End-to-End Support for Joins in Large-Scale Publish/Subscribe Systems*

Badrish Chandramouli Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{badrish,junyang}@cs.duke.edu

ABSTRACT

We address the problem of supporting a large number of select-join subscriptions for wide-area publish/subscribe. Subscriptions are joins over different tables, with varying interests expressed as range selection conditions over table attributes. Naive schemes, such as computing and sending join results from a server, are inefficient because they produce redundant data, and are unable to share dissemination costs across subscribers and events. We propose a novel, scalable scheme that group-processes and disseminates a general mix of multi-way select-join subscriptions. We also propose a simple and application-agnostic extension to *content-driven networks* (*CN*), which further improves sharing of dissemination costs. Experimental evaluations show that our schemes can generate orders of magnitude lower network traffic at very low processing cost. Our extension to CN can further reduce traffic by another order of magnitude, with almost no increase in notification latency.

1 Introduction

As computing continues to grow more ubiquitous and personal, there is an increasing need for customized, real-time data delivery. Millions of users want personalized information delivered to their desktops, phones, email and instant messaging clients, etc. A *publish/subscribe system* is a middleware for matching *events*, which are generated by data sources (*publishers*), to *subscriptions*, which specify the interests of users (*subscribers*). Traditional publish/subscribe systems only support *stateless* subscriptions, defined as filters over the contents of individual events. However, there is a pressing demand for efficient support of more complex subscriptions, such as those that correlate data across multiple sources. These subscriptions are *stateful*—given an incoming event, the system needs information beyond the content of this event itself in order to determine whether and how it affects these subscriptions. The relational join operator provides a convenient way to correlate data across sources, and *select-join* subscriptions are a common class of stateful subscriptions, as the following example illustrates.

Example 1. *Consider a publish/subscribe system for financial data. One source of data is basic stock information, represented by a table* `Stocks`$(\texttt{Symbol}, \texttt{PER}, \ldots)$*, where attribute* `PER` *records price-to-earning ratio, a popular measure of stock quality. Another source is analyst reports, represented by* `Reviews`$(\texttt{Symbol}, \texttt{Rating}, \ldots)$*. A stock may receive multiple ratings from different analysts.*

A subscriber may be interested in cases where a stock's rating and its `PER` *respectively belong to two prescribed ranges—for example, good ratings (no less than* 6 *on a scale of* 1 *to* 10*) for stocks with relatively high* `PER` *(between* 45 *and* 70*). This subscription, which we denote by* X_1*, can be expressed as a* select-join *query:*

$$\sigma_{\texttt{PER} \in [45,70]} \texttt{Stocks} \bowtie_{\texttt{Symbol}} \sigma_{\texttt{Rating} \in [6,10]} \texttt{Reviews}.$$

This subscription is stateful. Suppose an event comes with `PER` $=$ 60 *for a new stock. This event may or may not cause the subscriber to be notified, depending on whether the stock has any rating at or above* 6*. To determine whether notification is needed, and if yes, to compute the new join result tuples for notification, we must refer to the contents of* `Reviews`*, which are not part of the event itself.*

Challenges Continuing with Example 1, we illustrate the challenges (and opportunities) that arise in supporting select-join subscriptions for wide-area publish/subscribe. Suppose the current contents of `Stocks` and `Reviews` are as follows. For simplicity, let us first assume that each incoming `Stocks` event is an insertion into `Stocks` (i.e., `Stocks` tracks historical information).

`Stocks`:

	Symbol	PER	...
s_1	GOOG	51.7	...
s_2	YHOO	51.2	...
s_3	AMZN	92.8	...

`Reviews`:

	Symbol	Rating	...
r_1	GOOG	5.5	...
r_2	GOOG	6.0	...
r_3	GOOG	7.1	...
	...	...	...
r_{20}	GOOG	9.5	...
r_{21}	YHOO	7.5	...
r_{22}	AMZN	7.2	...
r_{23}	AMZN	7.8	...

Given a `Stocks` event, a naive approach is to compute, for each subscription, any change to the result of the select-join query, as in *incremental view maintenance* [13]. If this change is not empty, we say that the subscription is *affected* by this event, and we send the change to the subscriber in a notification message. For subscription X_1 in Example 1, computing the change requires joining the new `Stocks` tuple with `Reviews`. For example, on an event inserting a new GOOG tuple $s_4 = \langle \text{GOOG}, 52.1, \ldots \rangle$, we would send X_1 a message containing $s_4r_2, s_4r_3, \ldots, s_4r_{20}$, which is the subset of $\{s_4\} \bowtie$ `Reviews` satisfying the selection conditions of X_1.

The first obvious problem is that of *large server output size*. The server has to enumerate potentially many output tuples. This problem slows down processing, increases server load, and in turn aggravates notification latency. Upon closer examination, this problem is caused by three sources of redundancy:

- ***Result representation redundancy***. Within each notification message, the newly inserted tuple is repeated many times, once for each joining tuple. For example, the notification for X_1 when s_4 is inserted repeats s_4's content 19 times.

*Supported by National Science Foundation Grant IIS-0713498.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

- ***Current-content redundancy***. This redundancy arises when some information in the notification can be inferred from the current subscription content. In many application scenarios, it is reasonable to assume that a subscription client maintains the current subscription content. Thus, information that can be inferred from this content need not be included in the notification message. For example, because of s_1, X_1's subscription content should already contain $r_2, \ldots, r_{20}$ prior to the insertion of s_4; there is no need to send these joining `Reviews` tuples again. The naive approach does not take advantage of this option.
- ***Inter-event redundancy***: This redundancy arises when some information in the notification message cannot be recovered from the current subscription content, but nevertheless has been previously transmitted to the subscriber due to an earlier event. For example, suppose that `Stocks` maintains only a window of recent history. A sequence of new `Stocks` events for GOOG drop its `PER` to lower than 45, and the older GOOG tuples, s_1 and s_4, have been expired (i.e., deleted) from `Stocks`. At this point, the joining `Reviews` tuples $r_2, \ldots, r_{20}$ are no longer in X_1's content. However, a future event may raise GOOG's `PER` to above 45 again, and bring back these joining `Reviews` tuples. Ideally, we would like to avoid resending these tuples to X_1 if possible.

Another serious problem of the naive approach is the *lack of sharing of dissemination costs* across subscribers. A large-scale publish/subscribe system may have millions of subscribers, and many of them may be interested in similar join results. For example, another subscription X_2 may be interested in a `PER` range of $[50, 80]$ and a rating range of $[7, 10]$, which are different from but overlap with those of X_1. For insertion of s_4, changes to X_1 and X_2 are nearly identical—both contain $s_4r_3, \ldots, s_4r_{20}$, and the only difference is that X_2 should additionally get s_4r_2. When notifying multiple subscriptions, we wish to avoid ***inter-subscription redundancy***, which increases not only the overall communication cost, but also server stress and notification latency.

There is even more opportunity in improving efficiency by identifying and avoiding ***re-dissemination redundancy***, which is a generalization of current-content, inter-event, and inter-subscription redundancies. For example, suppose that a third subscription, X_3, is interested in the `PER` range of $[80, 100]$ and the same rating range as X_1. Thus, r_{22} and r_{23} (reviews for AMZN) are in X_3. Later on, if AMZN's `PER` becomes 55, X_1 should receive r_{22} and r_{23}. Suppose that X_1 and X_3 share some portion of their network dissemination paths; it would be nice if we could avoid resending the same contents through the shared path. This optimization enables cost sharing across both events and subscriptions.

The solution to these problems is challenging. 1) Compressing individual notification messages can avoid result representation redundancy, but is ineffective at removing other redundancies. 2) We can remove current-content redundancy at the server by checking the content of each outgoing notification against the current subscription content. However, the overhead is high, and it is difficult to scale to a large number of unique subscriptions. 3) Inter-subscription redundancy can be overcome by employing an efficient *dissemination network*, popular in wide-area publish/subscribe systems. This network consists of a set of nodes (sometimes called *brokers*) over the Internet, each responsible for a subset of the subscriptions. These nodes forward events to each other based on downstream subscription interests. Each event traverses down a tree of nodes spanning all affected subscribers. However, traditional publish/subscribe systems do not directly support stateful subscriptions such as joins, and their stateless nature makes it difficult to implement state- and history-based optimizations, such as avoiding current-content and inter-event redundancies. 4) No existing solutions or simple extensions (discussed in Section 2) are able to avoid all types of redundancies. Furthermore, these solutions do not mesh well with each other, so it is unclear how to combine them to simultaneously avoid redundancies of different types.

Design for Practicality Besides the challenges above, there are important practical considerations. In this paper, we make a conscious decision not to over-complicate the design of dissemination networks for publish/subscribe, e.g., by augmenting nodes in these networks with hard application state or application-specific processing logic. While complex, stateful networks may be appropriate in certain scenarios, we argue that they are best avoided in loosely-coupled wide-area systems, because adding such state and logic significantly increases system complexity, complicates failure and consistency issues, and creates deployment hurdles. Instead, we want to reuse common, off-the-shelf network substrates for dissemination because at large scales, such substrates are more robust and less likely to face deployment and maintenance problems.

Contributions We provide a complete solution to supporting a large number of select-join subscriptions in a publish/subscribe system. Our goals are to reduce and/or bound bandwidth consumption, notification delay, and server processing costs. Instead of designing a complex dissemination network, we base our solution on a simple, well-established type of dissemination substrates, which we termed *content-driven networks* (*CN*) [6]. We develop novel server-side processing algorithms and application-agnostic extensions to stateless CN to support stateful select-joins and to improve efficiency. More specifically, our contributions include:

- In Section 3.1, we propose a method to eliminate result representation and current-content redundancies by rewriting select-join subscriptions into select-semijoins. We demonstrate how these subscriptions can be efficiently processed at the server.
- In Section 3.2, we show how to further use *reformulation* to eliminate inter-subscription redundancy and enable the use of CN. The novel reformulation scheme is computationally efficient, and significantly reduces network traffic.
- In Section 5, we propose a simple and novel extension to CN, called *CN**, which further improves efficiency by reducing inter-event and, more generally, re-dissemination redundancies.
- Our solution is general. We show how to handle a generic mix of multi-way select-joins (Section 4). Our solution can also be extended to multi-attribute equijoin and range-selection conditions; see Appendix C for details.
- We have conducted comprehensive experimental evaluation of our solution and its competitors. We measure the performance of server-side processing and dissemination in a simulated wide-area network. Results show orders-of-magnitude improvement for several relevant metrics. We also include examinations of multi-way select-joins and the effectiveness of CN*.

Some of our contributions have applications beyond joins. Interestingly, even for simple selection subscriptions, if events contain bulky payloads (e.g., a PNG image attribute in `Stocks` containing the company logo, or a text attribute in `Review` carrying the analyst report), it would be more efficient to treat selections as select-joins and apply our schemes; see Appendix I for details.

CN*, the extension to CN, is also a contribution in its own right, as it can be applied to any type of subscription to improve efficiency by optimizing the transfer of payloads. Again, this extension has been designed with practicality in mind—it is application-agnostic, free of hard state, and *incrementally deployable*, i.e., CN* nodes (with the extension) can coexist with regular CN nodes in the same network, and bring commensurate improvement in efficiency.

2 Preliminaries

A publish/subscribe system is responsible for delivering data generated by *publishers* to interested *subscribers*, whose interests over data are defined as *subscriptions*. In large-scale, wide-area publish/subscribe system, the number of subscriptions is typically high, and subscribers can be located all over the network. Traditional publish/subscribe systems assume that events follow some schema, and subscriptions are filters over individual events (e.g., `PER` $\in [45, 70]$ for a `Stocks` event).

In this paper, however, we use a publish/subscribe model that supports more powerful database-style subscriptions. We assume that a central server maintains a database. Events from publishers arrive at the server and are treated as modifications (inserts, deletes, and updates) to the database. A subscription is a query Q over the database. Suppose an incoming event changes the database state from $\mathcal{D}$ to $\mathcal{D}'$. If $Q(\mathcal{D}') \neq Q(\mathcal{D})$, and we say that the subscription Q is *affected* by the event; the task of the system is to deliver a notification message to Q's subscriber, such that $Q(D')$ can be computed from $Q(D)$ and the content of this notification message. More details about our system architecture can be found in [7].

2.1 Dissemination Network

To enable efficient notification delivery, we employ a *dissemination network*, like most wide-area publish/subscribe systems (as discussed in Section 1). Each node in the network is responsible for a subset of the subscriptions, and the nodes collectively forward notification messages among themselves and to the subscribers. The last hop of notification delivery, from a node to an affected subscription assigned to it, is handled by a *message dispatcher* running at the node, which can use any delivery mechanism suitable to the subscription client, e.g., IP unicast, email, or instant message.

A popular technique for building the dissemination network is to use a ***content-driven network*** (**CN**), a term coined by [6] to refer to a class of overlay networks for data dissemination. CN has a clean and simple dissemination interface. Each message contains a list of attribute-value pairs. A subscription is a filter over individual messages, defined as a predicate involving message attributes. CN efficiently routes each message to all matching subscriptions registered in the network.

Many off-the-shelf overlay networks implement the CN interface. One example is *content-based network* (*CBN*) [4], which allows subscriptions to be arbitrary boolean predicates. Another example of CN is *Meghdoot* [14], based on a structured overlay called *content-addressable network* (*CAN*) [24]. In Meghdoot, a message contains d numeric attributes and each subscription is an orthogonal range predicate in the d-dimensional space. While this subscription language is more restrictive than CBN, it nevertheless conforms to the CN interface. More details on different examples of CN can be found in Appendix A. In this paper, we simply treat CN as a black-box delivery mechanism.

While the simplicity of CN's network interface enables efficient and scalable implementations, subscriptions directly supported by CN are limited to selections over event tables. Thus, CN cannot be directly used for stateful subscriptions such as select-joins, whose processing requires information beyond individual messages.

