

Subscribing to the Past in Content-based Publish/Subscribe

Guoli Li
CS Department
University of Toronto
gli@cs.toronto.edu

Vinod Muthusamy
ECE Department
University of Toronto
vinod@eecg.toronto.edu

Arno Jacobsen
ECE and CS Department
University of Toronto
jacobsen@eecg.toronto.edu

ABSTRACT

This paper introduces to the publish/subscribe model the ability to uniformly access data produced in the past and future. The new model can filter, aggregate, correlate and project any combination of historic and future data. A flexible architecture is proposed consisting of distributed and replicated data repositories that can be provisioned in ways to tradeoff availability, storage overhead, query overhead, query delay, load distribution, parallelism, redundancy and locality. Evaluations in a distributed testbed show that different provisioning policies perform better with read or write heavy workloads. In addition, a dynamic query routing algorithm is used to optimize the location where sub-queries are evaluated, yielding traffic reductions of up to 72%.

1. INTRODUCTION

The publish/subscribe paradigm [2, 5, 15, 13, 9, 7] provides a simple communication metaphor for entities that require complex interaction patterns while remaining decoupled. Data producers *publish* data to a *broker* which forwards the data to consumers who have *subscribed* to them. In this way, producers and consumers remain anonymous. The expressiveness of the subscription filtering constraints differentiates pub/sub implementations. While early incarnations based on simple group communication or channel-based models [13] have been successfully employed in industry, they are proving insufficient for a variety of applications demanding more powerful *content-based* pub/sub filtering and correlation capabilities. Applications in this space include business process execution [16], workflow management [5], business activity monitoring [6], stock-market monitoring [18], selective information dissemination and RSS filtering [15], complex event processing for algorithmic trading [8], and network monitoring and management [6].

A limitation of the traditional pub/sub model is that it only delivers to subscribers those publications produced after the subscription was issued. It is a model to query the future. There are many application contexts, however, where access to publications from the past is necessary, such as for auditing purposes, replaying a business process execution to debug it, or tracing system events to perform root cause analysis. Even more interesting uses arise when data from the past can be correlated with those in the future. For example, an algorithmic trader may wish to be notified when a stock behaves as it did during a recent economic downturn (incoming stock quotes are being correlated with those from the past), or a credit-card company may need to know if a client's usage pattern differs significantly from those on

the same day last week. While such analysis can be performed periodically with traditional databases, the pub/sub model can detect these complex patterns in real-time and immediately notify subscribers as they occur.

Another scenario involves a border security agency who prevent sensitive goods, such as radioactive material, from entering the country [11]. Containers entering at around 30 border points by ship, rail, and truck are scanned in sensing stations, where traces of radioactive material trigger alarms. There are however a number of genuine goods that set off the sensors, and to filter out false alarms, it is crucial that the detection of radioactive material and the container's serial number are correlated with a shipping manifest submitted by the shipping company ahead of time. This scenario serves as a running example throughout the paper.

The challenge is to support access to both historic and future publications¹ through a unified pub/sub interface while still preserving the desirable properties of the pub/sub model. For example, in content-based pub/sub, no direct addresses of participants are available, and so directly querying the databases is not an option. The client should not even know which database to query. Also, *composite subscriptions* [9] that allow correlations, or joins, across publications or publishers should work with any combination of historic and future data. Furthermore, to preserve the scalability of distributed pub/sub architectures, the historic data repository should not be centralized. But replicating or distributing the repositories presents further challenges such as synchronizing the replicas.

This paper extends the distributed content-based pub/sub model to support access to publications from the past and future. Sec. 3 proposes an expressive SQL-like subscription language that can express constraints, correlations, projections and aggregations on any combination of future and historic publications in a unified manner. Sec. 4 presents an architecture that supports the capabilities of the above language and develops algorithms that dynamically determine the optimal points for composite subscription correlation operations based on traffic volumes and network conditions. Sec. 5 addresses the management of the databases that serve as historic publication repositories, including the ability to partition, replicate, distribute, and synchronize the databases. Finally, Sec. 6 experimentally evaluates the implementation of the system in a distributed setting.

2. RELATED WORK

¹Publications issued before a subscription are considered historic. The past and future are with respect to the time the subscription is issued.

To the best of our knowledge, this work is among the first to unify access to publications in the past and future in the pub/sub paradigm. In this section, we position our work with research on querying historic and future data.

Pub/Sub: The pub/sub model supports push-based decoupled many-to-many interaction patterns between information producers (*publishers*) and consumers (*subscribers*) [2, 12, 5, 9]. Data (*publications*) are pushed to subscribers who express their interests using *subscriptions*. Early systems were channel-based and delivered all publications on a particular channel to interested subscribers, whereas the more expressive content-based model allows subscriptions to express constraints on the content of publications.

In all of the above systems, subscribers are only notified of publications issued after their subscriptions. None can retrieve publications from before the registration of subscriptions. In this paper, we propose a unified way to access both future and historic publications.

To improve subscription expressiveness, Badrish *et al.* [1] separate pub/sub processing into *subscription processing* and *notification dissemination*. Publications are first inserted into a database where subscriptions are evaluated, and then matching results are disseminated over a pub/sub overlay. Processing subscriptions at a database facilitates aggregation and join operations, but subscriptions are still restricted to matching publications in the future. Furthermore, the databases can be distributed but not replicated.

In this paper, we support subscriptions for both future and historic publications in a way that future publications are delivered directly without an intervening database, and aggregations and joins are processed within the distributed pub/sub overlay (for future publications), or at the databases (for historic data). We also exploit the opportunity to support projection capabilities missing in typical pub/sub systems. As well, the databases can be easily replicated in a flexible manner to tradeoff various availability, network traffic, query response, and storage overhead goals.

Stream processing and continuous queries: Continuous queries [17, 4, 10, 3] are issued once and run “continually” over the database. Tapestry [17] supports such queries over an append-only database via a limited SQL-like language, but does not support aggregations or joins. NiagaraCQ [4] uses an incremental query evaluation method but is not limited to append-only data sources. It uses a static query optimizer, and operators from different queries can be shared. CACQ [10] adds the notion of *tuple-lineage* to share queries beyond common query plan subtrees, and uses the *eddy* operator to continuously adapt query workload, data delivery rates or system performance. PSoup [3] treats data and queries symmetrically as streams, allowing queries to access both data that arrived before the query registration and data in the future. However, PSoup uses a main-memory data structure limiting historic data to memory size. Also as with any centralized architecture, it presents a failure and performance bottleneck.

Content-based pub/sub differs from continuous queries and stream processing. First, a subscription may match publications from an anonymous and unknown number of data sources which may not conform to a predefined schema, whereas queries on streams typically explicitly identify the source streams and can rely on data conforming to known schemas, greatly simplifying query processing and routing

decisions. Second, we allow clients to subscribe to both historic and future data. The historic data are stored in a distributed set of databases that are not limited by main memory. Third, none of the above systems address data space partitioning and replication. We propose partitioning algorithms, define consistency properties that conform to pub/sub semantics, and develop protocols to maintain consistency among replicas in the pub/sub system.

Relational Database: This paper is influenced by much of the work in relational and distributed database systems [14], including the SQL language features, database partitioning, and query planning. However, differences between the relational query and pub/sub models, as outlined in Sec. 3.1, prevent a direct application of these concepts.

The Padres System: Our work is based on the PADRES² distributed content-based pub/sub system. Brokers form an overlay network. A publisher first issues the schema of the publications it intends to publish as an advertisement, which is broadcast and forms a spanning tree in the overlay rooted at the publisher. Subscriptions from subscribers are then propagated towards the roots of those trees where there is potential for publications to match the subscription. Finally, a publication that matches a subscription is forwarded along the subscription tree until it reaches the subscriber.

3. THE SUBSCRIPTION LANGUAGE

We extend pub/sub to allow subscribing to publications in the future and the past. As the former problem has been the focus of pub/sub and the latter addressed by relational databases, it is instructive to compare the two models.

3.1 Basic Operations

Fundamentally, both pub/sub and databases are concerned with getting data from producers to interested consumers, and support the following four operations.

Define schema: An SQL administrator issues a `CREATE TABLE` command to define a data template. Likewise, a pub/sub producer issues an *advertisement*.

