throughput is achieved when batches of only about 15 publications are used, with throughput decreasing substantially for larger batches.

In general, the largest benefit of batching is obtained immediately for all workloads by simply amortizing the fixed costs of matching over multiple publications. For some workloads these fixed costs are very high compared to the cost per publication, in which case large batches may be used effectively. As the number of publications per batch increases the additional advantage of batching more publications is eventually outweighed by the cost of processing the publications, particularly when the resources required by matching (or estimates of resources required) exceed the available hardware resources.

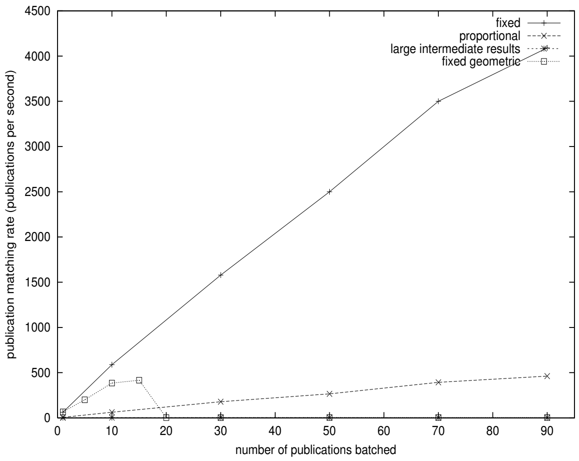

Figure 7: Publication throughput as the number of publishers increases.

7 Extensions

In this section we consider various transactional semantics that could be implemented with our symmetric publish / subscribe design and explore how symmetric publish / subscribe could be implemented efficiently to increase performance in a distributed cluster environment.

7.1 Transaction Semantics

Classical publish / subscribe semantics consist of the following timeline: (1) a large collection of subscriptions added to the subscription set over time using ACID semantics for each addition and the publication set is empty, (2) a publication arrives, (3) in a single ACID transaction, (a) the publication set is updated with the publication, (b) the match is computed, and (c) the publication is deleted from the publications set, then the transaction ends and (4) clients are notified. (These semantics assume unreliable notification. For reliable notification, the clients to be notified are written to another table with the transaction boundary of step 3.)

However, many other semantics are attractive depending on the application. Because our system rests entirely on a relational database, other transactions semantics are easily encoded. For the auction example in the introduction, the following semantics are desirable: publications are used for offers for sale and subscriptions are used for offers to buy. Every publication and subscription contains an identifier to indicate the product under negotiation and a time stamp that totally orders publications and subscriptions. (The time stamp is used to resolve the winner of identical multiple subscription matches that occur during any match cycle.) An additional table is kept for the current best offer to buy for any offer for sale. ACID transactions are used to update the publications and subscription tables. The system continuously executes a transaction that executes the matching operation and updates the best offer table. Outside of this transaction, publishers and subscribers are notified of any changes in the auction, i.e. notifications are sent whenever the current best offer changes. [Or, all participants simply observe the current best offer through a user interface.] When an auction closes, the publications and subscriptions corresponding to that auction are deleted. Good performance in this case assumes indexes on both the publisher and subscriber tables.

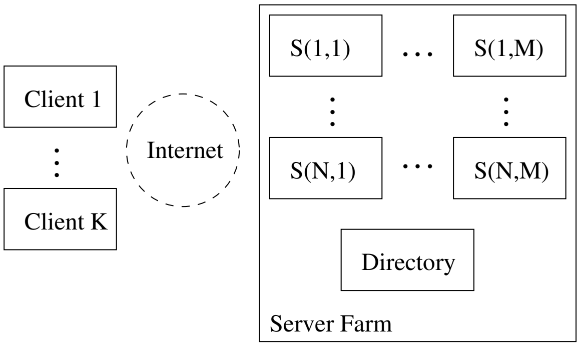

Figure 8: Distributed Architecture

7.2 Distributed Symmetric Publish / Subscribe

In this subsection we present the design of a distributed symmetric publish / subscribe system. Partitioning of the servers is accomplished by modeling this problem as a symmetric publish / subscribe system. This reflective modeling trick simplifies the overall design. Our design supports a large number of loosely connected clients operating over the internet. These clients are supported by a tightly connected server farm consisting of a logical grid of servers and a directory service. By loosely connected components, we mean that ACID transactions are not possible among the components. By tightly connected components, we mean that ACID transactions are possible via standard distributed transaction protocols.