2.2 Select-Join Subscriptions

In this paper, we consider subscriptions expressed as orthogonal range selections over natural joins (*select-joins* for short). These subscriptions constitute a significant portion of workloads in practice. The database schema can be modeled as an undirected *join graph*, which may contain cycles. Nodes in this graph represent tables, and edges represent possible joins between tables. Each select-join subscription corresponds to a connected subgraph of the join graph, which we refer to as the *join signature* of the subscription. In addition to the join conditions implied by the join signature, each subscription can specify a local selection for each table, in the form of an orthogonal range condition. In general, such a condition can involve multiple attributes, constrained by different ranges. In this paper, we assume that subscriptions return all attributes in the result (i.e., no projections), and there are no self-joins.

We will start with the scenario, formalized in the example below, where all subscriptions have the same binary join signature but possibly different single-attribute range selections for each table. In Section 4, we show how to extend our solution to a general mix of multi-way joins. The extension to multi-attribute join and selection conditions is discussed in Appendix C.

Example 2 (Binary Select-Joins)**.** *Let $\mathcal{X} = \{X_1, X_2, \ldots\}$ be a set of subscriptions over tables $R(A, B, P_R)$ and $S(B, C, P_S)$. B is the join attribute, A and C are local selection attributes, and P_R and P_S represent all remaining attributes in the respective tables, which we call the* payload *attributes. Each X_i is defined by query*

$$\left(\sigma_{A \in I_i^A} R\right) \bowtie_B \left(\sigma_{C \in I_i^C} S\right),$$

where $I_i^A = [a_i^{\text{low}}, a_i^{\text{high}}]$ and $I_i^C = [c_i^{\text{low}}, c_i^{\text{high}}]$ specify the ranges of X_i's local selection conditions on $R.A$ and $S.C$, respectively. Note that Example 1 fits this scenario.

2.3 Strawman Solutions

Select-join subscriptions cannot be handled directly by CN. We now present several strawman solutions: The first one does not use CN, while the others leverage CN in straightforward ways.

Enumeration of Unicast Notifications (Enum-J) Given an incoming event, one option is to process all subscriptions at the server, compute a notification message for each affected subscription, and unicast this message to the corresponding subscriber. A notification message contains the relational difference between new and old subscription contents, which, in the case of an insertion event, is the result of the select-join evaluated over the inserted tuple and the joining tables. Many processing techniques have been developed, e.g., NiagaraCQ [10] in the context of continuous query systems.

This scheme corresponds to the naive approach introduced in Section 1. Enum-J suffers from large server output size and lack of sharing of dissemination costs. These problems make the approach difficult to scale to a large number of subscriptions.

Relaxation into Single-Table Selections for CN (Rel-Sel) An n-way select-join subscription can be "relaxed" into n selection subscriptions, one for each joining table. In Example 2, X_i can be relaxed into two subscriptions, $\sigma_{A \in I_i^A} R$ and $\sigma_{C \in I_i^C} S$. All resulting selection subscriptions can then be directly handled by CN (Section 2.1). R and S events are directly injected into CN, which then routes each event to all matching selection subscriptions. The subscription client maintains the contents of selection subscriptions, and joins them to compute the original subscription.

Rel-Sel avoids result representation redundancy, and its use of CN alleviates inter-subscription redundancy. Rel-Sel may seem to avoid re-dissemination redundancy as well, but it is only because Rel-Sel may transmit far more data than necessary. Client for X_i in fact receives enough data to be able to maintain the cross product $\sigma_{A \in I_i^A} R \times \sigma_{C \in I_i^C} S$, even though only the join is necessary. Losing the filtering power of join conditions can result in much higher traffic due to transmission of unnecessary content to subscribers.

Reformulation as Selections over Join for CN (Ref-J/Ref-J$^+$) *Reformulation* [5] is a method for supporting stateful subscriptions

over the stateless CN dissemination interface. On a high level, reformulation works as follows. Instead of injecting an event directly into CN, the server reformulates it into one or more messages with attribute-value pairs carrying additional information computed by the server. On the other hand, each subscription is reformulated as a filter over the reformulated messages. CN automatically routes reformulated messages to matching filters (reformulated subscriptions). The reformulated messages, computed by the server with full access to the database and subscription definitions, carry the state necessary for processing the otherwise stateful subscriptions.

We can use a simple reformulation scheme, which we call Ref-J, to support all select-joins with the same join signature. Given an incoming event, the server computes the change to the result of the natural join corresponding to the join signature (with no selections). This change is represented by a set of join result tuples. In the case of an insertion event, for example, these are the result of joining the inserted tuple with the other tables. Then, each join result tuple is injected into CN as a message. Each select-join subscription, on the other hand, is reformulated into a filter over join result messages according to the subscription's local selection conditions. For instance, in Example 2, the reformulated messages have the format $\langle \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{P_R}, \mathtt{P_S} \rangle$, while each subscription X_i is simply reformulated into the filter $\mathtt{A} \in I_i^A \wedge \mathtt{C} \in I_i^C$.

Again, CN helps us avoid inter-subscription redundancy. However, result representation redundancy still remains; the content of the incoming event is repeated in every reformulated message that the event generates. Also, Ref-J does not avoid current-content, or more generally, inter-event and re-dissemination redundancies.

Another inefficiency with Ref-J, caused by delayed application of selections (in CN instead of at the server), is that Ref-J may send join result tuples that are not interesting to any subscriptions. This problem can be solved by checking each outgoing join result tuple to ensure that at least one subscription would be interested in it. However, this extension, which we call Ref-J$^+$, increases server processing cost, and still inherits all other problems of Ref-J.

3 Binary Select-Joins

We now present our solutions for the scenario described in Example 2, where all subscriptions are binary select-joins between R and S. We will show how to generalize our techniques to multi-way joins in Section 4. As a first step, in Section 3.1, we ignore the dissemination aspect, and focus on how to efficiently compute, at the server, the minimal information to send to each subscription for every event. Next, in Section 3.2, we show how to retool the solution in Section 3.1 to further leverage CN for efficient dissemination.

3.1 Enum-SJ: Towards a Semijoin Approach

We saw in Section 1 that two important sources of inefficiency are result representation and current-content redundancies. It turns out that both can be eliminated by decomposing a select-join subscription $X_i = \sigma_{A \in I_i^A} R \bowtie_B \sigma_{C \in I_i^C} S$ into two select-semijoins: $X_i^R = \sigma_{A \in I_i^A} R \ltimes_B \sigma_{C \in I_i^C} S$, and $X_i^S = \sigma_{C \in I_i^C} S \ltimes_B \sigma_{A \in I_i^A} R$. We call them ***R-semijoin*** and ***S-semijoin*** respectively. Semijoins are a well-known method in distributed databases [3] for reducing communication when joining across different machines. A semijoin $R \ltimes S$ can be regarded as a generalized filter on R, which returns only those R tuples that join with at least one S tuple.

The insertion of a tuple $t_R = \langle a, b, p_r \rangle$ into table R can affect both R-semijoin and S-semijoin subscriptions. First, the server has to send t_R to every R-semijoin X_i^R where $a \in I_i^A$ and there exists at least one joining S tuple $\langle b, c, p_s \rangle$ such that $c \in I_i^B$. We call these subscriptions t_R***-affected***, or simply (when clear from the context), ***affected*** R***-semijoins***. Second, for each S tuple $t_S = \langle b, c, p_s \rangle$ that joins with t_R, the server has to send t_S to every S-semijoin X_i^S where $a \in I_i^A$, $c \in I_i^C$, and X_i^S does not already contain t_S (i.e., before t_R is inserted into R, $t_S \notin \sigma_{C \in I_i^C} S \ltimes_B \sigma_{A \in I_i^A} R$). We call these subscriptions ***(***t_R***-)affected*** S***-semijoins***.

The client for subscription X_i maintains the contents of both X_i^R and X_i^S. Upon receiving notifications to X_i^R or X_i^S, it updates the result of $X_i^R \bowtie X_i^S$, which is equivalent to the original X_i.

As a concrete example, consider the subscription X_1 in Example 1 and the tables showing the current contents of `Stocks` and `Reviews` in Section 1. Upon the insertion of a new `Stocks` tuple $s_4 = \langle \text{GOOG}, 52.1, \ldots \rangle$, recall that the naive approach (Enum-J) needs to send $s_4 r_2, s_4 r_3, \ldots, s_4 r_{20}$ to X_1. With the decomposition, however, we only need to send s_4 to $X_1^{\texttt{Stocks}}$. Note how semijoin avoids the multiplicity caused by join, thereby eliminating result representation redundancy; only one copy of s_4 is sent no matter how many joining `Reviews` tuples there are. Furthermore, note that nothing needs to be sent to $X_1^{\texttt{Reviews}}$, because the `Reviews` tuples that join with s_4 and satisfy X_1's selection condition on rating, namely $r_2, \ldots, r_{20}$, are already in $X_1^{\texttt{Reviews}}$. Hence, current-content redundancy is also avoided. This example highlights the fact that having an affected R-semijoin does not imply the corresponding S-semijoin is affected as well.

It is not trivial to compute the changes to a large number of select-semijoin subscriptions, since every subscription has different selection conditions. It is especially tricky to decide which S-semijoins need to receive a joining S tuple, because the decision involves testing whether they already contain the tuple. The remainder of this section is devoted to the details of scalable maintenance of R-semijoins and S-semijoins on an insertion into table R. Insertion into S is symmetric; updates and deletions involve straightforward extensions, which we omit for brevity.

Once the changes are computed, we assume (for now) that the server unicasts them to each subscription. We call this scheme Enum-SJ. We will see how to use CN to avoid inter-subscription redundancy and further reduce output size in Section 3.2.

3.1.1 Computing Changes to R-Semijoins

On an insertion $t_R = \langle a, b, p_r \rangle$ into R, we need to identify all affected R-semijoins. Several techniques exist in the continuous query processing literature, e.g. NiagaraCQ [10]. For example, we can first find all subscriptions whose local selection conditions on $R.A$ are satisfied by t_R, which can be done efficiently with an interval tree indexing all I_i^A intervals. For each such subscription X_i, we then probe a composite-key B-tree index on $S(B, C)$ to determine whether there exists at least one S tuple with $B = b$ and $C \in I_i^C$; if yes, the subscription is affected. The problem with this approach is that its running time is linear in the number of subscriptions whose selection conditions on $R.A$ are satisfied, which can be much higher than the actual number of affected subscriptions.

Processing with Stabbing-Set Index (SSI) We base our solution on an algorithm from [1], which is especially suited to our setting because it exploits the clustering among local selection ranges for efficient processing. In publish/subscribe systems, we expect a fair degree of clustering among subscription ranges because users often share similar interests.

The algorithm is based on a *stabbing-set index* (*SSI*), which partitions the set of subscriptions $\mathcal{X}$ into τ *clusters* $\mathcal{X}^{(1)}, \ldots, \mathcal{X}^{(\tau)}$ according to their I_i^C's, i.e., their local selection ranges on $S.C$. All subscriptions in the k-th cluster $\mathcal{X}^{(k)}$ must have their I_i^C's stabbed by a common point $p^{(k)}$, called the *cluster anchor*. With clustering of user interests, we should be able to partition $\mathcal{X}$ into a relatively smaller number of clusters (i.e., $\tau \ll |\mathcal{X}|$). Efficient algorithms

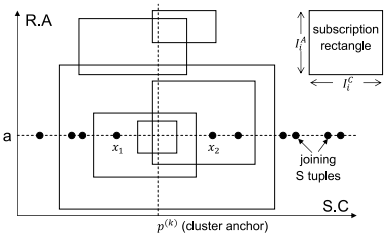

Figure 1: Processing Enum-SJ using SSI for cluster $\mathcal{X}^{(k)}$.

have been developed in [1] to maintain the partitions such that τ is close to the minimum possible. Figure 1 illustrates the set of subscriptions in cluster $\mathcal{X}^{(k)}$ in the space $S.C \times R.A$, where each subscription X_i is represented as a rectangle $I_i^C \times I_i^A$. The rectangles in each cluster are indexed by a 2-d R-tree.

On the insertion of $t_R = \langle a, b, p_r \rangle$ into R, we perform the following steps for each cluster $\mathcal{X}^{(k)}$. First, by looking up $(b, p^{(k)})$ in the B-tree index on $S(B, C)$, we can find the two joining S tuples with $S.C$ values (say x_1 and x_2) that are the closest possible to $p^{(k)}$ on each side of $p^{(k)}$. Figure 1 illustrates this step. Next, we probe the R-tree index for $\mathcal{X}^{(k)}$ to find those subscriptions in $\mathcal{X}^{(k)}$ that contain at least one of the points (x_1, a) and (x_2, a).

For each cluster, this procedure returns exactly all affected R-semijoins within the cluster. To see why, note that we can represent tuples in $\{t_R\} \bowtie S$ as points in the $S.C \times R.A$ space, and all of them must lie on the horizontal line $R.A = a$. An R-semijoin is affected iff its rectangle contains at least one of these points. Therefore, any affected subscription in $\mathcal{X}^{(k)}$ must necessarily contain $(p^{(k)}, a)$; for it to contain a tuple in $\{t_R\} \bowtie S$, it must further contain at least one of (x_1, a) and (x_2, a).

Complexity The space required for auxiliary data structures is $O(|\mathcal{X}| + |S|)$, i.e., linear in the number of subscriptions and the size of the database. Let k_R be the number of affected R-semijoins. Let $g = O(\sqrt{\max_{1 \le k \le \tau} |\mathcal{X}^{(k)}|})$ denote the cost of a lookup in an R-tree indexing all subscriptions in a cluster (excluding the cost component that is linear in the output size of the lookup). The running time is $O(\tau(\log |S| + g) + k_R)$. Let h_R be the size of one R tuple. The output size, as measured by the total size of all notification messages, is $O(k_R h_R)$.

3.1.2 Computing Changes to S-Semijoins

Given an insertion $t_R = \langle a, b, p_r \rangle$ into R, we need to send a joining S tuple t_S to an S-semijoin X_i^S iff t_S is ***exposed*** to X_i^S, i.e., t_S should be in X_i^S after the insertion but currently is not. It may seem that we need to examine the content of the S-semijoin to determine whether t_S is exposed to it. Interestingly, we show that exposure testing is not as hard as it seems—a single index lookup suffices.