Produce data: SQL data producers `INSERT` tuples into tables, and pub/sub producers *publish* publications.

Query data: SQL consumers query tuples with `SELECT` statements; pub/sub consumers *subscribe* to publications.

Consume data: The results of SQL `SELECT`s are returned to consumers as query *results*, whereas matching publications are delivered to pub/sub subscribers as *notifications*.

The principal difference in the above operations between database and pub/sub clients is that the order these operations are invoked: in the database model data is first produced, and queries return data produced in the past after which the query is complete, whereas pub/sub queries are evaluated continuously and return data produced after the query is issued. This paper extends the pub/sub model to support querying data from the past.

We now point out some key differences between the relational database and pub/sub models.

Data format: While both models structure data as attribute-value pairs (or tuples), pub/sub publications are not relational. Moreover, publications may include only a subset of the attributes defined in the corresponding advertisement schema. This is equivalent to null values in database tuples.

²PADRES is a pseudonym for this double-blind review.

Data collections: Database tuples, organized in tables, can be added (*INSERT*) or overwritten (*UPDATE*) by any producer. In pub/sub, data is only loosely organized as ordered publications corresponding to a particular advertisement. Furthermore, updates are not supported and the semantics are somewhat like an append only database.

Query source: While an SQL *SELECT* statement only queries those tables explicitly specified, a pub/sub subscription queries all publications.

Query expressiveness: Most pub/sub languages cannot express SQL notions of projections or joins.³

3.2 Requirements

As we describe in Sec. 4, in this paper pub/sub clients query future data in the usual way, but retrieve historic data from a set of provisioned databases that serve as repositories of previously published data. This difference, however, should not be visible to clients and we seek a single language to query both historic and future data. This language should satisfy the following requirements.

Retain pub/sub semantics: The language should support the current content-based pub/sub model, including the predicate constraints and queries for future data.

Future and historic queries: There should be a unified way to query any combination of future or historic data.

Simple mapping to SQL: Since we use standard relational databases to store historic data, it is desirable for queries to be easily mapped to SQL if querying historic data.

Joins and projections: The query language should have some limited ability to express joins and projections.

To avoid introducing an unfamiliar language, we restrict ourselves to extending either SQL or a pub/sub language. Since SQL is more expressive and is easy to “map” to SQL, we choose the former option, and call our language Hybrid-SQL (HSQL). We emphasize it is not our goal to develop a model that is a superset of databases and pub/sub, but to extend the pub/sub semantics with some concepts from SQL. For example, our data model is not relational, new data do not overwrite old ones, and queries do not support the full SQL feature set.

3.3 HSQL Language

The HSQL language is described using the framework of the four operations outlined above. To remind the reader that we are largely retaining pub/sub semantics, we will use pub/sub terms such as subscriptions and publications as opposed to queries and tuples.

The examples below will use a scenario where a country’s border security services monitors shipments into a country. Three sources of events are available: radioactive readings of shipments by sensors, shipping manifests issued by shipping firms, and internal audit reports on these firms.

Define schema: Each data producer (i.e., publisher) specifies a template that describes the publications it will publish. This is traditionally done with an advertisement message, but we adopt the equivalent SQL table creation statement.

```
CREATE TABLE (attr op val[, attr op val]*)
```

HSQL’s *CREATE TABLE* differs from that in SQL. Tables are unnamed since they need not be referred to by subscriptions or publications. Also, the range of values of each at-

tribute (or column) can be specified. Moreover, regardless of the attribute value constraint, each attribute can implicitly be a null value. The following statements setup the three event sources in our scenario. Sensor readings include the shipment id and a non-negative radioactivity level less than 10; shipping manifests indicate, for a given shipment, the expected contents of the shipment, and the shipping firm; and audit events indicate the trust level of a shipping firm.

```
CREATE TABLE (type = reading, shipID = *, level < 10)
CREATE TABLE (type = manifest, shipID = *, firm = *, content = *)
CREATE TABLE (type = audit, firm = *, trust >= 0)
```

In the above example, * is a wildcard that indicates the corresponding attribute may have any value.

Produce data: A publication is produced using a construct similar to SQL’s *INSERT* statement.

```
INSERT (attr[, attr]*) VALUES (val[, val]*)
```

The following publications are compatible with the advertisement schema defined above.

```
INSERT (type, shipID, level)
VALUES (reading, 123, 4)
INSERT (type, shipID, firm, content)
VALUES (manifest, 123, ACME)
```

Notice that only a subset of attributes defined in the schema need to be specified.

Query data: Subscribers issue *SELECT* statements to query both historic and future publications.

```
SELECT [ attr | function ]*
[FROM event]
WHERE [ attr op val]*
[HAVING function* ]
```

The *FROM* and *HAVING* clauses are optional and are used to express joins and aggregations as described below.

A traditional pub/sub subscription for future publications would look as follows in HSQL.

```
SELECT *
WHERE type = reading, level > 3
```

Note that the above statement does not query a single table, so the results may have any number of attributes. The only guarantee is that all notifications will have the *type* and *level* attributes with values constrained as specified.

Subscribers can specify time constraints using reserved attributes *start_time* and *end_time* in the *WHERE* clause. Time constraints can be used to query for publications from the past, the future, or both. For example, upon a heightened security alert, it may be necessary to also retrieve suspicious sensor readings shortly before the alert was issued. The following subscription queries data in a time window that begins one hour before the time the query is issued and extends into the future.

```
SELECT *
WHERE type = reading, level > 3,
start_time = NOW - 1h, end_time = NOW + 4h
```

The system internally splits the above subscription: one purely historic subscription that is evaluated once, and one ongoing future subscription. A subscription for both historic and future data is a *hybrid* subscription.

Pub/Sub composite subscriptions [9] can be expressed with simple join conditions. The event correlation is supported using the *FROM* clause, where the event pattern can be specified using Boolean expressions. To avoid false alarms, the subscription below (which is both hybrid and composite) will only report on shipments that are detected as radioactive, but whose material is not expected to be so. Furthermore, manifests are only trusted if the firm has been audited as trusted anytime from two months ago.

³PADRES supports a form of join with *composite subscriptions*, and in this paper, we add projection capabilities.

```

SELECT *
FROM e1 AND e2 AND e3
WHERE e1.type = reading, e1.level > 3,
e2.type = manifest, e2.content != fertilizer,
e3.type = audit, e3.trust > 7,
e3.start_time = NOW - 2 months,
e1.shipID = e2.shipID, e2.firm = e3.firm

```

The event in the FROM clause specify that three different publications are required to satisfy this query, and each publication must qualify the WHERE constraints. The three publications may come from different publishers, and may conform to different schema.

Notice that a composite subscription can collect, correlate, and filter publications in the network. Without this feature, a user must retrieve sensor readings for all potentially dangerous sensor shipments, and then issue a historic query for the associated manifest, and then another query for the associated audit record. This would be expensive (both for the user and in terms of network traffic) in cases where the sensor events are generated frequently.

Event aggregation is supported in HSQL as well. The HAVING clause can specify constraints across a set of matching publications. The functions $AVG(a_i, N)$, $MAX(a_i, N)$, and $MIN(a_i, N)$ compute the appropriate aggregation across attribute a_i in a window of N matching publications. The window may either slide over matching publications, or be reset when the HAVING constraints are satisfied. For example, perhaps because sensors are faulty, the following subscription will only match when the average reading from ten different sensors indicate radioactive material.

```

SELECT *
WHERE type = reading
HAVING AVG(level, 10) > 3
GROUP BY shipID

```

Any attributes specified by functions in the HAVING clause must appear in the publication. So, an implicit `price = *` condition is added to the WHERE clause above. Also, the GROUP BY clause has the same semantics as in SQL and serves to constrain the set of publications over which the HAVING clause operates.

Consume data: Notification semantics do not constrain notification results, but transform them. HSQL supports projections and aggregations over matching publications to simplify notifications delivered to subscribers and reduce overhead by eliminating unnecessary information.

Projections are a useful feature rarely supported in pub/sub. Notifications may include a subset of attributes in matching publications with the SELECT clause.

```

SELECT shipID, level
WHERE type = reading, level > 3

```

The above subscription only guarantees that matching publications include the `reading` and `level` attributes. Notifications may contain only a subset of the `shipID` and `level` attributes.