Figure 8 illustrates the distributed architecture. Clients are connected via the internet to a server farm. The server farm consists of N rows of M logical (virtual) servers. (Load balancing is accomplished by assigning physical servers to logical servers. From now on, we simply refer to servers.) Each row consists of M identical servers. Each row is also assigned one partition from $U = \bigcup u_i$. Each u_i is a partition, defined by a constraint c_i that partitions the attributes in U. For example, $c_1 = x < 2$ and $c_2 = x \geq 2$ partitions the attribute x. The directory service contains the current version of U and a version number Z. The version number is a counter that is incremented each time U changes.

In the distributed case, clients now have an addition join function for joining the publish / subscribe system.

***void* join()** contacts the directory and receives the current (U, Z) pair.

***boolean* publish(c, k, z)** client function is now implemented as

1. Compute match((id, c),U) locally at the client. The result is a single pair id, i where i is the server row responsible for the partition that matches constraint c.
2. Execute publish(c, k, z) on any one of the servers on row i. The server will compare its version with z and return false for the publish function if they do not match.
3. If previous step returns *false*, execute join and repeat. Otherwise, return *true*.

***boolean* subscribe(c, k, z)** client function is implemented in a similar manner.

Modifying U is straight forward through the new(U,U') function. This function implements a reconfiguration of the server farm. In a single transaction, sweep through all constraints in the server farm, moving them to the correct row depending on the results of the match function. Since this operation may take some time, another implementation uses a shadow page copy style of transaction management by making a copy the server farm using the same sweep, and then atomically flipping from the active server farm to the copy in a single transaction. modifications of subscriptions must be disabled during this operation, but not publications.

8 Related Work

A general survey of publish / subscribe appears in [11]. This survey covers many distributed computing issues but does not cover content-based matching in depth.

The theory of constraint databases is outlined in Kanellakis et al. [16]. The particular class of constraints permitted in this paper does not directly map to the taxonomy of that paper since this system is based on operators. However, both papers use point-set constraints and overlapping subclasses of constraints. In particular, Kanellakis et al. describe the connection between constraint representation and spatial data structures.

CCUBE [4, 3] is a constraint database that combines database technology with in-memory linear constraint evaluation (via Simplex). LyriC [2] is the associated query language and object model. The overlap between symmetric publish / subscribe and constraint database functionality is an area of ongoing research.

Many works are concerned with aggregating subscriptions in classical publish / subscribe systems for more efficient content-based processing. Mühl [17] describes an algorithm for merging subscription constraints based on identical conjuncts. Crespo et al. [7] explore optimization algorithms and cost models for subscription aggregation in a multicast environment. Application of these techniques to symmetric publish / subscribe is an open research problem.

The method of counting matched field and value pairs is similar to Yan and Garcia's [19] counting method for Boolean selective dissemination of information profiles. The technique of counting matched conjuncts appears in many works. Conjunctive predicate counting augmented with cache line analysis and other techniques is described in [12].

Our method of embedding multiple different types for a single generic value into a relation is similar to that of Yalamanchi, Srinivasan and Gawlick [18]. This work also describes a powerful generalization where expressions are treated as data and the evaluation of expressions can be combined with standard SQL processing. However, they do not appear to reify constraints, a key issue in the choice of a representation, nor do they use indexes, a key issue in performance.

Franklin et al. [1, 8, 10, 9] introduced and explored a method of compiling subscriptions into an in-memory finite state machine (FSM). The finite state machine represents common path prefixes of different subscriptions only once, thus providing a form of common sub-expression elimination. Matching a published document with subscriptions is implemented by traversing this FSM. The finite state machine methodology inspired several subsequent publications, e.g. [15, 6].

Our work is closely related to and inspired by Fink, Johnson and Hu's work [13] on an auction system that matches buy and sell orders. While auction systems and publish / subscribe systems differ in many details, they share some fundamental questions, such as index construction and its relationship to the complexity of the match operation. Fink et al. combine all constraints into a single large index. Each node of the index corresponds to a constraint attribute. Constructing this index requires choosing an attribute order and thus introduces a bias into the index search. The system described here does not exhibit this bias. However, auction systems and Fink et al. in particular compute the best match between a publisher and a set of subscribers. This problem is an open area of research for symmetric publish / subscribe.