Theorem 1 (Exposure Test)**.** *Given an insertion t_R into R, let*

- $a^-(t_R) = \max\{t.A \mid t \in R \wedge t.B = t_R.B \wedge t.A \le t_R.A\}$, *or $-\infty$ if the input to* max *is empty; and*
- $a^+(t_R) = \min\{t.A \mid t \in R \wedge t.B = t_R.B \wedge t.A \ge t_R.A\}$, *or $+\infty$ if the input to* min *is empty.*

In both definitions, R refers to the state of R before the insertion. Consider any $t_S \in S$ that joins with t_R (i.e., $t_S.B = t_R.B$): t_S is exposed to X_i^S iff $t_S.C \in I_i^C$ and $t_R.A \in I_i^A \subset [a^-(t_R), a^+(t_R)]$.

Proof. (Sketch) If $t_R.A \notin I_i^A$, X_i^S is clearly not affected by the insertion. Let $t_R.A \in I_i^A$. There are two cases: 1) If $I_i^A \not\subset [a^-(t_R), a^+(t_R)]$, it must contain one of $a^-(t_R)$ and $a^+(t_R)$. Either way, there already exists $t'_R \in R$ with $t'_R.A \in I^A$ and $t'_R.B = t_R.B$, such that any S tuple that should be in X_i^S due to insertion of t_R must already be in X_i^S because of t'_R. 2) If $I_i^A \subset [a^-(t_R), a^+(t_R)]$, there is no other $t'_R \in R$ with $t'_R.A \in I^A$ and $t'_R.B = t_R.B$. Therefore, any S tuple that should be in X_i^S due to insertion of t_R must not be in X_i^S currently. □

Applying Theorem 1, we can compute changes to S-semijoins as follows. First, we compute $a^-(t_R)$ and $a^+(t_R)$, which can be done efficiently by looking up (b, a) in the B-tree index on $R(B, A)$. Next, we process each subscription cluster $\mathcal{X}^k$ in turn, in the same loop where we compute affected R-semijoins (Section 3.1.1). For each affected R-semijoin X_i^R, if $I_i^A \subset [a^-(t_R), a^+(t_R)]$, the corresponding S-semijoin X_i^S is also affected, and its change includes the set of S tuples with $S.B = b$ and $S.C \in I_i^C$, which can be easily found in a B-tree index on $S(B, C)$. As an optimization, we do not need to repeat this lookup from the B-tree root for every affected S-semijoin. Instead, we use a single lookup of $(b, p^{(k)})$ for each cluster; to find S tuples with $S.B = b$ and $S.C \in I_i^C$ for X_i in this cluster, we simply traverse the B-tree leaves towards left and right starting from this point.

Complexity The total space requirement (including auxiliary data structures in Section 3.1.1) is $O(|\mathcal{X}| + |R| + |S|)$, i.e., linear in the number of subscriptions and the size of the database. Let s be the number of S tuples that join with t_R. Recall that k_R denotes the number of affected R-semijoins. Let k_S ($\le k_R$) denote the number of ***(t_R-)affected S-semijoins*** (i.e., those with at least one exposed joining S tuple), and let $\bar{s}'$ ($\le s$) be the average number of joining S tuples exposed to each affected S-semijoin. The running time, combined with the algorithm in Section 3.1.1, is $O(\log |R| + \tau(\log |S| + g) + k_R + k_S \bar{s}')$. The output size, as measured by the total size of all notification messages for S-semijoins, is $O(k_S \bar{s}' h_S)$, where h_S is the size of one S tuple.

3.2 Ref-SJ: Scalable Dissemination

Enum-SJ suffers from inter-subscription redundancy because it unicasts notifications to each subscription. In this section, we present Ref-SJ, which uses novel reformulation techniques to leverage CN for sharing dissemination costs. The overall approach is to have the server reformulate each incoming event into CN messages carrying additional information computed by the server, such that the otherwise stateful subscriptions can be reformulated accordingly into stateless filters supported by CN. Assume as before that we insert an R tuple $t_R = \langle a, b, p_r \rangle$.

3.2.1 Reformulating R-Semijoins

We need to send t_R to the set of affected R-semijoins efficiently. Can we characterize this set succinctly without enumerating the set membership? The answer is yes. To illustrate, let us map each R-semijoin X_i^R with $I_i^C = [c_i^{\text{low}}, c_i^{\text{high}}]$ to a point $(c_i^{\text{low}}, c_i^{\text{high}})$ in a 2-d space, as shown in Figure 2 (a).

Consider just the subset of R-semijoins whose I_i^A ranges contain a; we call these the ***(t_R-)selected*** R-semijoins. Obviously, every affected R-semijoin must first be selected. Among the selected R-semijoins, it turns out that the affected ones can be separated from the unaffected ones by a skyline, as stated by Theorem 2 below.

Definition 1. *A* skyline (viewing from northwest) *in* 2*-d is specified by a set of* skyline points $\{(x_1, y_1), (x_2, y_2), \ldots\}$ *with the property that no point lies to the northwest of another point; i.e., there exist no i, j such that $x_i \le x_j$ and $y_i \ge y_j$. The* subspace covered *by this skyline is the union of northwest quadrants of all skyline points, i.e.,* $\{(x, y) \mid \exists i : x \le x_i \wedge y \ge y_i\}$.

Theorem 2 (Descriptive Skyline)**.** *There exists a skyline (viewing from northwest) L such that 1) every t_R-selected R-semijoin in the subspace covered by L is t_R-affected, and 2) every R-semijoin not*

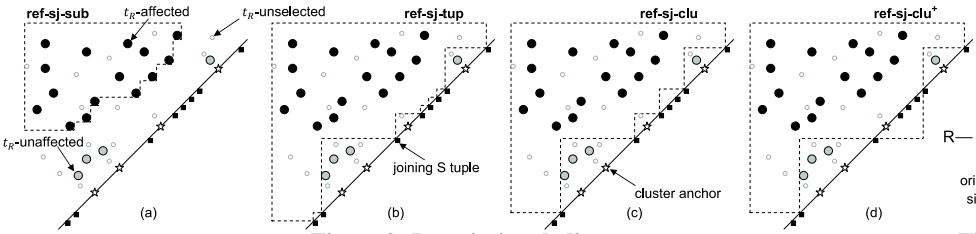

Figure 2: Descriptive skylines.

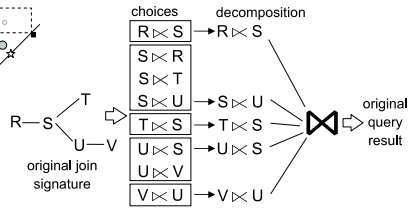

Figure 3: Multi-way decomposition.

in this subspace is not t_R-affected. We call a skyline satisfying these properties a descriptive skyline*.*

Proof. (Sketch) Let P be the (duplicate-free) set of points corresponding to all affected R-semijoins. Let P' be a subset of P, such that $p \in P'$ iff p does not lie to the northwest of any other point in P. The skyline specified by P' satisfies both properties. 1) If a selected R-semijoin X_i^R is covered by the skyline, it must lie to the northwest of some affected R-semijoin X_j^R, which implies that $I_i^C \supseteq I_j^C$. Therefore, any joining S tuple that satisfies X_j^R's selection condition must satisfy X_i^R's too. Hence, X_i^R must also be affected. 2) This property follows from the construction of P'. □

For example, Figure 2 (a) shows one such skyline with 7 skyline points (as constructed by the proof of Theorem 2). Using Theorem 2, our approach is to first compute a descriptive skyline at the server, given $t_R = \langle a, b, p_R \rangle$. Suppose this skyline has k_L points $(x_1, y_1), \ldots, (x_{k_L}, y_{k_L})$. We reformulate t_R into the message:

$$\langle \mathtt{A}{:}a, \mathtt{B}{:}b, \mathtt{P_R}{:}p_r, \mathtt{X}_1{:}x_1, \mathtt{Y}_1{:}y_1, \ldots, \mathtt{X}_{k_L}{:}x_{k_L}, \mathtt{Y}_{k_L}{:}y_{k_L} \rangle.$$

Accordingly, an R-semijoin is reformulated as a filter over this message, which now can directly be handled by stateless CN:

$$\mathtt{A} \in I_i^A \wedge (\exists j : [\mathtt{X}_j, \mathtt{Y}_j] \subseteq I_i^C).$$

Compared with the approach in Section 3.1.1, which sends out k_R unicast messages with a total size of $O(k_R h_R)$, this approach sends out a single CN message with size $O(h_R + k_L)$, where k_L, the number of points in the descriptive skyline, can be potentially much smaller than k_R, the number of points covered by the skyline.

Note that in general there can be many possible descriptive skylines. We now examine several techniques for computing one. Several factors influence our choice. We want to: 1) compute the skyline efficiently; 2) reduce the number of skyline points, because it affects the reformulated message size; 3) reduce the area of the subspace covered by the skyline, because a larger area may cause slightly higher dissemination costs for some CN implementations.

Minimum-Area Skyline (Ref-SJ-Sub) This is the skyline constructed in the proof of Theorem 2. While it minimizes the area, it may still have a large number of points. Furthermore, computing this skyline requires additional $O(k_R \log k_R)$ time after identifying all k_R points, which is not very desirable.

Joining-Tuple Skyline (Ref-SJ-Tup) Suppose there are s joining S tuples, with $S.C$ values $c^1, \ldots, c^s$. Interestingly, the skyline with points $(c^1, c^1), \ldots, (c^s, c^s)$ is also a descriptive skyline, as illustrated by Figure 2 (b). While very easy to compute, the number of skyline points may be large and can even exceed k_R.

Cluster-Based Skyline (Ref-SJ-Clu) Here, as in Section 3.1.1, we again leverage the SSI to exploit the clustering among subscription ranges. We start with an empty point set P. For each cluster $\mathcal{X}^{(k)}$, we look for the two joining S tuples with $S.C$ values (say x_1 and x_2) that are the closest possible to $p^{(k)}$ on each side of $p^{(k)}$. They can be found by looking up $(b, p^{(k)})$ in the B-tree index on $S(B, C)$. We add (x_1, x_1) and (x_2, x_2) to P. Intuitively, the set of affected R-semijoins in this cluster is exactly the set of selected R-semijoins in this cluster that lie to the northwest of (x_1, x_1) or (x_2, x_2). The justification (omitted for brevity) follows the same line of reasoning as the SSI-based algorithm in Section 3.1.1.

When all clusters have been processed, the point set P specifies a descriptive skyline with up to 2τ points, as illustrated by Figure 2(c). This descriptive skyline can be quite succinct for workloads that exhibit a high degree of subscription clustering. The running time of the algorithm is $O(\tau \log |S|)$.

Improved Cluster-Based Skyline (Ref-SJ-Clu$^+$) The skyline of Ref-SJ-Clu can be further compressed. Let $(q_1, q_1), \ldots, (q_n, q_n)$ be a contiguous subsequence of skyline points, sorted from southwest to northeast. Consider the sawtooth region between the skyline and the diagonal line segment from (q_1, q_1) to (q_n, q_n). If this sawtooth region contains no R-semijoin that is both selected and unaffected, we can replace the subsequence of skyline points by a single point (q_n, q_1) and obtain a simpler descriptive skyline, as illustrated by Figure 2 (d).

This improvement, which we call Ref-SJ-Clu$^+$, can be realized algorithmically as follows. We generate the Ref-SJ-Clu skyline points in order, by processing clusters in the increasing order of their anchors. During this process, we check, for each pair of consecutive skyline points (q_j, q_j) and (q_{j+1}, q_{j+1}), whether the triangle they form together with (q_j, q_{j+1}) is *disposable*, i.e., contains no R-semijoin that is both selected and unaffected. Once we identify a maximal consecutive sequence of at least two disposable triangles, we can replace the corresponding skyline points by a single one. We can locate all disposable triangles using at most τ R-tree lookups (see Appendix B for details). The number of Ref-SJ-Clu$^+$ skyline points is still 2τ in the worst case, but in practice we find the number to be much lower. Although the algorithm by itself is more costly than Ref-SJ-Clu, it can be combined with the algorithm for computing reformulated messages for S-semijoins (Section 3.2.2) without increasing the overall asymptotic complexity.

Finally, we note that even Ref-SJ-Clu$^+$ may not find a descriptive skyline with the minimum number of points. It is possible to find such a skyline, but the running time would become $O(k_R^2)$. We leave a more detailed investigation of this alternative as future work, since our experiments show that Ref-SJ-Clu$^+$ works well in practice and offers very good compression at low cost.

3.2.2 *Reformulating S-Semijoins*

Given an insertion t_R, our overall strategy is to first find the set of S tuples that are exposed to at least one S-semijoin; for each such S tuple, we create a CN message containing its content plus some additional information so that CN can route it to the exact set of S-semijoins to which it is exposed.

The algorithm proceeds as follows. Given $t_R = \langle a, b, p_r \rangle$, we first compute $a^-(t_R)$ and $a^+(t_R)$ by looking up (b, a) in the B-tree index on $R(B, A)$, as in Section 3.1.2. Next, we compute $\mathcal{I}$, the union of I_i^C ranges of all affected S-semijoins, by iterating through subscription clusters. For each cluster $\mathcal{X}^{(k)}$, we find the affected R-semijoins using two R-tree lookups, as in Section 3.1.1. For every affected R-semijoin X_i^R with $I_i^A \subset [a^-(t_R), a^+(t_R)]$, we add its I_i^C range to $\mathcal{I}$. During this process, we maintain $\mathcal{I}$ as a sequence of maximal, non-overlapping ranges. Finally, for each range I in

$\mathcal{I}$, we retrieve all S tuples with $S.B = b$ and $S.C \in I$, using a B-tree index on $S(B, C)$. For each retrieved S tuple t_S, we inject the following reformulated CN message:

$$\langle \mathtt{B}{:}b, \mathtt{C}{:}t_S.C, \mathtt{P_S}{:}t_S.p_s, \mathtt{A}{:}a, \mathtt{A^-}{:}a^-(t_R), \mathtt{A^+}{:}a^+(t_R)\rangle.$$

An S-semijoin is reformulated as a simple filter over this message:

$$\mathtt{A} \in I_i^A \subset [\mathtt{A^-}, \mathtt{A^+}] \wedge \mathtt{C} \in I_i^C.$$

For one incoming event, the number of notification messages is s', the number of S tuples exposed to at least one affected S-semijoin. The total message size is $O(s'h_S)$. The running time of the algorithm is $O(\log|R| + \tau(\log|S| + g) + k_R \log k_S + k_S \log|S| + s')$.