Aggregation functions may appear in the SELECT clause. For example, the following subscription returns the average sensor reading for all shipments.

```

SELECT shipID, AVG(level, 10)
WHERE type = reading
GROUP BY shipID

```

Again, an implicit `level = *` condition is added to the WHERE clause since the `level` attribute appears in a function.

Note that functions in the SELECT and HAVING clauses have different semantics. In the preceding subscription, for every ten matching publications, a single notification is delivered to the subscriber containing the average radioactive level, whereas the subscription with the HAVING clause we saw ear-

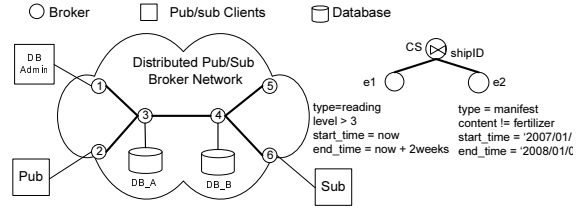


Figure 1: Historic Data Access

lier delivers all publications within a window of ten whose average satisfies the specified condition.

3.4 Discussion

The HSQL language unifies access to publications from the past and future. It retains predicate-based pub/sub subscription functionality, and provides a consistent way to query any combination of future and historic publications with simple conditions on the `start_time` and `end_time` attributes. Evaluating future HSQL subscriptions is straightforward, and mapping historic subscriptions into SQL statements is simplified by basing the language on SQL. An SQL-like language also afforded a familiar construct to express simple joins (composite subscriptions), and notification semantics such as projections and aggregation constraints, which are not typically supported by pub/sub systems.

4. ARCHITECTURE AND ROUTING

We propose a new pub/sub system that allows subscribers to access both future and historic data with a single interface as described in Sec. 3. The system architecture, shown in Fig. 1, consists of a traditional distributed pub/sub overlay network of brokers and clients, to which a set of databases are added. The databases are provisioned to sink a specified subset of publications, and to later respond to queries

Subscriptions for future publications are routed and handled as usual [2, 5, 12]. To support historic subscriptions, databases are attached to a subset of brokers as shown in Fig. 1. The set of possible publications, as determined by the advertisements in the system, is partitioned and these partitions assigned to the databases. A partition may be assigned to multiple databases to achieve replication, and multiple partitions may be assigned to the same database if database consolidation is desired. Partition assignments can be modified at any time, and any replicas will synchronize among themselves. The only constraint is that each partition be assigned to at least one database so no publications are lost. Partitioning algorithms as well and partition selection and assignment policies are described in Sec. 5 and evaluated in Sec. 6.

The remainder of this section describes how subscriptions retrieve, filter, and correlate combinations of historic and future publications from multiple sources.

4.1 Subscription Routing

Subscriptions can be *atomic* expressing constraints on single publications, or *composite* expressing correlation constraints over multiple publications. This section discusses their routing under the extended pub/sub model.

Atomic Subscription Routing: When a broker receives an atomic subscription, it checks the `start_time` and `end_time` attributes. A future subscription is forwarded to

potential publishers using standard pub/sub routing [2, 5, 12]. A hybrid subscription is split into future and historic parts, with the historic subscription routed to potential databases as described next.

For historic subscriptions, a broker determines the set of advertisements that *overlap* the subscription, and for each partition R_i , selects the database with the minimum routing delay. The subscription is forwarded to only one database per partition to avoid duplicate results. When a database receives a historic subscription, it evaluates it as a database query, and publishes the results as publications to be routed back to the subscriber. Upon receiving an END publication after the final result, the subscriber's host broker unsubscribes the historic subscription. This broker also unsubscribes future subscriptions whose *end.time* has expired.

Adaptive Routing: Topology-based composite subscription routing [9] evaluates correlation constraints in the network where the paths from the publishers to subscriber merge. If a composite subscription correlates a historic data source and a publisher, where the former produces more publications, correlation detection would save network traffic if moved closer to the database, thereby filtering potentially unnecessary historic publications earlier in the network. Based on this observation, we propose the adaptive composite subscription routing algorithm described next.

The WHERE clause constraints of a composite subscription can be represented as a tree where the internal nodes are operators, leaf nodes are atomic subscriptions, and the root node represents the composite subscription. The composite subscription example from Sec. 3 is represented as the tree in Fig. 1. A recursive algorithm computes the destination of each node in the tree to determine how to split and route the subscription. The algorithm traverses the tree as follows: if the root of the tree is a leaf, that is, an atomic subscription, the atomic subscription's next hop is assigned to the root. Otherwise, the algorithm processes the left and right children's destination trees separately. If the two children have the same destination, the root node is assigned this destination, and the composite subscription is routed to the next hop as a whole. If the children have different destinations, the algorithm estimates the *total routing cost* (to be presented shortly) for potential candidate brokers, and the minimum cost destination is assigned to the root. If the root's destination is the current broker, the composite subscription is split here, and the current broker is the *join point* and performs the composite detection. The algorithm assigns destinations to the tree nodes bottom up.

Dynamic Join Point Movement: When network conditions change, join points may no longer be optimal and should be recomputed. A join point broker periodically evaluates the cost model, and upon finding a broker able to perform detection cheaper than itself, initiates a join point movement. The state transfer from the original join point to the new one includes routing path information and partial matching states. Each part of the composite subscription should be routed to the proper destinations so routing information is consistent. Publications that partially match composite subscriptions stored at the join point broker must be delivered to the new join point.

4.2 Join Point Placement Cost Model

A broker routing a composite subscription makes locally optimal decisions based on a cost model that incorporates

its own resource consumption and that of its neighbors. Resources such as memory, CPU, and network traffic can be modeled. Suppose composite subscription CS is split at broker B . The *total routing cost* for CS is

$$TRC_B(CS) = RC_B(CS) + \sum_{i=1}^n RC_{B_{N_i}}(CS_{B_{N_i}}),$$

and includes the *routing cost* of CS at broker B and those neighbors with publications contributing to the composite subscription. $CS_{B_{N_i}}$ denotes the part of CS routed to broker B_{N_i} , and may be atomic or composite.

Routing cost at each broker: The cost of CS at a broker includes the time to match publications (from n neighbors) against CS , the time these publications spent in the broker's input queue, and the time that matching results (to m neighbors) spend in output queues:

$$RC_B(CS) = \sum_{i=1}^n T_{in} * |P(CS_{B_{N_i}})| + \sum_{i=1}^n T_{matching} * |P(CS_{B_{N_i}})| + \sum_{i=1}^m T_{out_i} * |P(CS)|,$$

where $T_{matching}$ denotes the average matching time at a broker, and T_{in} and T_{out_i} are the average time messages spent in the input and output queue to the i^{th} neighbor. $|P(S)|$ is the cardinality of (atomic or composite) subscription S , and is defined below.

To compute the cost at a neighbor, brokers periodically exchange T_{in} and $T_{matching}$, and use this information in an M/M/1 queuing model to estimate queuing times at neighbor brokers as a result of the additional traffic attracted by splitting a composite subscription there.

Subscription cardinality: The cardinality of a subscription S is the number of matches of S per unit of time. Let the attribute set of advertisement A be $Attr(A) \leftarrow \{a_1, a_2, \dots, a_n\}$. To evaluate the cost model we need the cardinality of $\varphi(A)$ per attribute ($|\Pi_{a_i} \varphi(A)|$)⁴ and the distribution of values in an attribute domain ($dom[a_i]$).