Independently of our work, Fischer and Kossmann [14] analyze a variety of strategies for batching publications for the classical publish / subscribe case. Our batching results confirm that batching is an effective strategy for the symmetric publish / subscribe case. We believe that the various other strategies described by Fischer and Kossman probably apply here as well.

Our work is similar in some respects to Chandrasekaran and Franklin's work on stream queries and data [5] that highly optimizes a particular transaction semantics of publish / subscribe, with the addition of support of query operations matches, maintenance of result sets, time windows, etc. However, this work does not consider constraints for publications.

9 Conclusion

This paper is the first reported investigation into symmetric publish / subscribe systems. These systems permit publications as well as subscriptions to express constraints. The system computes the intersection of publisher constraints with subscriber constraints to determine matches. This additional expressive power provides a new area of research and new possibilities for applications of publish / subscribe systems.

In certain cases, the complexity of the matching algorithm is $O(o^2t)$ with o operators and t types used in constraint expressions, in addition to other factors. We implement symmetric publish / subscribe as an application on a relational DBMS and evaluate our system, showing that symmetric publish / subscribe is practical using current technology. Finally, we evaluate the effect of publication batching on performance and show how the optimal publication batch size for our system is dependent on workload characteristics and hardware resources. Initial batching of publications may amortize the fixed costs of processing across multiple operations, but processing too large a batch may degrade performance.

Overall, our implementation enables response times as low as a few milliseconds for a single publication, and we demonstrate that throughputs of thousands of publications per second should be obtainable for some workloads. We show that the performance cost of symmetric publish / subscribe is highly dependent on the workload, but that this cost is not substantial compared to classical publish / subscribe systems in some instances.

9.1 Future Work

Many issues remain open for this new class of systems. Existing techniques, such as multi-cast management of notification, subscription merging, or postprocessing matching tuple sets may also benefit symmetric publish / subscribe. Given that a publication matches many subscribers, ordering subscribers by best match and notifying the top subscriber is a key question in auction systems.

Improving the performance of the matching function for symmetric publish / subscribe is an open area of research. One very promising approach is the maintenance of materialized views to reduce the overhead incurred by the queries used in intermediate answer generation as well as the final aggregation step.

10 Acknowledgments

Thanks to Christopher Olston for comments on the paper. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHC030029.

References

[1] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. *The VLDB Journal*, pages 53–64, 2000.

[2] Alexander Brodsky and Yoram Kornatzky. The LyriC language: Querying constraint objects. In *SIGMOD '95*, 1995.

[3] Alexander Brodsky, Victor E. Segal, Jia Chen, and Pavel A. Exarkhopoulo. The ccube constraint object-oriented database system. *Constraints*, 2(3,4):245–279, December 1997.

[4] Alexander Brodsky, Victor E. Segal, Jia Chen, and Pavel A. Exarkhopoulo. The ccube constraint object-oriented database system. In *Proceedings of SIGMOD 1999*, 1999.

[5] Sirish Candrasekaran and Michael J. Franklin. Streaming quereies over streaming data. In *VLDB*, 2002.

[6] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.

[7] Arturo Crespo, Orkut Buyukkokten, and Hector Garcia-Molina. Efficient query subscription processing in a multicast environment, 1999. Technical Report http://www-db.stanford.edu/pub/papers/badd.ps.

[8] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *The 18th International Conference on Data Engineering*, pages 341–342, 2002.

[9] Y. Diao and M. Franklin. Query processing for high-volume xml message brokering, 2003. Technical Report, University of California, Berkeley, http://citeseer.ist.psu.edu/diao03query.html.

[10] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[11] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[12] Francoise Fabret, H. Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 115–126. ACM Press, 2001.

[13] Eugene Fink, Josh Johnson, and Jenny Hu. Exchange market for complex goods: Theory and experiments. *Netnomics: Economic Research and Electronic Networking*, 6(1):21–42, 2004.

[14] Peter M. Fischer and Donald Kossmann. Batched processing for information filters. In *ICDE*, 2005.

[15] Ashish Kumar Gupta and Dan Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM Press, 2003.

[16] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proceedings 9th ACM PODS*, 1990.

[17] Gero Mühl. Generic constraints for content-based publish/subscribe. In *CoopIS*.

[18] A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *Conference on Innovative Directions in Research*, 2003.

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

[20] Tak W. Yan and Hector Garcia-Molina. Index structures for information filtering under the vector space model. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 337–347. IEEE Computer Society, 1994.