3.2.3 Recap and Comparison with Enum-SJ

To recap, Ref-SJ operates as follows. When a select-join subscription is created, it is decomposed into R- and S-semijoins, which are then reformulated into message filters (as described in Sections 3.2.1 and 3.2.2, respectively) and registered in the CN.

Given an incoming insertion t_R into R, the server generates a message containing t_R together with the descriptive skyline (Section 3.2.1). When injected into CN, this message is automatically routed by CN to reach all affected R-semijoins. Also, the server generates a series of messages, one for each S tuple that is exposed to least one S-semijoin (Section 3.2.2). Every message is augmented with $t_R.A$, $a^-(t_R)$, and $a^+(t_R)$, and is automatically routed by CN to reach any affected S-semijoin that it is exposed to.

Compared with Enum-SJ, not only does Ref-SJ leverage CN for efficient dissemination, but Ref-SJ also makes it possible to speed up server processing. Intuitively, the complexity of Enum-SJ, as any method that enumerates all affected subscriptions, is fundamentally lower-bounded by the number of affected subscriptions. Reformulation-based approaches, on the other hand, only need to generate a description of this set, which can be much more concise.

4 Multi-Way Select-Joins

We now consider how to handle a general mix of select-join subscriptions over multiple tables. Recall from Section 2 that we represent the schema as a join graph, and each select-join has a signature corresponding to a connected subgraph of the join graph.

Our approach decomposes each n-way select-join over n input tables into n binary select-semijoins (exactly one for each input table). The select-semijoin for input table R is the semijoin of R with one of the neighboring input tables of R (say S) in the join signature, i.e., $\sigma_{p_R} R \ltimes_B \sigma_{p_S} S$, where B is the common join attribute, p_R and p_S refer to the selection conditions on R and S in the original subscription. We call this binary select-semijoin an *R^S-semijoin*; here, R^S denotes the *form* of the semijoin, where R is called the *base table* and S is called the *filtering table*. For example, the five-way select-join whose signature is shown in Figure 3 might be decomposed into 5 semijoins: R^S, S^U, T^S, U^S, and V^U. Note that there are three choices of filtering table for base table S, corresponding to the three edges incident to S.

Instead of the original multi-way select-join subscription, the subscription client would maintain the decomposed binary select-semijoins, and simply join them together to reconstruct the content of the original subscription. Note that the client does not need to apply additional selections, because the semijoins carry all applicable local selections from the original subscription.

Using the techniques from Section 3, we process the decomposed binary select-semijoins in groups, where each group contains all binary select-semijoins of the same form.

Optimizing Decomposition Given a set of multi-way select-joins (with different signatures in general), we are faced with the following optimization problem. How should we decompose each of these multi-way select-joins such that the resulting groups of binary select-semijoin are least expensive to process? We refer to the decomposition decisions we make on the given set of select-joins collectively as a *decomposition* for this set.

For some intuition, suppose we want to choose a binary select-semijoin for a base table R in a multi-way select-join. Consider the implications of choosing a particular filtering table S. 1) On the insertion of tuple t_R into table R, t_R has to be sent to all t_R-affected R^S-semijoins. The associated cost depends on the probability of insertion into table t_R and the expected number of t_R-affected R^S-semijoins. Intuitively, we want to choose a "selective" filtering table S that is expected to lead to fewer t_R-affected R^S-semijoins. 2) On insertion of a tuple t_S into table S, each of the joining R tuples needs to be sent to all R^S-semijoins to which it is exposed. The associated cost depends on the probability of insertion into table t_S, the expected number of joining R tuples, and the expected number of t_S-affected R^S-semijoins. Intuitively, we again prefer to choose a "selective" neighbor S whose updates impact R^S-semijoins minimally. Note that choosing R^S-semijoin does not imply a preference for choosing S^R-semijoin, i.e., the choice of filtering table for each base table need not be reciprocal.

We cost a decomposition as follows. The total cost is the sum over all resulting *semijoin groups*. A semijoin group (say R^S) contains all decomposed binary select-semijoins of the same form (R^S-semijoins). The cost of the R^S group is the number of R^S-semijoins assigned to the group times a per-semijoin cost $c(R^S)$. We propose two techniques for estimating the per-semijoin cost for a group: One is based on periodic simulation of samples of subscriptions and events, and the other one is based on a simple parametric cost model (see Appendix D for details).

Given a set of multi-way select-joins whose signatures are drawn from a join graph, we use a greedy algorithm to find the decomposition with the lowest cost. We repeat the following for each node R in the join graph. We find the neighbor S with the lowest $c(R^S)$, and choose R^S-semijoins for all subscriptions involving both R and S. If there is any subscription involving R for which we have not yet chosen a semijoin for R, we repeat the process using the neighbor S' with the next lowest per-semijoin cost $c(R^{S'})$. After a choice for base table R has been made for all subscriptions involving R, we move on to another node in the join graph.

Under the assumptions of our cost function, it is easy to see that this greedy algorithm finds the optimal decomposition (see Appendix E for a proof). In Section 6, we shall see that optimization based on the simple parametric cost model gives good results.

Complexity Let e denote the number of edges in the join graph. The time complexity of the greedy algorithm is $O(e(\log e + |\mathcal{X}|))$, dominated by outputting the result decomposition. The part of the running time attributed to decision making is only $O(e \log e)$.

The total space required by all our auxiliary data structures is $O(\bar{n}\,|\mathcal{X}| + \sum_i e_i\,|T_i|)$, where $\bar{n}$ is the average number of tables in a multi-way join, and e_i is the number of edges incident to T_i in the join graph. Basically, each n-way select-join contributes n binary select-semijoins. For each group of binary select-semijoins, we need two SSIs—one for each of the two local selection attributes. An SSI (including R-trees for its clusters) takes space linear in the number of semijoins in the group. Therefore, the total space taken by SSIs is $O(\bar{n}\,|\mathcal{X}|)$. In addition, for each table, we need one composite-key B-tree for each of its join attributes (in combination with the local selection attribute). These B-trees together take $O(\sum_i e_i\,|T_i|)$ space.

Alternative Approaches There are a number of other approaches to handling multi-way select-joins, such as Rel-Sel and Ref-J (Section 2.3), as well as a non-trivial extension of the semijoin approach

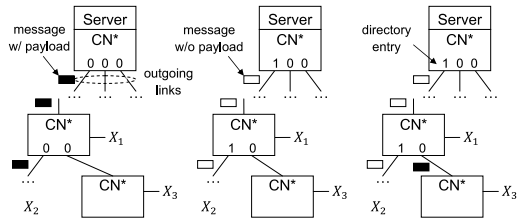

Figure 4: Payload dissemination using CN*.

to longer select-semijoins of the form $R \ltimes (S \bowtie T \bowtie \cdots)$. In particular, it may seem that this last alternative, with its greater filtering power, could be better than our approach of using only binary select-semijoins, because with only binary select-semijoins we might send tuples participating in a local binary select-join but not the original multi-way query. However, the alternatives have problems that make them less attractive than our approach; we refer the interested readers to Appendix F for details.

5 Reducing Re-Dissemination Redundancy

The techniques in Sections 3 and 4 avoid result representation, current-content, and inter-subscription redundancies, which cover a significant portion of network cost. However, they cannot completely avoid re-dissemination redundancy across events and subscriptions. In Example 1, a deletion from `Stocks` could remove the joining reviews from X_1's current content. A future update that brings them back into X_1 would cause the reviews to be sent again from the server. More generally, the bits delivered to a node are not available for reuse for a future event, because CN is stateless by design. This problem is exacerbated when tables have large payloads, such as inline text, video, etc.

The re-dissemination problem is not specific to join subscriptions, but is universal for any subscriptions requesting events with non-negligible payload. Hence, we opt for a general, CN-based solution that attempts to avoid retransmitting the same payload through the same link (between any two CN nodes). At the same time, we aim to maintain the clean, stateless interface of CN, and avoid introducing hard state or complex processing in the nodes.

One straightforward solution to the problem is caching. The idea is to assign each payload a unique reference, and push messages with the reference, but not the payload, to subscriptions. Upon receiving the message, a subscription client sends a request to the server for the payload with the reference. The request and its reply are routed over an overlay network that implements caching, so as to serve future requests before they reach the server. Unfortunately, this scheme fundamentally changes the push-style dissemination of publish/subscribe to pull (for payloads), which may not be acceptable to some applications. Moreover, cache misses add considerable latency, and new payloads will always result in initial cache misses, potentially causing a high amount of additional pull traffic.

Overview of CN* We now present our solution called *CN**. In order to avoid sending the payload over a link multiple times, we extend CN to maintain a *payload directory* and a *payload repository* at each node. The server assigns a unique ID for each payload (e.g., using a content hash). The payload repository essentially caches payloads along the push path. The payload directory remembers whether a particular payload has been sent through an outgoing link. Briefly, we avoid re-dissemination as follows: If a payload has been sent over an outgoing link, the node replaces the payload with just the payload ID, and sends this "lighter" version of the message instead; otherwise, the message is sent with the payload. The node responsible for a subscription will replace any payload ID in the message with the actual payload before finally forwarding the message to the subscription client.

Consider the example in Figure 4. On the first event, subscriptions X_1 and X_2, both interested in this event, receive the same payload through CN* (Figure 4 left). Later, if the same payload is needed again by X_1 and X_2 due to another event (which joins with the same tuple as the first event), CN* will send the message without the payload (Figure 4 center). Finally, if another event causes a third subscription X_3 to need the same payload, CN* will transmit the payload only along those links that have not sent it previously; other links will transmit without the payload (Figure 4 right).

One important feature of CN* is that both payload directory and repository in a CN* node are soft state whose size can be capped. We do not assume that each CN* node can remember the entire transmission history. Soft state also aids in failure handling, because we do not have to recover any state related to the payload.

Operational Details A standard CN message, which is a list of attribute-value pairs, is augmented with three attributes with special meanings to CN*. 1) PayloadAttrs specify which attributes in the message collectively form the payload. 2) ID is a unique identifier for the content of the payload in this message. 3) Source records the last encountered CN* node along the dissemination path, which may have a copy of the payload in its cache. If no such CN* node exists (e.g., when the message just enters the CN*), Source is set to the IP address of the server, which is ultimately responsible for supplying the content of the payload when necessary.

At each CN* node, the payload directory entries have the form [ID, Bitmap]. Bitmap is a bitmap with one bit for each outgoing link, which records whether a payload has been previously sent over that link. The payload repository entries have the form [ID, Payload], where Payload stores the actual payload content. Both the directory and the repository are indexed on ID to support quick access to a particular entry.

Assume for now that entries in the directory and repository are never purged. The server first injects the original message with the three additional attributes into CN*. Upon receiving a message m, each CN* node checks its directory to see if an entry for m.ID is present. If not, it creates a new directory entry with m.ID and an empty bitmap. It also adds the content of the payload, if available in m, to its payload repository. Link matching is then done just as in regular CN, to determine the set of outgoing links to which the incoming message needs to be forwarded. For each matching link, if the same payload was previously sent over that link (indicated by a 1 in the directory entry bitmap), the node sends the message without the payload (by removing values for attributes in m.PayloadAttrs). Otherwise, the node sends the message with the payload (reconstructed from the local repository if necessary), and sets the appropriate bit in the directory entry bitmap to 1. In either case, a node that has the payload in its repository always sets Source to its own address before forwarding the message.

Directory and Repository Maintenance To limit space usage, CN* may need to purge entries from the repository and/or directory. The choice of which entry to purge is analogous to a cache replacement policy; we use a least-recently-used (LRU) scheme. Further, if a directory entry is 1 for all outgoing links, it is okay to purge the entry from the repository. We next discuss the handling of purged repository and directory entries separately.

Handling Purged Repository Entries: Assume that a message m without the payload arrives at node n, but there is no repository entry for the payload at n (because it has been purged). If n needs to send the payload along some outgoing link or to a subscription client, n must first obtain the payload by contacting m.Source with m.ID. In this case, the increase in notification latency for subscriptions in n's subtree is the roundtrip time between n and m.Source.

Method	Delivery	Technique	Comment
Select	CN/CN*	Rel-Sel	Inject events in CN/CN* w/o joining (Sec. 2.3)
Select-Join	Unicast	Enum-J	Compute join results for each sub (Sec. 2.3)
	CN/CN*	Ref-J, -J+	Inject join results in CN/CN* (Sec.2.3)
Select-Semijoin	Unicast	Enum-SJ	Compute semijoins for each sub (Sec. 3.1)
	CN/CN*	Ref-SJ	Inject semijoin results in CN/CN* (Sec. 3.2); four flavors: -Sub, -Tup, -Clu, -Clu+

Table 1: Summary of solutions.

Note that if m.Source has dropped the entry in the interim period, n can directly contact the server as a fallback mechanism.

Handling Purged Directory Entries: When n receives a message whose payload ID corresponds to a directory entry purged earlier, n simply assumes that all bitmap entries are 0 (as if the payload is new). Thus, with purging of directory entries, n may send the same payload again along a link. However, note that the directory occupies very little space (a 512kB directory can hold tens of thousands of entries), so the chance of purging a useful directory entry is very low in practice, and re-dissemination can be usually avoided.