To estimate the cardinality of a publication set matching a subscription s , we define a *selection factor* as $|\sigma_s \varphi(A)| = sf_A(s) * |\varphi(A)|$. An atomic subscription's selecting factor is $sf_A(s) = \prod_{i=1}^n sf_A(p_i)$, where p_i is the predicate of attribute a_i and $sf_A(p_i)$ is the selecting factor of predicate p_i in subscription s . The selection factors of individual predicates are computed based on the predicate's operator and the distribution of attribute values d_{a_i} across publications, as shown in Table 1.

$$\begin{aligned} \text{Let } D &= \int_{Min(dom[a_i])}^{Max(dom[a_i])} d_{a_i} \\ sf_A(a_i = val) &= d_{a_i}(val) / D \\ sf_A(a_i > val) &= \int_{val}^{Max(dom[a_i])} d_{a_i} / D \\ sf_A(a_i < val) &= \int_{Min(dom[a_i])}^{val} d_{a_i} / D \end{aligned}$$

Table 1: Selection Factor

An advertisement's attribute distributions are disseminated as a histogram as part of the advertisement. We can now calculate the cardinality of an atomic subscription S that intersects advertisements $\{A_1, A_2, \dots, A_q\}$, as $|P(S)| = \sum_{i=1}^q r_i * sf_{A_i}(S)$, where r_i is the publication rate associated with advertisement A_i .

The selection factor of a composite subscription with an attribute join is $sf(A_1 \bowtie_{a_i} A_2) = \frac{|\Pi_{a_i} A_1 \bowtie_{a_i} A_2|}{|\varphi(A_1)| * |\varphi(A_2)|}$. The cardinality of a composite subscription $CS = S_l \text{ op } S_r$ is:

$$|P(CS)| = \begin{cases} |P(S_l)| + |P(S_r)| & \text{if } op = \parallel; \\ \min(|P(S_l)|, |P(S_r)|) & \text{if } op = \&; \\ sf(A_1 \bowtie_{a_i} A_2) * |s_1 \& s_2| & \text{if } op = \bowtie_{a_i}. \end{cases}$$

⁴Estimated based on publications over a period of time.

5. DATABASE PARTITIONING

We propose a new pub/sub model where databases provisioned as part of the broker network store publications and respond to historic subscriptions. The databases are integrated such that subscribers access both future and historic data with a single interface as described in Sec. 3. Databases can store different sets of publications for load balancing or administrative purposes, and databases can be setup as replicas for availability or fault tolerance. We address publication space partitioning in Sec. 5.1, and the management of database replicas in Sec. 5.2.

5.1 Publication Space Partitions

Partitioning the publication space allows us to provision one or more databases to store publications belonging to the partition and respond to queries for them. To simplify the processing of historic subscriptions, we only consider horizontal partitioning here, and reserve vertical partitioning for future work.

Horizontal Partitioning: An advertisement (i.e., the CREATE TABLE schema definition) describes the publication space of a publisher. The global publication space $\wp(\mathbb{A}) = \bigcup \wp(A_i)$, where A_i is an advertisement and $\wp(A_i)$ is the publication set induced by A_i , can be divided into partitions, which are the smallest units maintained by a database. Publication space partitioning in the pub/sub model differs from database partitions in that we partition a global space instead of individual tables. Furthermore, we need not consider referential integrity constraints. A formal definition of a partition is given below.

Definition 1. Publication Space Partition – A partition is a division of the publication space $\wp(\mathbb{A})$ into distinct publication sets, denoted as $R = \sigma_F(\wp(\mathbb{A}))$, where F is a *selection formula* in form of *predicate* conjunctions.

That is, a partition is a projection of the publication space on selection formula F , where publications in partition R satisfy the constraints in F . A partitioning of the publication space should be complete and disjoint as defined below.

Completeness: Every publication p belongs to at least one partition R_i . Formally, $\forall p \in \wp(\mathbb{A}), \exists i$ such that $p \in R_i$.

Disjointness: The publication sets of partitions do not overlap. Formally, $\forall p \in R_i, p \notin R_j, i \neq j$.

The partitioning algorithm takes as input the set of advertisements in the system, and a set of predicates observed in query workloads. First, the algorithm computes a set of simple predicates that will form the selection formulas. An iterative algorithm such as COM_MIN [14] can generate a *complete*⁵ and *minimal*⁶ set of predicates P' given a set of predicates P . The pub/sub definitions of completeness and minimality differ, and we adjust the COM_MIN algorithm accordingly. A predicate set is *complete* if it covers all attributes and values defined in the publication space, and is *minimal* if predicates on the same attribute do not overlap. Since publications may have any subset of attributes defined in the advertisement, and there is no *primary key* concept, every attribute must be used to partition the publication space to guarantee completeness. Predicates are grouped according to attributes. Predicates across groups are orthogonal and predicates within groups are minimal.

⁵In distributed databases, a set of predicates is *complete* if there is an equal probability of access by every application to any tuple belonging to any partition.

⁶Every predicate is *relevant* in determining a partition.

Algorithm 1 Uniform Partition Accessibility

Require: F : selection formulas, δ : accessibility deviation
Ensure: Std. dev. of partition access probabilities $< \delta$

```

1:  $R \leftarrow \{R_i | \sigma_{f_i} \wp(\mathbb{A}), f_i \in F\}$ 
2: repeat
3:    $\bar{\rho} \leftarrow \frac{\sum_{i=1}^{|R|} \rho(R_i)}{|R|}$ 
4:    $\sigma \leftarrow \sqrt{\frac{\sum_{i=1}^{|R|} (\rho(R_i) - \bar{\rho})^2}{|R|}}$ 
5:   if  $\sigma > \delta$  then
6:      $R_s \leftarrow \text{Max}(\rho(R_i), i = 1 \sim |R|)$ 
7:     Select an attribute with  $\text{Max}(|\prod_a R_i|)$  and  $p_a$ , so that  $f'_i =$ 
        $f_i \wedge p_a$  and  $f''_i = f_i \wedge \neg p_a$ 
8:      $F \leftarrow F - f_i + f'_i + f''_i$ 
9:      $R \leftarrow R - R_i + R_{f'_i} + R_{f''_i}$ 
10:  end if
11: until  $\sigma \leq \delta$ 
12: return  $R$ 

```

Next, the algorithm derives a set of selection formulas, which are conjunctions of orthogonal predicates, as candidates for partitioning. Finally unnecessary selection formulas are eliminated. For example, for selection formula f , if $\wp(\mathbb{A}) \wedge \wp(f) = \Phi$, then partition R_f is empty, no publications match f , and we need not consider f during partitioning.

The resulting partitions are guaranteed to be complete and disjoint because the algorithm uses a set of complete and minimal predicates.

Uniform Partition Accessibility: To balance load, a uniform distribution of partition access by historic subscriptions is desirable. To achieve this, we estimate the *access probability* $\rho(R_i)$ of a partition R_i based on queries observed over a period of time T as

$$\rho(R_i) = \frac{\sum_{s_j} \rho(R_i(s_j)) * |\sigma_{s_j} \wp(\mathbb{A})|}{|\wp(\mathbb{A})|},$$

where subscription s_j is issued within T , and the access probability of R_i by subscription s is $\rho(R_i(s)) = \frac{|\sigma_{f_i} \wp(s)|}{|\sigma_s \wp(\mathbb{A})|}$.

Computing $\rho(R_i(s))$ requires two assumptions: independence of predicates in the selection formula f and subset proportionality conservation. The former is denoted as

$$\rho(R_{p_1 \wedge p_2}(s)) = \rho(R_{p_1}(s)) * \rho(R_{p_2}(s)) \quad (1)$$

and the latter assumption is represented as

$$\frac{\rho(R_{p_1 \wedge p_2}(s))}{\rho(R_{p_1 \wedge \neg p_2}(s))} = \frac{|\sigma_{p_1 \wedge p_2} \wp(\mathbb{A})|}{|\sigma_{p_1 \wedge \neg p_2} \wp(\mathbb{A})|}. \quad (2)$$

By algebraic substitution and transformation of the above two equations, we obtain

$$\rho(R_{p_1 \wedge p_2}(s)) = \rho(R_{p_1}(s)) \frac{|\sigma_{p_1 \wedge p_2} \wp(\mathbb{A})|}{|\sigma_{p_1} \wp(\mathbb{A})|} = \rho(R_{p_1}(s)) \rho(R_{p_2}/R_{p_1}),$$

where $\rho(a/b)$ is the conditional probability of event a given event b .

Algorithm 1 computes partitions such that the standard deviation of partition access probabilities is less than a given δ . Until the standard deviation falls below δ , the algorithm splits the partition with highest accessibility along the attribute with the maximum number of domain values.

Example: We demonstrate the partitioning algorithms with the previous example. To simplify the example, we use a subset of attributes to describe the publication space.

```

CREATE TABLE (shipID = *, level >= 0)
CREATE TABLE (shipID = *, firm = *, content = *)

```

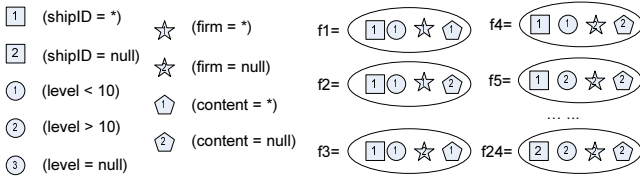


Figure 2: Partition Example

Suppose we discover the complete and minimal set of predicates⁷ from a subscription workload, as shown in Fig. 2. Choosing one predicate per attribute, there are $2 \times 3 \times 2 \times 2 = 24$ selection formula candidates, including those in Fig. 2. We eliminate candidates using implication rules derived from advertisements. One rule is to eliminate selection formulas that do not follow any advertisement schema. For example, R_{f_1} is not defined by any advertisements and can be removed. Some partitions may be empty because no publication in these partitions will be produced. For example, since all readings have $level \leq 10$, R_{f_5} is empty, and f_5 can also be eliminated from the candidate set. Hence, the publication space is divided into six partitions based on the remaining selection formulas. The access probability of the partitions are calculated based on observed workloads and the model discussed above. If partitions are not evenly accessed, a large partition, say R_{f_4} , can be further split by, for example, predicates ($level < 5$) and ($level \geq 5$).

Discussion: The granularity of partitions may range from a single partition for the global space to one per possible publication. Increasing partition granularity presents trade-offs with the overhead of managing more partitions, potential parallelization of partition access, and performance overhead of routing the corresponding advertisement issued by the database. The appropriate number of partitions is application-specific and can be tuned with different sets of predicates input to the partitioning algorithm, or with an appropriate choice of δ .

Partitions can not only be distributed among databases, but may be replicated across them. Replicas improve the fault-tolerance of the system, and provide an opportunity to select the closest replica to answer a historic subscription. However, having to maintain consistency among the replicas increases the complexity of the system.

5.2 Database Provisioning

Subscriptions for future publications are routed and handled as usual by the pub/sub broker overlay network [2, 5, 12]. To support historic subscriptions, databases are attached to a subset of brokers as shown in Fig. 1. Each database is managed by a database administrator. Administration consists of assigning publication space partitions to one or more databases. A partition may be assigned to multiple databases to achieve replication, and multiple partitions may be assigned to the same database if database consolidation is desired. Partition assignments can be modified at any time, and any replicas will synchronize among themselves. The only constraint is that each partition be assigned to at least one database so no publications are lost.

We first outline how partitions are assigned to databases, and then explain the consistency model of database replicas

⁷The *null* predicate value means the attribute is not present in a matching publication.

followed by techniques used to maintain synchronization of replicas according to the consistency model.

Partition Assignment: Each database subscribes to `DB_CONTROL` publications addressed to it, and the administrator assigns partitions to databases by sending publications with `STORE` commands to the appropriate database. For example, the following publications assigns a partition to database `DB_A`.

```
INSERT (class, command, db, partition_spec)
VALUES (DB_CONTROL, STORE, DB_A,
        'SELECT * WHERE shipID = *, level < 10')
```

The partition specification is itself a subscription with the selection formula expressed in the `WHERE` clause. A database that receives the `STORE` command will extract the partition specification and issue it as an ordinary future subscription. Matching publications will then be delivered to the database which will store them. A database assigned a partition also issues an advertisement that defines its partition.

When the first broker receives a historic subscription issued by a subscriber, it assigns it a unique query identifier and then routes the subscription as usual towards publishers whose advertisements intersect the subscription. This ensures that the subscription will arrive at databases whose partitions intersect the subscription. The database(s) convert the subscription into a SQL query, retrieve matching publications from the database, and publish the results. These “historic” publications are annotated with the subscription’s unique query identifier so they are only delivered to the requesting subscriber. After the result set has been published, the database will issue an `END` publication, which is used to *unsubscribe* the historic subscription.

The interaction with the databases fully leverages the content-based pub/sub model, and the databases are never addressed directly. In fact, it is impossible for publishers to discover where their publications are being stored, or for subscribers to know which databases process their queries. This simplifies management since databases can be moved, added or removed, and partitions reassigned at will.

Partition Replication: To improve availability, fault-tolerance and query performance, a partition may be replicated. Partition assignment strategies include partitioning, partial replication and full replication.

With partitioning, a database may be assigned several partitions, but each partition is assigned to only one database. That is, there is only one replica per partition. With partial replication, a given partition may be replicated by assigning it to multiple databases. The choice and location of replicas is studied in Sec. 6. With full replication, every database maintains replicas of all partitions. That is, each database stores all publications.

The various strategies have tradeoffs and are appropriate under different circumstances. The partitioning policy is simple and avoids replica consistency issues, but is sensitive to failures. Partial replication can tolerate failures of all but one replica, but requires logic to ensure the historic subscription is answered by only one of the replicas. Full replication is even more robust, and historic subscriptions can always be answered fully by the nearest database, minimizing network traffic. However, the high degree of replication imposes greater overall traffic and storage costs, as well as larger synchronization overhead. The partition assignment policies allow an administrator to tradeoff storage space, routing complexity, query delay, network traffic, parallelism of queries, and robustness.

Consistency Model: In standard pub/sub systems, two subscribers with identical subscriptions may receive notifications in different order. While the PADRES system will preserve the order of publications from any particular publisher, it cannot guarantee that subscribers observe the same interleaving of publications from multiple sources. Due to network delays, subscribers will typically receive publications from nearby sources sooner than those from distant ones. This is a generally accepted semantic in distributed pub/sub systems and many applications work with this assumption. Imposing a global ordering of publications will require centralized or quorum-based techniques that diminish the scalable distributed architecture.

In this paper, we continue to provide this traditional pub/sub ordering semantic for both future and historic subscriptions. To clarify the discussion, we define various levels of consistency for subscription results.

Definition 2. Eventual Consistency – There exist time periods t_1 and t_2 such that if two subscribers issue identical subscriptions at least t_1 time before any publisher, then after a period of t_2 after all publishers have stopped publishing, the publications delivered to the two subscribers are identical.

Notice that the eventual consistency definition does not specify anything about the order in which publications are delivered. As stated earlier, it is possible in distributed pub/sub systems to ensure per-source ordering of publications to interested subscribers. Let $p' \prec p$ mean that the same publisher issued publications p' and p and that it published p' before p . Then, all subscribers that receive both p' and p will receive p' first. However, subscribers may see a different order of interleaved publications from multiple sources. We define weak consistency to describe this generally accepted pub/sub semantic.

Definition 3. Weak Consistency – Consider any pair of subscribers s_1 and s_2 that have issued identical subscriptions (at any time) and have received matching publication sets P_1 and P_2 , respectively. For any pair of publications p' and p that appear in both P_1 and P_2 , where $p' \prec p$, both subscribers received p' before p . In addition, if there exists a p'' that satisfies $p' \prec p'' \prec p$, then p'' must belong to both P_1 and P_2 .

Informally, weak consistency states that all interested subscribers receive publications from a single source in the order the publications were published, and there are no “gaps” in the per-source publication stream. Note that the subscribers may receive different sets of publications because they may issue their subscriptions at different times.

Since subscribers do not know when data sources have stopped publishing, in practice, they can only rely on weak consistency semantics as opposed to eventual consistency.

In our system, the databases act as ordinary subscribers so database replicas can rely on, and hence provide, weak consistency semantics for historic subscriptions. Note that the publications sinked into a database replica are inserted in an append-only manner, and there is only one “writer” per advertisement, so we avoid write conflicts even when a database receives publications from multiple publishers.

Partition Replica Management: Our system allows an administrator to add and remove replicas of a partition at any time, as well as PAUSE and RESUME a database. Consequently, database replicas may issue their subscriptions and

sink publications during different periods of time. However, we require that a replica be able to deliver results for historic queries for its assigned partition, and the results must include all publications ever published since the system became operational (modulo the weak consistency allowance among replicas), including those publications issued before the replica was provisioned. To capture this point, we define a stronger consistency requirement for database replicas.