Remarks By design, CN* nodes can co-exist with regular CN nodes. This feature facilitates incremental deployment of CN*, consistent with our goal of practicality. Regular CN nodes simply forward the messages they receive without interpreting the additional attributes special to CN*. A CN* node always sends the payload if its next hop is a CN node (we have a simple and efficient technique for testing this case, described in Appendix H).

CN* differs from traditional caching in two important ways. First, traditional caching applies to values of identifiable objects, and hence must deal with coherency issues when values change. In contrast, CN* caches just values (of payloads), which identify themselves; each different value is a separate cacheable payload that is immutable by definition. Hence, CN* need not worry about cache updates. Second, a straightforward caching solution would generate lots of initial cache misses for any new payload, adding considerable notification latency. In contrast, CN* preserves the push-style dissemination of publish/subscribe. Dissemination of a new payload through CN* involves no misses and is identical in communication pattern to dissemination through regular CN.

6 Evaluation

Server Setup We have implemented all our novel schemes: Enum-SJ, Ref-SJ-Sub, Ref-SJ-Tup, Ref-SJ-Clu, and Ref-SJ-Clu+. For comparison, we have also implemented Enum-J, Rel-Sel, Ref-J, and Ref-J+. Table 1 summarizes the techniques for quick reference. Enum-J uses SSI [1] for computing select-join results. We support tuple inserts, deletes, and updates. The implementation writes its output to local disk at ~70 Mbps speed, which is roughly similar to a dedicated OC1 (~52 Mbps) connection to the Internet. We use main-memory data structures for optimal performance, though it should be easy to replace them with standard I/O-efficient versions if needed. The experiments are performed on a set of dual-core Intel Xeon 2.0GHz machines running Linux kernel 2.6.18.

Network Setup We evaluate network performance by implementing a simulator for large-scale networks. The simulator generates application-level routing traces that can then be analyzed using link-level simulation, which uses a 20,000-node topology produced by INET [9], a generator of Internet-like network topologies. A subset of 1000 nodes form the overlay dissemination network. In this paper, we focus only on measurements between these 1000 overlay nodes, because we have observed that IP-level costs generally follow similar trends as node-level costs.

The vanilla CN we have implemented is based on CAN [24], which uses a semantic space for routing (Appendix A). A similar CN has been used in other systems, e.g., [14, 5]. We also implement and report results using our CN* extension.

Evaluation Metrics We track both server- and network-side metrics. At the server, we measure the average processing time per event, including both server processing cost and the output of messages to be injected into CN. On the network side, we track: 1) *Network traffic* per event, which measures the total number of bytes transferred between overlay nodes. 2) *Number of overlay hops* per event, which measures the total number of messages sent between overlay nodes. 3) *Node stress* per event, which measures the communication load on an overlay node. In this paper, we report stress in terms of the total amount of traffic originating from a node. 4) *Hop latency*, which measures the number of overlay hops it takes for a notification message to reach a subscription. Hop latency roughly corresponds to subscription notification latency, assuming uniform network delays between nodes.

Workload For binary select-joins $R \bowtie S$ (see Example 2), we generate synthetic subscriptions as follows. Let $N(\mu, \sigma)$ represent a normal distribution with mean μ and standard deviation σ. Refer to Table 2 for a summary of parameters. We use normal distributions to generate the centers of subscription ranges over $R.A$ and $S.C$. The range centers are located in either low or high portions of event space, to model corresponding user interests. Range widths are derived using normal distributions as well (see Table 2).

We experiment with synthetic and real event workloads. The synthetic event workloads use 100 unique values of the join attribute. The number of S tuples for each join attribute value follows a truncated Zipf distribution with parameter 0.8. R tuples are inserted for each unique join attribute value, and 70% of R insertions produce at least one join result.[1] The total number of R tuples in the database is kept constant by deleting older tuples when necessary. We also experiment with a real event workload based on stock data from Yahoo! Finance (see Section 6.1 for details). Finally, the workloads for our mix of multi-way select-join subscriptions (general and star schema) are described in Section 6.3.

Repeatability To verify repeatability across runs, we perform each experiment multiple (up to 10) times, by varying the random seed for the event workload. We found the variation across runs to be minimal—for more than 90% of data points, the 95% confidence interval falls within $\pm 7\%$ of the respective mean. Given the significant difference (often orders of magnitude) across the approaches being compared, we plot only the mean value across runs.

6.1 Binary Select-Joins, Unmodified CN

We first examine the benefits of our new schemes, without the added benefits of CN*. Even without CN*, our techniques can easily outperform simpler ones. Unless otherwise indicated, these experiments use 100k subscriptions, with R and S tables having 16k and 5.1k tuples respectively. Each tuple has 100 bytes of payload.

Varying Number of Subscriptions In this set of experiments, we test scalability by varying the number of subscriptions from 100k to 1 million, and measure average costs per event (over 59k events). Note that the y-axis is logarithmic for all results.

The results for server processing time are shown in Figure 5. We see that Enum-J is the worst. Even with very efficient processing techniques, the output size dominates and makes this technique perform badly. Enum-SJ is better as it avoids result representation and current-content redundancies, but it still suffers from redundant output across subscriptions. Reformulation-based techniques are generally more efficient since they avoid enumerating affected subscriptions. The simple reformulations (Ref-J and Ref-J+) are

[1] We have derived this parameter from examining our real stock event workload for the fraction of stocks having at least one rating.

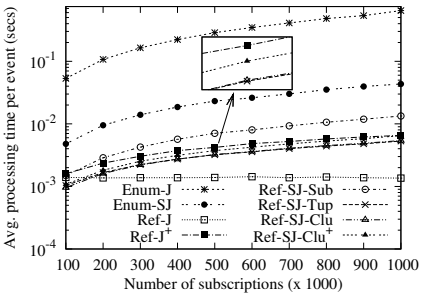

Figure 5: Processing time; increasing number of subscriptions.

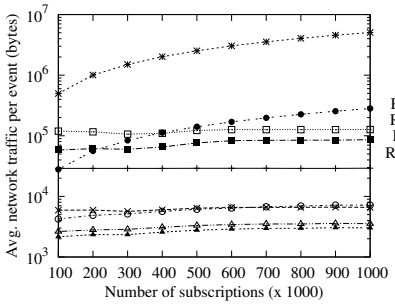

Figure 6: Network traffic; increasing number of subscriptions.

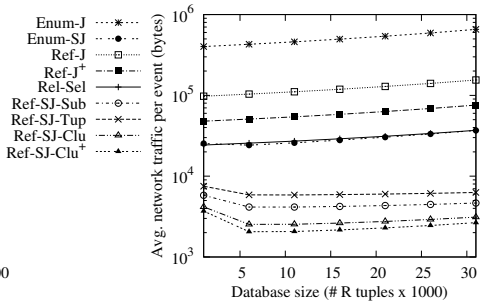

Figure 7: Network traffic; increasing database size.

parameter	value
domain of attributes	$[0, 100k]$
number of subscriptions	100k–1M
$R.A$ range centers	N(30k/70k, 10k)
$R.A$ range widths	N(20k, 5k)
$S.C$ range centers	N(15k/85k, 6k)
$S.C$ range widths	N(6k, 5k)
number of events	44k–74k
number of R tuples in DB	1k–31k
N_1	N(25k/75k, 3k)
distribution of $R.A$	N(N_1, 8k)
number of S tuples in DB	100–10100
N_2	N(50k, 10k)
distribution of $S.C$	N(N_2, 3k)

Table 2: Summary of parameters.

# subs.	100k	300k	500k
Enum-J	484.8	1464.0	2461.5
Enum-SJ	27.3	82.0	137.6
Ref-J+	13.4	14.3	14.6
Rel-Sel	0.86	0.95	0.88
Ref-SJ-Clu+	0.38	0.38	0.39

Table 3: Average server stress (kB).

# subs.	100k	300k	500k
Ref-SJ-Sub	73.12	99.54	111.84
Ref-SJ-Tup	312.78	305.88	302.85
Ref-SJ-Clu	29.59	38.15	42.87
Ref-SJ-Clu+	16.88	21.96	24.68

Table 4: Average description size (bytes).

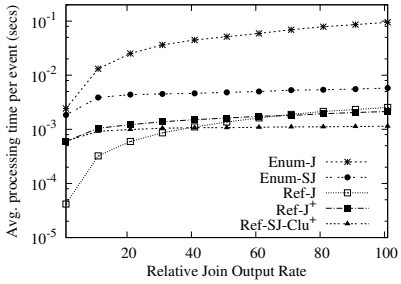

Figure 8: Processing time; increasing relative join output rate (RJOR).

quite fast, with Ref-J$^+$ being worse as it needs more processing to skip unnecessary join results. Semijoin reformulations (Ref-SJ-Sub, Ref-SJ-Tup, Ref-SJ-Clu, Ref-SJ-Clu$^+$) are fast as well. Ref-SJ-Sub is slower as it has to compute a skyline of affected subscriptions for each event. The compression techniques of Ref-SJ-Clu and Ref-SJ-Clu$^+$ create insignificant overhead above Ref-SJ-Tup.

Network traffic (Figure 6) for Enum-J is extremely high as expected. Enum-SJ is better, but degrades quickly with the number of subscriptions because of inefficient unicasting. Reformulation-based techniques are able to drastically reduce communication cost by leveraging CN. The simple reformulations (Ref-J and Ref-J$^+$), while better than Enum-J, still incur unnecessary traffic due to result representation and current-content redundancies. Relaxation (Rel-Sel) is slightly better, but at the expense of disseminating and adding irrelevant state at subscriptions. Our four semijoin reformulations avoid unnecessary dissemination and share costs across subscriptions. Ref-SJ-Clu$^+$ incurs the lowest traffic overall, giving at least an order of magnitude improvement over the strawman solutions. We also measured the number of overlay hops per event (Appendix J), and found that our semijoin reformulation techniques are at least an order of magnitude better than other schemes.

Table 3 compares the node stress at the server across various techniques, with Ref-SJ-Clu$^+$ representing the four semijoin reformulation techniques. As expected, Ref-SJ-Clu$^+$ generates the lowest stress, while enumeration-based techniques consume orders of magnitude more outgoing bandwidth at the server. From Table 4, we see that the descriptive skylines generated by Ref-SJ-Clu$^+$ are the most compact (with fewest number of points), beating more naive skylines (Ref-SJ-Sub and Ref-SJ-Tup) by a wide margin.

Varying Database Size We now examine the effect of increasing the number of R tuples in the database (older tuples are deleted as new ones are inserted, to keep the table size constant). Figure 7 shows the average network traffic. Other factors being equal, a smaller database implies that a new R tuple is likely to cause more subscriptions to need the joining S tuples, because it is less likely that a subscription already has a different R tuple with the same join attribute value. Hence, without CN*, semijoin reformulations degrade slightly in performance at low database sizes. However, they are still able to easily outperform the other approaches.

Varying Relative Join Output Rate *Relative join output rate* (*RJOR*) is the average number of join result tuples generated for each inserted event. Like join selectivity, RJOR can impact the performance of some algorithms. Figure 8 shows the server processing cost for increasing RJOR. We control RJOR by varying the number of tuples in table S. We see that Ref-J is very good at low RJOR, but quickly degrades due to output size. Ref-SJ-Clu$^+$ scales well with increasing RJOR. Figure 9 shows the network traffic for increasing RJOR. Again, semijoin reformulations schemes are clearly superior. The simple Ref-SJ-Tup degrades due to increasing S table size, but the other semijoin reformulations do well even at high RJOR. Although Ref-J and Ref-J$^+$ are good at low RJOR (due to lower result representation redundancy), they quickly degrade with increasing RJOR. Ref-SJ-Clu$^+$ is usually more than an order of magnitude better than the strawman techniques.

Results of Real Workload We gather real data from Yahoo! Finance to model Example 1. We obtain historical price-to-earning ratios (PER) of 100 random stocks, for a period of 7 months. The PER values are mapped to the range $[0, 100k]$ for use with our subscription workload. Stock ratings (ranging from 1 to 5) are also gathered, mapped, and perturbed using a normal distribution to derive 2300 unique stock ratings from the original set of 460 ratings. Subscription traces are the same as before. Figure 10 shows network traffic, as we increase the number of subscriptions. Enum-J and Enum-SJ are very expensive as expected, and degrade with number of subscriptions. Ref-J$^+$ performs better than before because the RJOR is lower (around 23). Still, Ref-SJ-Clu$^+$ is at least an order of magnitude better than the other schemes.

Other Experiments We have also experimented with varying payload sizes and subscription overlap, and evaluated how the "last hop" of dissemination (from overlay nodes to clients) affects our techniques. See Appendix J for details. Briefly, we confirm the intuition that the savings offered by our schemes over strawman so-

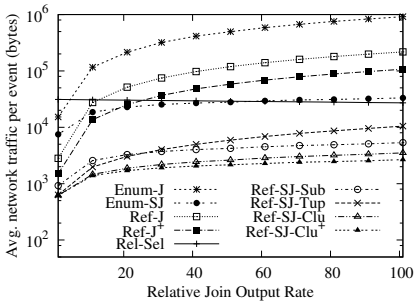

Figure 9: Network traffic; increasing relative join output rate (RJOR).

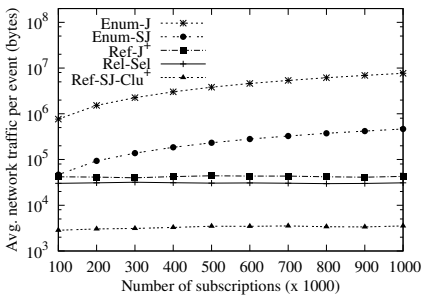

Figure 10: Network traffic; real event workload.

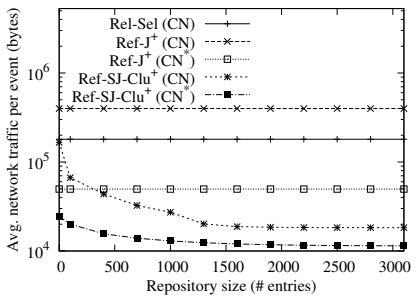

Figure 11: Network traffic; increasing repository size.