Definition 4. Replica Consistency – The publications stored in any pair of replicas r_1 and r_2 of a partition R must (i) be weakly consistent and (ii) if a publication p appears in both r_1 and r_2 , then for every publication $p' \prec p$, p' must appear in both r_1 and r_2 .

Informally, replica consistency adds the requirement that replicas contain a complete history of publications before some point in time.

Since partition replicas may be added and removed at any time, they need to synchronize among themselves to maintain the replica consistency. We assume that every partition is always assigned to at least one replica so that every publication is available in at least one database.

Databases transition among three states for each of the partitions assigned to them: **synchronizing**, **active**, and **paused**. A database can only answer historic subscriptions for a partition when it reaches the **active** state. When a new partition is assigned to a database, it begins in the **synchronizing** state and attempts to transition to the **active** state. A transition from **synchronizing** or **paused** states to the **active** state invokes a synchronization protocol so that the partition maintains replica consistency with its replicas.

Synchronization consists of four steps. First, the database issues the partition specification as a subscription as described earlier. *Ack* messages are collected from the leaves of the subscription propagation tree and are hierarchically propagated back to the database. The acknowledgments also include a count of potential data sources N_S for the partition. The receipt of the cumulative acknowledgment confirms that the subscription has been propagated throughout the overlay and any future publications will (eventually) be delivered to the database. Any publications P_{new} received by the database before the protocol is complete is buffered. In addition, upon receiving a partition specification subscription, a broker with a publisher p connected to it that has submitted an advertisement that intersects the subscription now issues a FLUSH publication to all database replicas whose partitions intersect the advertisement. The delivery of this publication at an active database replica confirms that all publications issued by publisher p before FLUSH have been delivered to the database. Third, the new database replica requests all the publications P_{old} stored at an active replica.⁸ The existing active replica will respond to the request only after receiving N_S corresponding FLUSH publications. (The value of N_S was included as part of the request from the new replica.) Finally, the new replica will interleave publications P_{old} and P_{new} so they respect weak consistency ordering (i.e., per-source ordering), filter out duplicate publications, and insert them into the database. Publishers tag

⁸As an optimization, a paused replica may send, along with the request, the most recent publication p_s it has from each source s to indicate that it only needs publications p'_s published after p_s : $p_s \prec p'_s$. The assumption is that the replica was replica consistent before it was paused.

their publications with an increasing sequence number inserted to facilitate this ordering. At this point the newly provisioned (or resumed) partition is replica consistent with any other active replica, and it transfers to the active state and begins to answer historic queries. The correctness of this protocol is established by the property below.

PROPERTY 1. *The synchronization protocol guarantees replica consistency.*⁹

5.3 Dynamic partitions

As the subscription and publication workload changes, it may be worthwhile to repartition the publication space. Once a new partitioning scheme has been computed, the administration must assign these new partitions to either existing or new databases. The databases will synchronize among themselves as outlined above, so that the new partitions retrieve their relevant data from one or more old existing partitions. Once all the new partitions are active, the administration may shut down the old partitions.

During this transition period, there may be times where both old and new partitions are running in the system. In particular there may be partitions with overlapping advertisements. Consider Fig. 3 where a subscription S intersects two

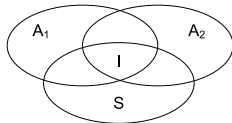


Figure 3: Overlaps

overlapping partitions with advertisements A_1 and A_2 . Assume $I = P(S) \cap P(A_1) \cap P(A_2)$ is not empty. Now, publications not in I , must be retrieved from the appropriate database, but those in I may be retrieved from either. Anytime a subscription retrieves data from a partition with advertisement A_i that overlaps some other partitions, the database is given, in addition to S , an *exclusion set* of advertisements \bar{A}_i . Publications that belong to advertisements in this exclusion set are discarded from the result set returned by the database. Formally, if there exists an advertisement $A \in \bar{A}_i$ such that publication $p \in P(A_i)$ and $p \in P(A)$, then p is discarded. The next section describes how the system decides which publications to retrieve from which database partitions.

5.4 Partition Selection Cost Model

When partitions are replicated or overlap, the database from which to retrieve publications affects the subscription evaluation performance, including network traffic and database load distribution. This section presents functions that quantify the cost of retrieving data from various partitions, and heuristics to minimize these cost functions. Note that Sec. 4.2 models the cost of composite subscription correlation location, whereas this section quantifies the cost of a decomposition of an atomic subscription among partitions.

Given a set of partitions represented by advertisements $\{A_1, \dots, A_n\}$ that intersect a subscription, the problem is to decide what subset of $P(A_i)$ should be retrieved from partition i . Formally, we want a transformation function F that maps each A_i to an A'_i such that $\cup_i P(A_i) = \cup_i P(A'_i)$ and $P(A'_i) \cap P(A'_j) = \emptyset$ for all $i \neq j$. More specifically, the A'_i are computed using an exclusion set \bar{A}_i : $A_i \xrightarrow{F} A'_i = A_i - \bar{A}_i$.

⁹The proof is presented in a companion report but is not cited here due to the double-blind review.

We seek a function F that minimizes some metric. For example minimizing the network cost of retrieving data from the databases can be formalized as

$$\min_F \sum_i |P(S) \cap P(A'_i)| d(A_i),$$

where $d(A_i)$ is the network cost of retrieving a publication from database partition A_i . For this optimization metric, a greedy algorithm that always selects the nearest database partition results in an optimal selection of partitions. Evaluations in Sec. 6 demonstrate the significant savings this algorithm provides when partition replicas exist.

Another optimization is to minimize the number of databases queried. This also has the effect of reducing the exclusion set size $|\bar{A}_i|$. Large exclusion sets increase the complexity of processing the query at the database, since either the query must be rewritten so as not to retrieve publications in the exclusion set, or more joins need to be computed. This optimization can be expressed as

$$\min_F \sum_i NE(P(S) \cap P(A'_i)),$$

where function $NE(X)$ (not empty) evaluates to 1 if set $X \neq \emptyset$, and 0 otherwise. A greedy algorithm like the one used earlier would not optimize this metric. Instead, we use a heuristic in which the partition advertisements are sorted by the cardinality of the advertisement space that intersects the subscription: $|P(S) \cap P(A_i)|$. Then the exclusion set for each advertisement are simply those with lower rank: $A'_i = \{A_1, \dots, A_{i-1}\}$. Intuitively, this heuristic prioritizes partitions with large result sets.

To fully answer a historic subscription, data from every database partition queried must be retrieved and returned to the subscriber. Therefore, another strategy is to minimize the time needed to retrieve and transfer publications from the slowest database (i.e., the database with the largest result set). Formally,

$$\min_F (\max_i |P(S) \cap P(A'_i)|).$$

This optimization function is motivated by the results in Sec. 6 which show that significant impact on notification delay when a large part of a historic subscription is answered by a small number of databases. A heuristic to approximate an optimal solution to this metric works in a manner similar to the previous one where the advertisements are first ranked by their cardinality. However, this time we favour advertisements with smaller cardinalities. Therefore, the exclusion sets for A'_i are $\{A_{i+1}, \dots, A_n\}$.

Some metrics may conflict one another. For example, favouring the nearest database or attempting to minimize the slowest database may cause more databases to be queried. To consider these tradeoffs, a cost metric that minimizes the total subscription processing time considering all three factors above can be expressed as

$$\min_F (\max_i [C|P(S) \cap P(A_i)| |\bar{A}_i| + |P(S) \cap P(A'_i)| d(A_i)]).$$

Here C represents a constant for the average time to retrieve a historic publication from the database. The function also makes a worst case assumption on the exclusion set processing, namely that publications retrieved from the database are individually filtered by each advertisement in the exclusion set. Our approximate solution here is similar to the previous one. This time, we first rank the partition advertisements not simply based on the expected cardinality, but by the estimated time to query the database and transfer the publications to the subscriber. The estimated query time is the expression inside the max function in the above optimization function. Then, we compute the exclusion sets for

each advertisement as in the previous metric.

6. EVALUATION

This section evaluates an implementation of the subscription routing (Sec. 4) and partitioning (Sec. 5) algorithms in the PADRES pub/sub system. A 30 broker pub/sub overlay (and one database attached to each broker) with 10 publishers and 20 subscribers are deployed across a 20 node cluster of 1.86 GHz machines with 4 GB of RAM. This size of topology is representative of the border security scenario described earlier. Also, the publications used are derived from this scenario, and subscriptions generated with predicates following a Zipf distribution model locality among subscribers.

We investigate the following policies: centralized (all partitions in one database replica), partitioning (one replica per partition per database), partial replication (each database is still assigned at most one partition but 30%, 60% or 100% of the partitions have two replicas), full replication (every partition assigned to every database) and merged partial replication (60% of partitions have two replicas, and two partitions are assigned per database). A centralized policy in which a single database replica manages all partitions serves as a baseline. In all cases, the global publication space is divided into five partitions.

Micro-benchmarks: We first evaluate the performance of mapping between pub/sub messages and SQL statements. To store a publication in a database, the publication is converted into an SQL `INSERT` statement. Results, which have been omitted, show that the storage time increases linearly with the number of predicates. A historic subscription arriving at a database is converted into an SQL `SELECT` statement, and each query result is transformed into a publication message. Results show that this processing time increases roughly linearly with the number of subscription predicates and result set size.

Storage and Query Traffic: Publications must be routed to interested subscribers and to all corresponding database replicas. The latter cost depends on the path length between the publisher and database, and increases (sublinearly due to pub/sub multicasting) with the number of replicas. Fig. 4 shows that the number of overlay hops traversed by publications increases with the number of replicas. The partitioning scheme has no replicas and each publication is routed to exactly one database, whereas publications are broadcast to all databases under full replication.

Whereas a publication must be routed to all replicas of a single partition, a historic query is processed by a single replica of all relevant partitions. This affords opportunities for parallelism (if partitions are assigned to different databases) and path optimization (by choosing a closer replica). We use atomic historic subscriptions with two to five predicates each, each of which access between two and four partitions. In contrast with the storage traffic, Fig. 4 shows that total query result publication traffic decreases with more replicas. Compared to the partitioning strategy, partial replication of 30%, 60%, and 100% saves 7.9%, 36.1%, and 59.1% of the query traffic, respectively. Full replication performs best since all publications can be retrieved from the database at the subscriber’s host broker. In the centralized case, all storage and query traffic go to a single database, and neither metric can be improved by a different selection of replicas. The first optimization met-

ric from Sec. 5.4—which optimizes the subscription network traffic—was used in the above experiments, and the results indicate that nearby replicas are indeed used and decrease the query traffic.

There is a clear tradeoff between storage and query traffic when varying the number of replicas in Fig. 4. If network traffic is the primary concern, the choice of the replication policy will depend on the bias of the workload towards writes (publications) or reads (subscriptions).

Notification Delay: The replication policy not only affects query traffic, but also the time to deliver query results to subscribers. Fig. 5 shows the average delay of processing subscriptions with increasing result sets. The delay is computed as the period from when a subscriber issues a subscription to when it receives a matching notification, averaged over all notifications. Each subscription accesses an average of four partitions. It may be surprising that the full replication policy suffers one of the longer delays despite all the data being available at the closest database. We mentioned earlier that queries with larger result sets take longer to process. With full replication, the entire query is answered by a single database whereas the other policies process queries with smaller result sets in parallel at multiple databases. The centralized policy is even worse than full replication since it not only suffers from a large query result set but provides no choice in selecting a closer replica. Policies with fewer partitions per database perform better, indicating that the query processing time at the database dominates the time to route the results to the subscriber. Among the policies that assign one partition per database (partitioning and partial replication), Fig. 5 shows that increasing replicas decreases the delay. In this case, the query processing delay remains constant but more replicas provide opportunities to decrease the routing delays by selecting a nearby replica. However, the benefits of replication are small relative to those of parallelism, reinforcing the observation that query processing delays dominate routing delays.

Fig. 6 presents the delay results from Fig. 5 in more detail, now showing the period of time between the first and last notification at the subscriber. We see that even though full replication routes results faster than with partitioning (the time range between the first and last notification is smaller), the query processing time is worse (the first notification takes longer to deliver). For example, in the case of subscriptions with about 18 000 matches, full replication requires about 30 s to deliver the first notification compared to less than 0.5 s with partitioning. We also see that the first notification is delivered just as fast with partial replication as with partitioning since they both exploit parallelism, but the former delivers the final notification sooner since it benefits from database locality. These results support the assumption underlying the third metric in Sec. 5.4 which tries to minimize the number of results returned by the database that returns the most results, effectively balancing the query load across replicas and achieving greater parallelism.

The observations confirm that database processing delays and parallelism benefit more than locality and routing path lengths. They also suggest that if notification delay as opposed to network traffic is to be optimized, queries should not simply be processed by the nearest replica, but query result set size and parallelism should be considered.

Partitioning: Algorithm 1 produces partitions that are evenly accessed. Comparing this even partitioning with

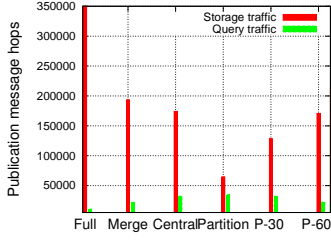


Figure 4: Traffic

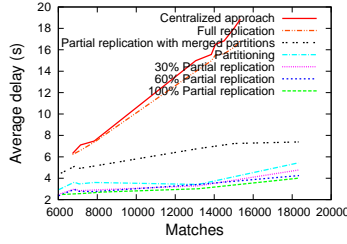


Figure 5: Notif. Delay

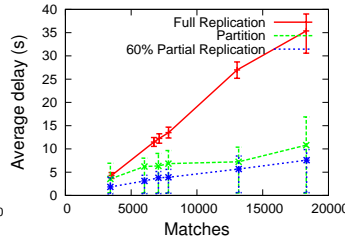


Figure 6: Delay Range

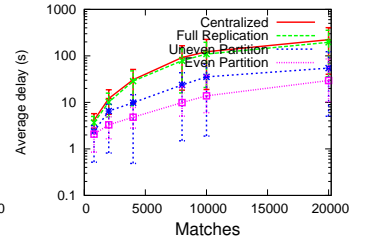


Figure 7: Partitioning

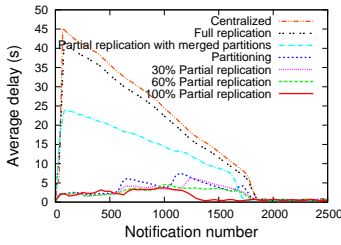


Figure 8: Hybrid Sub.

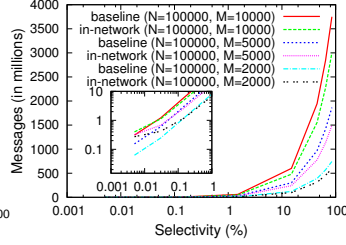


Figure 9: Hybrid CS

one which randomly divides the publication space, Fig. 7 shows that the average notification delay with even partitioning is 61% better. Also, since the number of publications from each partition under uneven partitioning vary widely, the lightly loaded partitions deliver notifications soon, but heavily loaded partitions are slow and ultimately cause the average delay to be worse. In the centralized approach and the full replication, where all publications are stored in one database, the processing time of a query dominates the notification delay. Full replication performs 12% better than the centralized approach because it decreases the routing delay by using the local database.

Hybrid Subscriptions: In Fig. 8 we examine the notification delay for a hybrid atomic subscription that queries the databases (the past) and two publishers (the future). The figure plots the average delay of the n^{th} notification. The relative delays of the historic notifications in the presence of active publishers conforms to the earlier results: delay increases with more partitions per database. It is interesting to observe the hills and valleys for some of the schemes in Fig. 8, such as the partitioning policy, where we can see bursts of notifications arriving from different databases.