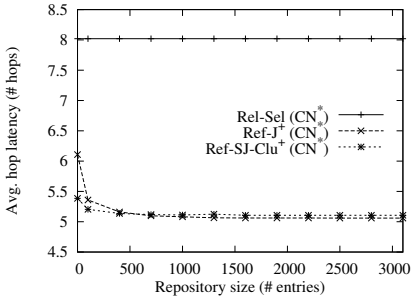

Figure 12: Hop latency; increasing repository size.

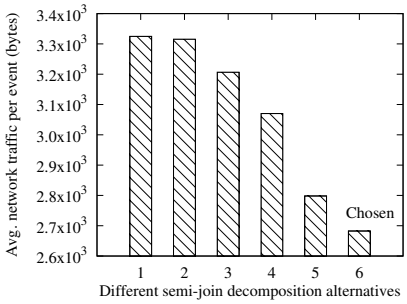

Figure 13: Network traffic; multi-way join mix, general schema.

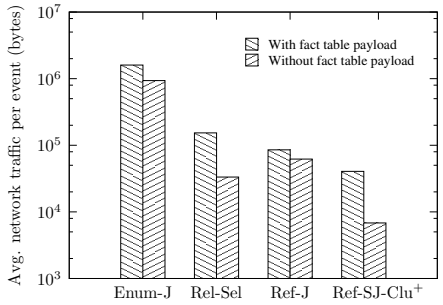

Figure 14: Network traffic; multi-way join mix, star schema.

lutions increase with payload size as well as subscription overlap. Moreover, the consideration of "last hop" strengthens the conclusion on the superiority of our techniques.

6.2 Results of Adding CN*

We now examine the additional performance benefits of using CN*, which can reduce re-dissemination redundancy. We only show Ref-SJ-Clu+ and Ref-J+ (with and without CN*) since these perform the best among their respective variants. We also show Rel-Sel for comparison. Enum-J and Enum-SJ do not use CN or CN*.

Effect on Traffic We set the payload directory to be 512kB, and vary the repository from 0 to 3100 entries. R and S tuples carry 1000-byte payloads. The size of R is kept low (500) to ensure that the cost of Ref-SJ-Clu+ is significant. Figure 11 shows network traffic. CN* reduces re-dissemination redundancy for Ref-SJ-Clu+ and Ref-J+, even with small repositories. Ref-J+ benefits more as it disseminates more unnecessary data, leaving more room for CN* to improve. Ref-SJ-Clu+ is better overall. Even with 0 repository size, the directory reduces cost by having only nodes with affected subscriptions pull data from the server. Further, a repository of just 400 entries reduces server stress (not shown in this figure) by a factor of 7 for Ref-J+ (3 for Ref-SJ-Clu+) compared with CN.

Effect on Hop Latency Figure 12 shows the average hop latency across all subscriptions and events. The size of R is very low (200) to penalize Ref-SJ-Clu+. We see that even at low repository sizes, the potential extra roundtrip does not increase the hop latency by much. Again, Ref-SJ-Clu+ is impacted minimally at low directory sizes because it relies less on CN* to perform well. Note also that Rel-Sel has a higher hop latency, since a message needs to reach many more subscribers dispersed across many nodes.

6.3 Multi-Way Select-Joins

We use a join graph with tables R, S, T, and U, with S in the center connecting the other three tables. We experiment with a mix of 50k $R \bowtie S$, 50k $R \bowtie S \bowtie T$, 50k $R \bowtie S \bowtie T \bowtie U$, and 20k $S \bowtie T \bowtie U$ queries (all with selections). Subscriptions and events are generated using normal distributions as before.

General Schema Here, R, S, T, and U have 10, 70, 70, and 30 tuples per unique join attribute value, respectively. Enum-J and Ref-J are found to be prohibitively expensive due to the large number of join results for the multi-way joins. Rel-Sel is found to generate 23kB traffic per event, around 9 times worse than the optimal Ref-SJ-Clu+ semijoin decomposition. In Figure 13, we compare the costs of six different random decompositions (without CN*). For the three semijoin groups with S as the base table, our cost model orders their per-semijoin costs as: $c(S^R) > c(S^U) > c(S^T)$. The corresponding greedy decomposition (rightmost bar) is optimal and gives around 20% lower traffic than the worst one.

Star Schema Star schemas are common in practice, and their referential integrity constraints lead to very small RJOR. In particular, each fact table insertion produces exactly one join result. For such "easy" joins, one might expect the strawman solutions to perform as well. This experiment, however, shows that our techniques still have a significant advantage. We consider the same join graph as before, where S is now a fact table with 25,000 tuples and the others are dimension tables with 1000 tuples each. Dimension table tuples have 100-byte payloads. Figure 14 shows the results. When the fact table has no payload, Ref-SJ-Clu+ is 5, 9, and 127 times better than Rel-Sel, Ref-J, and Enum-J respectively, because it disseminates the bulky dimension table tuples only when needed. Ref-J has to send out complete join results. The advantage is less when the fact table has equal payload (100 bytes) because it diminishes the relative advantage (all schemes have to send the bulky S tuples for each S insertion). Yet we find that Ref-SJ-Clu+ outperforms other techniques by a wide margin (even without CN*).

7 Related Work

Continuous Query Systems Continuous query systems (e.g., [21, 10, 15]) can be regarded as a form of publish/subscribe, where continuous queries over streams correspond to our subscriptions. NiagaraCQ [10] supports select-join processing at a server. CACQ [22]

group-processes filters, and supports dynamic reordering of joins and filters. PSoup [8] exploits set-oriented processing on joins with arbitrary join conditions. These systems correspond to Enum-J: They ignore the dissemination aspect and do not jointly optimize processing and dissemination. Consequently, they cannot avoid the redundancies intrinsic to producing traditional join results. Cayuga [15] supports queries joining two XML streams, but their schemes also ignore dissemination and are optimized for value joins over XML. We focus on relational select-joins, support multi-way joins, and consider both processing and dissemination.

Publish/Subscribe Systems Several publish/subscribe systems have made the subscription language more powerful (e.g., [11, 19, 5, 6, 12].). *SMILE* [19] supports SQL queries, while *PADRES* [12] supports subscriptions that can express correlations across events. These systems add application-specific logic and state into the network and do not optimize for group-processing or disseminating select-join subscriptions with varying selection predicates. They operate similarly to Ref-J, and can reduce only inter-subscription redundancy. We process queries efficiently, reduce all types of redundancies, and use a simple CN interface for efficient dissemination. Our techniques can be employed by these systems to handle a large number of multi-way select-join queries efficiently. In earlier work [5, 6], we have used reformulation to support complex queries over CN. However, [5] focuses on range aggregation, while [6] tackles subscriptions with value-based notification conditions.

Distributed Joins Distributed join processing systems, which distribute state across overlay nodes, correspond to Rel-Sel if selects are applied first. PIER [16] supports SQL queries (including joins) over DHTs, but targets one-time queries and does not optimize for multiple subscriptions. Idreos et al. [18] support two-way joins over overlay networks by re-indexing queries and routing tuples to them. This can incur high overhead because each query may be replicated for every unique join attribute value, and selects are done only as post-processing. Ahmad et al. [2] tackle distributed joins, but they focus on network locality and data locality issues, with the objective of reducing delay. These systems add complexity by designing new distributed schemes with application-specific logic and state in the network. We optimize processing and dissemination of a mix of multi-way select-join queries, and use the simple, stateless, off-the-shelf CN interface, making our novel techniques easy to deploy and manage, yet ensuring very high efficiency.

Other Related Work Semijoins have been employed by many systems [3, 25] to reduce communication in distributed databases. Work on view maintenance (e.g., [20, 17, 23]) also considers joins. However, they do not address the problem of simultaneously supporting a large number of select-joins. Moreover, like Enum-SJ, they do not reduce redundancies across queries and updates.

8 Conclusions

A publish/subscribe system needs to optimize both subscription processing and result dissemination, particularly for complex subscriptions such as select-joins. To develop an end-to-end solution to support a large mix of multi-way select-joins, we identified several key redundancies in traditional techniques, and reduced these redundancies using novel semijoin-based reformulation schemes. The schemes are easy to deploy and maintain, yet ensure very high efficiency. We also proposed an extension (CN*) to content-driven networks, which further reduces redundancy in disseminating bulky payloads. Extensive experiments on real and synthetic workloads validated the benefit of our schemes, and demonstrated orders-of-magnitude improvement over standard techniques for both server and network metrics.

References

[1] P. K. Agarwal, J. Xie, J. Yang, and H. Yu. Scalable continuous query processing by tracking hotspots. In *VLDB*, 2006.

[2] Y. Ahmad, U. Cetintemel, J. Jannotti, and A. Zgolinski. Locality aware networked join evaluation. In *NetDB*, 2005.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. J. B. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM TODS*, 1981.

[4] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, 2001.

[5] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD*, 2006.

[6] B. Chandramouli, J. M. Phillips, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems. In *VLDB*, 2007.

[7] B. Chandramouli, J. Yang, P. K. Agarwal, A. Yu, and Y. Zheng. ProSem: Scalable wide-area publish/subscribe. In *SIGMOD*, 2008.

[8] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB J.*, 2003.

[9] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, 2002.

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[11] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.

[12] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *FIW*, 2005.

[13] A. Gupta and I. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[14] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Middleware*, 2004.

[15] M. Hong et al. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.

[16] R. Huebsch et al. Querying the internet with PIER. In *VLDB*, 2003.

[17] N. Huyn. Speeding up view maintenance using cheap filters at the warehouse. In *ICDE*, 2000.

[18] S. Idreos, C. Tryfonopoulos, and M. Koubarakis. Distributed evaluation of continuous equi-join queries over large structured overlay networks. In *ICDE*, 2006.

[19] Y. Jin and R. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *DEBS*, 2003.

[20] B. Liu and E. Rundensteiner. Cost-driven general join view maintenance over distributed data sources. In *ICDE*, 2005.

[21] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *TKDE*, 1999.

[22] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[23] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *PDIS*, 1996.

[24] S. Ratnasamy et al. A scalable content addressable network. In *SIGCOMM*, 2001.

[25] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. Integrating semi-join-reducers into state-of-the-art query processors. In *ICDE*, 2001.

APPENDIX

A Additional Background on CN

Refer to our introduction to content-driven networks (CN)[2] in Section 2.1. Many structured and unstructured overlay networks can be classified as CN. We next describe several instances of CN with varying degrees of expressiveness in the predicates they support. The appropriate choice of CN may depend on convenience and the desired level of expressiveness (application-dependent).

Content-Based Networks *Content-based networks (CBN)* [4] are perhaps the most general incarnation of CN. They support messages with arbitrary attributes and destinations with interests expressed as arbitrary boolean predicates involving message attributes. There are many systems that use content-based networking for dissemination, e.g., SIENA [26], Gryphon [34], REBECA [33], Hermes [36], PADRES [12], JEDI [31], XNet [28], etc. Hence, a CBN is easy to adopt in a large-scale system.

A content-based network can accept *messages* to be forwarded to a set of matching *destinations*. A message is a set of attributes following some schema. For example, a message M with a schema that includes `Symbol` and `Price` as two attributes, might look like $\langle$`Symbol`: "GOOG", `Price`: $550, \ldots\rangle$. Destinations, on the other hand, are predicates over the message attributes. For example, a destination may be expressed as (`Symbol` = "GOOG")$\wedge$(`Price` $\in [500, 600]$). Notice that the message M matches this destination, and therefore M would be delivered to this destination.

Operational Details: The network is responsible for delivering every message to all matching destinations. Message delivery is performed in a multi-hop manner over an overlay network. Delivery consists of two phases. The first phase is the establishment of flow paths through the network, by creating local forwarding tables at each node. A forwarding table is used to decide which outgoing links a given message should be sent over, and this matching process of a message at a node is the second phase. Taken together, the forwarding performed at the nodes causes messages to be routed through the network, until they reach all the affected destinations. A CBN uses sophisticated techniques [26, 34] to build concise forwarding tables at the nodes, and to perform forwarding efficiently. Figure 15 shows the routing of a message using forwarding tables, for 4 subscriptions with predicates on an attribute X.

Content-Addressable Networks Another example of CN, which supports less expressive subscriptions, is a *content-addressable network* (*CAN*) [24]. A CAN is a decentralized structured overlay network that uses a logical d-dimensional Cartesian coordinate space that is partitioned across all participating peers. Each peer is responsible for a subspace in the form of a hypercube, called a *zone*. Each zone has knowledge of only its immediate neighbors, and can route messages only to them. Routing from a source point to a destination point in the CAN space is carried out in multiple hops until the destination is reached.

We can use a CAN to build a structured stateless dissemination layer for a publish/subscribe system. Assume that each subscription is mapped as a point in the CAN space. For example, subscriptions with k range selection predicates (over k attributes) can be indexed as a point in a $2k$-dimensional CAN as follows. Each range selection predicate is mapped to two dimensions in the CAN space, one for the low end of the range and the other for the high end of the range. Figure 16 illustrates a 2-d CAN. Note that in general, a subscription could be mapped as a point in d-dimensional CAN space based on the values of any d subscription-specific parameters (not necessarily range predicates).

The CAN space is partitioned into rectangular zones, each with a *zone owner*—an overlay network node responsible for all the subscriptions in its zone. Partitioning of the CAN space into zones can use load balancing criteria [14]; for example, the number of subscriptions residing in the zone and the number of events handled by the zone. We inject a message into CN with additional state that can be interpreted as defining an arbitrary complex *region* in CAN space that we wish to cover. Every subscription lying within the region is considered affected by the message. CAN routing can easily be adapted to reach all zones within a specified region. For example, *Meghdoot* [14] is a publish/subscribe system that uses a CAN-based CN, but supports only stateless subscriptions with simple range predicates.

Expressiveness: If subscriptions are mapped to the CAN space based on their range predicates, a hypercube region in a CAN space of d dimensions can express a conjunction of d predicates, where each predicate specifies either 1) containment of a subscription's range predicate within a specified range, or 2) containment of a specified range within a subscription's range predicate. For example, a CN message $\langle \ldots$,`UL_X`: a, `UL_Y`: b, `LR_X`: c, `LR_Y`: $c\rangle$ may be interpreted as a rectangular region (in 2-d CAN) with upper-left coordinate (a, b) and lower-right coordinate (c, c), shown shaded in Figure 16. This description identifies every subscription whose range predicate 1) is contained within the range $[a, b]$, and 2) contains the range $[c, c]$. Equivalently, in a CBN we could rewrite every subscription X_i with range predicate $[low_i, high_i]$ as the predicate $([low_i, high_i] \subseteq [\texttt{UL\_X}, \texttt{UL\_Y}]) \wedge ([\texttt{LR\_X}, \texttt{LR\_Y}] \subseteq [low_i, high_i])$. Note that unlike a CBN, a CAN-based CN cannot support arbitrary predicates, including keyword matches and user-defined functions.

Other CN Instances Many other networks fall under the CN umbrella, including multicast networks (e.g., [27, 35]) and distributed indexes such as prefix hash trees [29], P-trees [30], SD-Rtrees [32], etc. However, these mechanisms have limited expressiveness. For example, a multicast network supporting multiple multicast groups can be viewed as CN because messages carry a group ID attribute. Destination interests, implied by group memberships, can be regarded as message predicates that select particular group IDs. A distributed 1-d range search index (e.g., [29]) is also an instance of CN, because we can regard a node responsible for data item s as interested in all range search messages satisfying the predicate $(\texttt{S\_L} \leq s) \wedge (s \leq \texttt{S\_R})$, where `S_L` and `S_R` are the two message attributes denoting the left and right endpoints of the search range.

B Disposable Triangles in Ref-SJ-Clu$^+$

Refer to the discussion on the improved cluster-based skyline (Ref-SJ-Clu$^+$) in Section 3.2.1. We can locate all disposable triangles using at most τ R-tree lookups. Specifically, we check whether a triangle between a pair of consecutive skyline points (q_j, q_j) and (q_{j+1}, q_{j+1}) is disposable as follows.

If there is no cluster anchor between q_j and q_{j+1}, the triangle cannot contain any subscription at all (otherwise this subscription would not belong to any cluster); therefore, the triangle is obviously disposable.

Suppose there are one or more cluster anchors between q_j and q_{j+1}. For each such cluster anchor $p^{(k)}$, we look up $(p^{(k)}, a)$ in the cluster R-tree. For each t_R-selected R-semijoin returned by the lookup, we check if that semijoin is also t_R-affected, by testing whether it contains either (q_j, a) or (q_{j+1}, a). As soon as we

[2]Terms such as *content-based routing*, *content-based networking*, and *semantic multicast* capture similar concepts. We choose not to use these terms because they are often associated with specific projects and systems, e.g., [26, 35]; we want to capture a broader class of systems with different designs and varying degrees of expressiveness.

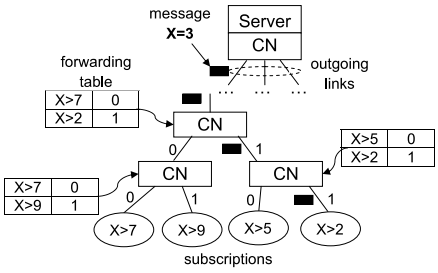

Figure 15: Using a content-based network for publish/subscribe.

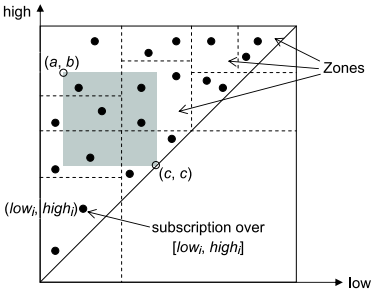

Figure 16: Using a content-addressable network for publish/subscribe.

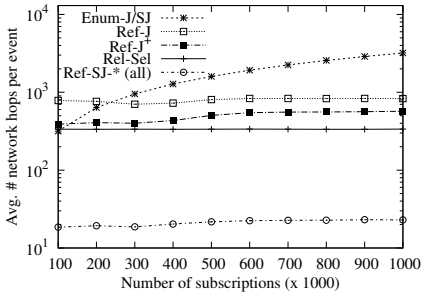

Figure 17: Network hops; increasing number of subscriptions.

encounter a semijoin that is not t_R-affected, we can terminate the process immediately for that triangle, and declare that triangle to be not disposable. Otherwise, the triangle is reported as disposable.

Because of the above termination condition, for each group, we examine at most one semijoin that is not t_R-affected; all other semijoins we examine are t_R-affected. Therefore, the total running time for locating all disposable triangles is $O(\tau g + k_R)$, where k_R is the total number of t_R-affected semijoins.

C Multi-Attribute Conditions

Multi-attribute equijoin conditions are straightforward to handle, as we can conceptually treat the set of join attributes as a single composite attribute. Consider the binary join between tables R and S with join attributes $(B_1, \ldots, B_m)$. The only change to our algorithms is to use B-trees indexing composite keys $(B_1, \ldots, B_m, A)$ and $(B_1, \ldots, B_m, C)$, instead of (B, A) and (B, C) respectively.

Now, suppose the local selection conditions on R and S are conjunctions of d_R and d_S range conditions, respectively. Consider the case of inserting an R tuple t_R (the case of inserting an S tuple is symmetric). We extend our semijoin reformulation techniques in Section 3 as follows. A subscription is a hypercube with $d_R + d_S$ dimensions, one for each local selection attribute. The joining S tuples can be mapped as a set of points $\mathcal{J}_S$ in a d_S-dimensional space, which we call the *S-space*, whose dimensions correspond to the local selection attributes of S.

Handling R-Semijoins The reformulation scheme is the same as before, but the descriptive skyline is in a $2d_S$-dimensional space instead of a 2-dimensional space. For each local selection attribute of S, there are two dimensions in this space that correspond to the two endpoints of a range over this attribute. We call this $2d_S$-dimensional space the S^2-space.

Finding a descriptive skyline becomes more complicated in the S^2-space. The joining-tuple skyline (Section 3.2) can be directly derived from $\mathcal{J}_S$—each point $(c_1, \ldots, c_{d_S})$ of $\mathcal{J}_S$ in the S-space is converted into a skyline point in the S^2-space by simply repeating each coordinate twice, i.e., $(c_1, c_1, \ldots, c_{d_S}, c_{d_S})$.

We next describe how to derive the cluster-based skyline. First, we introduce the concept of a *skyline envelope*.

Definition 2 (Skyline Envelope)**.** *$\mathsf{Sky}(\mathcal{X}, z)$, the* skyline envelope *of point z with respect to a set of points $\mathcal{X}$, is the set of all points $\mathcal{Y} \subseteq \mathcal{X}$ such that for each point $y \in \mathcal{Y}$, no point in $\mathcal{X}$ lies within the minimum hypercube containing both points z and y.*

For each cluster of subscriptions whose local selections on S are satisfied by a common cluster anchor $p = (c_1, \ldots, c_{d_S})$, we compute $\mathsf{Sky}(\mathcal{J}_S, p)$, i.e., the skyline envelope of p with respect to the joining S tuples in the S-space. Then, for each point in the skyline envelope, we convert this point from the S-space to the S^2-space, again by simply repeating each coordinate twice. The final cluster-based skyline consists of all such points in the S^2-space, obtained from the skyline envelopes of all subscription clusters.

Handling S-Semijoins For each joining S tuple t_S, consider the set of previously joining R tuples $\mathcal{J}_R$ as points in the d_R-dimensional *R-space* (defined analogously as the S-space). We compute $\mathsf{Sky}(\mathcal{J}_R, t_S)$, i.e., the skyline envelope of t_S with respect to $\mathcal{J}_\mathcal{R}$ in the R-space. The points in this skyline envelope are included in the reformulated message for t_S. Each subscription is reformulated to require that its selection conditions are satisfied by t_S and t_R, and that its selection conditions on R are satisfied by none of the points in the skyline envelope. The latter condition ensures that we do not notify subscriptions whose current contents already contain t_S due to joining with some other selected R tuple.

Discussion While the extension described above is very aggressive in trying to minimize the amount of data to be disseminated, it may be less desirable in practice due to difficulties with high-dimensional indexing and skyline computation, as well as large descriptive skylines and lower degrees of user-interest clustering expected in higher dimensions. Similar to practices in traditional databases, one solution to tackle the problem of higher dimensions is to choose one selection attribute (per table) for group processing and dissemination. The remaining selection conditions are applied by subscribers in a post-processing step. We leave the study of how to choose the best single selection attribute as future work.

D Estimating Per-Semijoin Cost

As briefly discussed in Section 4, we propose two methods for estimating the per-semijoin cost for a semijoin group in a decomposition of a set of multi-way select-joins.

Periodic Simulation We use a random sample of subscriptions and events, and simulate processing and dissemination for each possible group. We let each group R^S include all R^S-semijoins available for choice (only for the purpose of estimation—such assignments do not collectively form a valid decomposition). The per-group cost obtained from simulation is divided by the size of the group to give a per-semijoin cost for this group. This approach is general and can adapt to the actual subscription and event workloads, but simulation incurs overhead.

Parametric Cost Model To keep model complexity low, we make a number of assumptions and simplifications. Consider the select-semijoin $\sigma_{p_R} R \ltimes_B \sigma_{p_S} S$, with the CAN-style CN introduced in Appendix A. Assume that 1) subscriptions are uniformly distributed in terms of their range selection predicates, and 2) the events' local selection attribute values are uniformly distributed over their respective domains.

With a binary join, the CAN space is four-dimensional. Two dimensions correspond to the left and right endpoints of $R.A$ selection ranges. We call the projection of the CAN space onto these

two dimensions the R^2-*space*. The remaining two dimensions of the CAN space correspond to the left and right endpoints of $S.C$ selection ranges, and we call this space the S^2-*space*.

All subscriptions lie in what we call the *routing area* of the CAN space, which is the product of its projections onto the R^2-space and the S^2-space. The projection of the routing area onto the R^2-space (S^2-space) is the area to the upper-left of the diagonal of the R^2-space (S^2-space, respectively). We assume that the routing area is divided by a uniform grid into zones, each of which hosts approximately the same number of subscriptions. The cost of disseminating a message to a region depends on the number of zones covered during routing, and hence is roughly proportional to the fraction of the routing area covered by the region.

Our cost model uses the following parameters: p_R, probability that a given event is an insertion into table R; p_S, probability that a given event is an insertion into table S; j_R, expected number of R tuples having the same join attribute value; j_S, expected number of S tuples having the same join attribute value.

The cost of the select-semijoin has two components:

- *Cost due to insertion into R.* On an insertion t_R into R, consider first the projection of the affected region onto the R^2-space. The ratio of the area of the affected region to that of the routing area, in R^2-space, is at least 0 (when $t_R.A$ is one of the two extreme values of its domain) and at most $\frac{1}{2}$ (when $t_R.A$ is right in the middle of its domain). Assuming that $t_R.A$ is uniformly distributed, the expected ratio is $(\int_0^1 (1-x)x\,dx)/\frac{1}{2} = \frac{1}{3}$.

  Next, consider the projection of the affected region onto the S^2-space. The ratio of the area of the affected region to that of the routing area, in S^2-space, is at most $\frac{j_S}{j_S+1}$, achieved when the joining S tuples' local selection attribute values divides its domain into j_S+1 equal intervals. Assuming that the j_S values are drawn uniformly, the expected ratio turns out to be $\frac{j_S}{j_S+2}$, which can be calculated by

$$1 - \frac{\int \cdots \int_G (\sum_{i=1}^{j_S} x_i^2 + (1 - \sum_{i=1}^{j_S} x_i)^2)\, dx_{j_S} \cdots dx_1}{\int \cdots \int_G\, dx_{j_S} \cdots dx_1},$$

  where G is the volume $\{(x_1, \ldots, x_{j_S}) \mid x_1, \ldots, x_{j_S} \geq 0 \wedge \sum_{i=1}^{j_S} x_i \leq 1\}$.
- *Cost due to insertion into S.* On an insertion t_S into S, we need to send j_R messages, one for each joining R tuple. Consider the message for a joining R tuple t_R. In the R^2-space, the expected ratio of the area of the affected region to that of the routing area is $\frac{1}{3}$ as before. In the S^2-space, the affected region is characterized by a rectangle cornered at $(t_S.C, t_S.C)$ and $(c^-(t_S.C), c^+(t_S.C))$, where c^- and c^+ are defined analogously as a^- and a^+ in Section 3. Suppose there were j_S existing S tuples with the same join attribute value as t_S. Their C values divide a unit-size domain into j_S+1 intervals with lengths $x_1, \ldots, x_{j_S}$, and $x_{j_S+1} = 1 - \sum_{i=1}^{j_S} x_i$. The new insertion falls into the i-th interval with probability x_i; when that happens, the expected area of the affected region in the S^2-space is $\int_0^{x_i} (x_i - x)x\,dx = x_i^2/6$. Overall, in the S^2-space, the expected ratio of the area of the affected region to that of the routing area turns out to be $\frac{2}{(j_S+2)(j_S+3)}$, computed by

$$\frac{\int \cdots \int_G (\sum_{i=1}^{j_S} x_i^3/6 + (1 - \sum_{i=1}^{j_S} x_i)^3/6)\, dx_{j_S} \cdots dx_1}{\int \cdots \int_G\, dx_{j_S} \cdots dx_1} : \frac{1}{2},$$

  where G is the volume $\{(x_1, \ldots, x_{j_S}) \mid x_1, \ldots, x_{j_S} \geq 0 \wedge \sum_{i=1}^{j_S} x_i \leq 1\}$.