We also evaluate a hybrid composite subscription that correlates historic and future publications. Without this capability, a subscriber must subscribe to the future publications, and for each publication it receives generate an SQL statement to query the database for correlated data. We refer to this as the *baseline* approach. With hybrid composite subscriptions, however, the correlation is done by the brokers in the network, and the subscriber is only notified of the correlation results. Fig. 9 compares the network traffic for both approaches, where N is the number of (historic) publications in the database and M is the number of (future) publications from publishers. The x-axis is the average selectivity of a publication over the historic data, and controls the correlation between historic and future data. The results show that when selectivity is high, that is, there is little correlation between historic and future publications,

the baseline approach incurs less traffic since the uncorrelated historic publications are not forwarded. The drawback with the baseline approach is that the subscriber must receive all future publications (even those not correlated with any historic publications), and issue SQL queries for each. With hybrid composite subscriptions, this work is done by the brokers in the network. When both N and M are large, and the correlation is high, Fig. 9 shows that the in-network approach saves up to 20% of the network traffic.

Composite Subscriptions: We evaluate the composite subscription routing protocols from Sec. 4: simple, topology-based, and adaptive routing. A composite subscription is issued that correlates data from eight sources: seven publishers publishing at rates from 50 to 500 msg/min, and one database. We measure the network traffic and notification delay as measured from the publication time of the last publication that contributes to the composite subscription match to the time when the subscriber is notified of the match. In both Figs. 10 and 11, adjacent brokers (those one unit away in the row and/or column axes) are neighbours in the overlay. Also, the composite subscriber is connected to the broker at (column,row) coordinate (10,8) and publishers are connected to the brokers at (*,1).

Fig. 10 plots

the outgoing traffic at each broker in the overlay. In simple routing, a composite subscription is split into atomic subscriptions at the broker

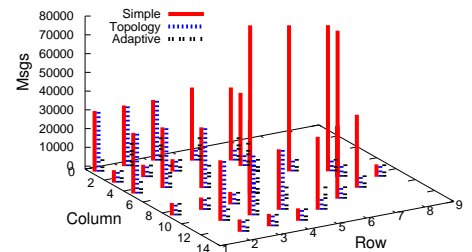


Figure 10: Composite Sub.

that first receives the composite subscription from a subscriber. In this case, broker (10,8) evaluates the composite subscription and filters out unmatched publications. Since filtering does not occur until the last broker along the paths from publishers to subscriber, each broker along the path from publishers to the subscriber has high outgoing traffic.

In topology-based routing, a composite subscription is forwarded as a whole until it reaches the broker where its atomic subscriptions must be forwarded in different directions in the topology. In our topology, the composite subscription is split at brokers (1,5) and (12,5), and we observe in Fig. 10 that filtering occurs at those brokers.

The adaptive routing algorithm determines the composite detection location based on potential publication traffic. In this scenario, publishers with high publication rates con-

nect to brokers (1,1) and (5,1), and hence it is desirable to detect composite subscriptions near them. The results in Fig 10 show that the adaptive algorithm reduces traffic by an average of about 66% and 43% compared to simple and topology-based routing, respectively. These savings are enjoyed by all brokers downstream of the join points. The average notification delay in topology-based routing is about 0.1 s, which the adaptive algorithm manages to reduce by about 48% by filtering out messages early in the network and hence reduce queuing and routing delays.

Dynamic

workload: We further evaluate adaptive routing in a scenario where conditions change. Publication rates are modified during the experiment: heavy publishers re-

duce their rates from 400 to 50 msg/min, and others increase their rates from 100 to 500 msg/min. Under the new workload, the original join point brokers initially chosen by the adaptive approach may no longer be optimal. Fig. 11 shows traffic with the changing workload when the join points remain at their optimal locations as determined during subscription routing (the adaptive case), and when they are allowed to react to the changing workload (the dynamic case). For comparison, the figure also replots from Fig. 10 the adaptive results under a static workload. In all cases, we only measure the traffic after the workload has changed.

When the join points do not move (adaptive cases), a change in the publication workload increases the overall traffic by about 220% since the join points are no longer optimal. However, by moving the join points to brokers (3,1) and (9,1), where more publications are being generated, the dynamic algorithm reduces the total network traffic by about 72% compared to the case when the join points remain fixed despite changing conditions. Again, the traffic reductions are enjoyed by all brokers downstream of the new join points.

7. CONCLUSIONS

The traditional pub/sub model does not support the retrieval of data published before a subscription is issued. However, the ability to access data from the past and the future in a unified manner is an important and useful requirement in many applications.

The proposed HSQL language provides an SQL-based interface to subscribe to historic and future publications. The language fully retains content-based pub/sub semantics and features, and can express filtering constraints, aggregation functions, projections, and correlations (joins) across any combination of future and historic data in a manner that preserves pub/sub decoupling and anonymity.

An architecture to evaluate HSQL subscriptions is presented and employs a set of databases to store publications and respond to historic subscriptions. Concepts from distributed databases are used but the pub/sub model differs sufficiently from the relational model to preclude a direct transfer of the ideas. The database architecture supports a

range of distribution and replication strategies with the ability to arbitrarily assign portions of the data space to one or more databases. An algorithm is presented to uniformly partition the data space, and a synchronization protocol is developed to ensure *replica consistency* among replicas.

To optimize the evaluation of HSQL queries, various subscription routing protocols are presented including a dynamic one that determines the least costly location in the overlay to compute composite subscription correlations.

Evaluations demonstrate the tradeoffs of several partition assignment and replication policies. Assigning all partitions to every database imposes the greatest publication storage traffic but cheapest historic subscription overhead, whereas a fully distributed partitioning results in the opposite tradeoff. The relative bias of writes (publications) and reads (subscriptions) in an application will determine the appropriate replication policy. In terms of delivering notifications, however, full replication takes longer than with complete partitioning because the latter is able to parallelize subscription processing across several databases, even though the databases are further away. That is, parallelization of subscription processing benefits much more than locality of databases. Furthermore, partitioning is able to start delivering notifications sooner than full replication, again due to parallelism. Adaptive subscription routing successfully determines efficient locations in the network to evaluate composite subscription correlations, and dynamic routing is able to react to changes in workloads and readjust the correlation evaluation locations, reducing network traffic even further.

8. REFERENCES

- [1] B. Chandramouli *et al.* On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD '06*.
- [2] A. Carzaniga *et al.* Forwarding in a content-based network. In *SIGCOMM*, 2003.
- [3] S. Chandrasekaran *et al.* Streaming queries over streaming data. In *VLDB '02*.
- [4] J. Chen *et al.* NiagaraCQ: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 2000.
- [5] G. Cugola *et al.* The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE TSE*, 2001.
- [6] T. Fawcett *et al.* Activity monitoring: Noticing interesting changes in behavior. In *SIGKDD*, 1999.
- [7] L. Fiege *et al.* Engineering event-based systems with scopes. In *ECOOOP*, 2002.
- [8] I. Koenig. Event processing as a core capability of your content distribution fabric. In *Gartner Event Processing Summit, Orlando, Florida*, 2007.
- [9] G. Li *et al.* Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, 2005.
- [10] S. Madden *et al.* Continuously adaptive continuous queries over streams. In *SIGMOD '02*.
- [11] T. Mungham *et al.* Integration of sensors for automated radiation detection. In *NATO SET-125*, 2008.
- [12] L. Opyrchal *et al.* Exploiting IP multicast in content-based publish-subscribe systems. In *Middleware*, 2000.
- [13] K. Ostrowski *et al.* Extensible web services architecture for notification in large-scale systems. In *IEEE ICWS*, 2006.
- [14] T. Ozsu and P. Valduriez. Principles of distributed database systems, 2nd edition, 1999.
- [15] I. Rose *et al.* Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds. In *NSDI*, 2007.

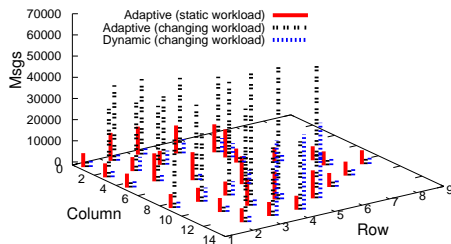


Figure 11: Dynamic workload

- [16] C. Schuler et al. Supporting reliable transactional business processes by publish/subscribe techniques. In *TES*, 2001.
- [17] D. Terry et al. Continuous queries over append-only databases. In *SIGMOD '92*.
- [18] Y. Tock et al. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, 2005.