Since insertions into R and S occur with probabilities p_R and p_S respectively, the expected total cost is

$$\alpha \cdot p_R \cdot \frac{1}{3} \cdot \frac{j_S}{j_S+2} + \beta \cdot p_S \cdot j_R \cdot \frac{1}{3} \cdot \frac{2}{(j_S+2)(j_S+3)},$$

where α and β are constants.

E Optimality of Greedy Decomposition for Multi-Way Select-Joins

The greedy decomposition in Section 4 gives the optimal choice under our cost model. We prove this claim using a cut-and-paste argument as follows. If some table R were to select a binary semijoin R^S-semijoin with a per-semijoin cost c_1, where c_1 is not the minimum, i.e., c_1 is greater than the per-semijoin cost c_2 of some other binary semijoin (say R^T-semijoin) for R, then by selecting R^T-semijoin with per-semijoin cost c_2 instead of R^S-semijoin for table R, we would be able to reduce the total cost of the decomposition by $(c_1 - c_2)$, thus proving that the original choice of R^S-semijoin (with cost c_1) was suboptimal.

F Alternatives for Multi-Way Select-Joins

Relaxation Each query over d tables can be relaxed into d selection queries, similar to Rel-Sel (Section 2.3). However, this scheme may suffer from excessive notifications due to the relaxation.

Simple Join Reformulation On any insertion, we can derive and disseminate all the join tuples produced as a result of the insertion, similar to Ref-J and Ref-J$^+$ in Section 2.3. However, the problem of unnecessary data is exacerbated because each insertion into some table would generate all the new joining result tuples (along with the payload for each joining relation in a result tuple). This overhead could be quite large in case many tuples satisfy the join conditions. Furthermore, if there are multiple subscription signatures involving the table of insertion, join results need to be generated separately for every signature. When signatures overlap (e.g., $R \bowtie S \bowtie T_1$ and $R \bowtie S \bowtie T_2$), additional redundancy can arise across results for different signatures.

However, if the join is very selective (i.e., the number of result tuples generated due to an event is small), then this solution may be viable. Using CN* for routing can somewhat mitigate the problem of repeated dissemination of payload.

Fully Extending Two-Way to Multi-Way Joins We can directly extend the reformulation procedure for binary select-joins in Section 3 to consider longer semijoins instead of binary semijoins. We briefly outline the approach below, and point out why it may not work as well as our binary decomposition approach in Section 4.

We can group-process all join queries having the same join signature; let $\mathcal{Q}$ denote the corresponding join graph. For each table $R \in \mathcal{Q}$, removing R from $\mathcal{Q}$ would in general result in a set of (mutually disjoint) connected subgraphs which we call the *remainder graphs* of R; we denote this set by $\bar{\mathcal{R}}$. We define the $R^{\bar{\mathcal{R}}}$-semijoin as a semijoin of the form $R \ltimes (\times_{\mathcal{T} \in \bar{\mathcal{R}}} \bowtie_{T \in \mathcal{T}} T)$, with local selection conditions attached to appropriate tables. Before we outline an approach for handling $R^{\bar{\mathcal{R}}}$-semijoins, we introduce the notion of an *induced projected partial join*:

Definition 3 (Induced Projected Partial Join)**.** *A t_R-induced projected $\mathcal{T}$-partial join, where $\mathcal{T}$ is a connected subgraph of the join graph $\mathcal{Q}$ and t_R is a tuple from a table R connected to $\mathcal{T}$ in $\mathcal{Q}$, is the natural join over $\mathcal{T}$ and $\{t_R\}$, with no selection predicates applied, followed by a projection over the local selection attributes in $\mathcal{T}$. The result tuples of a t_R-induced projected $\mathcal{T}$-partial join can be regarded as points in a $|\mathcal{T}|$-dimensional space called the $\mathcal{T}$-space.*

Insertion into R On the insertion of a tuple t_R into table R, the reformulation for an $R^{\bar{\mathcal{R}}}$-semijoin is a conjunction of predicates, one for each remainder subgraph $\mathcal{T} \in \bar{\mathcal{R}}$. The predicate for each $\mathcal{T}$ is similar to that derived for R-semijoins in the multi-attribute selection case (Appendix C), with the difference that the descriptive skyline (in the $\mathcal{T}^2$-space) in this case is computed for the result tuples of the t_R-induced projected $\mathcal{T}$-partial join.

Insertion into S Here, S can belong to any one of the remainder subgraphs of R. Consider the set of all R tuples that join (directly or indirectly) with the insertion t_S. For each such joining R tuple, say t_R, we generate a message as follows:

- For each remainder subgraph $\mathcal{T}$ not containing S, we include in the message a descriptive skyline for the result tuples of the t_R-induced projected $\mathcal{T}$-partial join (just like the case of inserting t_R described above). Correspondingly, the reformulated subscription includes a condition that ensures that subscription contains at least one point of the descriptive skyline.
- For the remainder subgraph $\mathcal{T}_S$ containing S, consider $\mathcal{J}_{\mathcal{T}_S}$, the set of *old* result tuples of the t_R-induced projected $\mathcal{T}_S$-partial join, prior to the insertion of t_S. We compute the cluster-based descriptive skyline for all *new* result tuples of the t_R-induced projected $\mathcal{T}_S$-partial join (i.e., those involving t_S). Note that points in this cluster-based descriptive skyline are in the $\mathcal{T}_S^2$-space, but by construction of the cluster-based descriptive skyline, they have same coordinates in each pair of dimensions (left and right endpoints) that correspond to the same dimension in the $\mathcal{T}_S$-space. Therefore, we can "collapse" these points into a set of points P in the $\mathcal{T}_S$-space by removing one dimension from each pair. For each point in P, we compute the skyline envelope of the point with respect to $\mathcal{J}_{\mathcal{T}_S}$. We include both P as well as the skyline envelope points for each point in P in the reformulated message for t_R. Correspondingly, the reformulated subscription includes a condition that ensures that the subscription contains at least one point in P but none of its skyline envelope points.

Finally, the reformulated subscription also checks that t_R satisfies its local selection condition on R.

Discussion With the full-extension approach, the $R^{\bar{\mathcal{R}}}$-semijoins carry all conditions in the original queries, so every tuple in these semijoins participates in the final result of the original queries. In contrast, our binary decomposition approach in Section 4 does not offer this guarantee. On the other hand, as with the case of the multi-attribute extension in Appendix C, the higher dimensions pose practical issues. The reformulated messages are larger and the reformulated subscriptions contain more complex (though still stateless) predicates. Processing efficiency also suffers. Although binary semijoins lose some filtering power, they are simple to implement and efficient in practice. Furthermore, breaking queries up into binary semijoins creates more opportunities for group processing and dissemination, while the full-extension approach may end up with many more groups and fewer semijoins per groups.

G Forwarding Algorithm for CN*

Refer to Section 5. Algorithm 1 shows the CN* forwarding procedure for an incoming message m. For simplicity of presentation, we assume that there is only one attribute (Payload) in PayloadAttrs. In line 3, QUERY retrieves the payload directory and repository entries (represented together as X) for m.ID (a new directory entry is created if necessary). If the payload is present in the incoming message m, QUERY also adds the payload to the payload repository. Line 4 identifies the matching outgoing network interfaces just as in CN. In lines 7–11, we check the directory entry bitmap (X.Bitmap) for each affected interface. If the payload was previously sent over that interface, the node strips the payload from the message (line 8). Otherwise, if m does not already contain the payload, GETPAYLOAD (line 10) retrieves the payload (X.Payload) from the payload repository at the closest provider source (m.Source) or the server (in the worst case). The retrieved payload is added to the message (line 11). If the repository contains the payload, line 12 updates the Source field in the outgoing message to the machine's network address. Finally, the message is disseminated along the outgoing link (line 13).

Algorithm 1: Forwarding algorithm for CN*.

```
1  FORWARD (message m) begin
2      p ← ∅ ;                                  // placeholder for payload
3      X ← QUERY (m) ;      // query the directory and repository
4      H ← GETMATCHES (m) ;       // get the matching interfaces
5      foreach interface i in H do
6          m' ← m;
7          if X.Bitmap[i] = 1 then
8              m'.Payload ← ∅ ;          // payload sent previously
9          else if m.Payload = ∅ then
               // add payload to message
10             if p = ∅ then p = GETPAYLOAD (X, m.Source);
11             m'.Payload ← p;
12         if X.Payload ≠ ∅ then m'.Source ← local address;
13         DISSEMINATE (m', i) ;  // send message along interface i
14 end
```

H Co-Existence of CN* with Regular CN

Recall from Section 5 that CN* nodes can co-exist with regular CN nodes, which facilitates incremental deployment and adoption. We have developed an efficient technique for CN* to detect that a next-hop neighbor is a CN node, without adding overhead to the critical path of dissemination. When a payload is sent along an outgoing link i, the corresponding bit in the payload directory entry is not immediately set. Instead, it is set only if this node receives a special acknowledgment from the next hop (indicating that the next-hop node is a CN* node). Thus, in case the next hop is a plain CN node, the bit would never get set and the payload would always get sent along that link. The acknowledgment does not delay message forwarding at the next hop, and therefore does not increase notification latency.

I Select with Payload

A stateless selection subscription, supported by traditional publish/subscribe systems, can benefit from being expressed as a join. For example, a subscriber may be interested in receiving a news feed with all detailed product information for products whose ratings fall within some prescribed range. The event schema may look like (ID, Rating, Photo, ...), where each event reports the new rating and includes other relevant information, such as a picture for the product. The problem with such a stateless subscription is that new rating event for the same product would have to carry the bulky Photo (and other attributes) repeatedly, and must be delivered all the way to interested subscribers. If we represent each subscription as a select-join over two tables (ID, Rating) and (ID, Photo, ...), our techniques can bring two benefits:

- The use of binary semijoin reformulation directly eliminates result representation and current-content redundancies. In the example above, the Photo attribute would not be sent to a subscription if the product's previous rating was already within the subscription's range of interest.

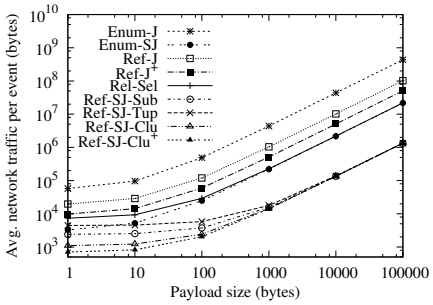

Figure 18: Network traffic; increasing payload size.

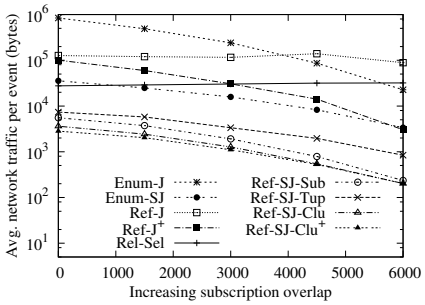

Figure 19: Network traffic; increasing subscription overlap.

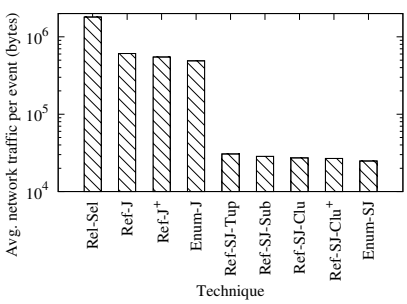

Figure 20: Network traffic; considering last hop.

- CN* can provide further benefits by reducing re-dissemination redundancy. In the example, assume that the Photo attribute has been previously delivered to some subscription X_1, and later needs to be delivered to some other subscription X_2. Let X_1 and X_2 share some common path in the overlay dissemination network. In this case, the Photo attribute would not be re-disseminated on the common path if it was retained by CN* in some payload repositories.

J Additional Experimental Results

In this section, we report additional experimental results for binary select-joins on unmodified CN (to augment Section 6.1).

Number of Overlay Hops When Varying Number of Subscriptions In terms of overlay hops (Figure 17), our techniques are at least an order of magnitude better. All Ref-SJ techniques use the same number of hops (they differ only in the size of the skyline). Subscription relaxation (Rel-Sel) does worse due to tuples being unnecessarily sent to overlay nodes. The enumeration techniques degrade with increasing number of subscriptions. Ref-J and Ref-J+ also incur a large number of hops.

Increasing Payload Size We increase the payload size (of both R and S tuples) from 1 byte to 100kB, and show the effect on network traffic in Figure 18. Note that the x-axis also uses a logarithmic scale. As payload size increases, all approaches incur additional traffic, but the absolute difference in performance is much larger for larger payloads. With increasing payload size, the differences between the various semijoin reformulations diminish because payload size dominates over the description.

Increasing Overlap among Subscriptions We increase the extent of overlap of subscriptions, by reducing the standard deviation of the distributions from which the $R.A$ and $S.C$ range centers are drawn (see Table 2). We set the standard deviation for $R.A$ and $S.C$ ranges to $13000 - 2x$ and $7500 - x$ respectively, and vary x from 0 to 6000. Figure 19 shows the network traffic. As we increase overlap, fewer subscriptions are affected by an update because of the concentration of interests in narrow regions. Ref-J is unaffected by overlap since it sends out joining tuples regardless of subscriptions. Ref-J+, which takes subscriptions into account, shows lower network traffic as the overlap increases, due to fewer affected subscriptions. Enum-J, Enum-SJ, and the Ref-SJ schemes also see reduced traffic with increasing overlap due to the same reason. Among the Ref-SJ approaches, the performance improvement is least for Ref-SJ-Tup since the descriptive skyline is independent of subscription clustering. Ref-SJ-Sub, Ref-SJ-Clu, and Ref-SJ-Clu+ converge in performance at high subscription overlap due to very high amount of clustering. Finally, Rel-Sel actually degrades in performance with increasing subscription overlap because it does not take the join into account, and more overlap (clustering) means that events that fall in the "hot" region of $R.A$ (which many events do) have to be sent to many subscriptions.

Considering Last Hop We have ignored the "last hop" from an overlay node to a subscriber because we have focused on the performance of the core publish/subscribe middleware, and the final delivery mechanism (e.g., unicast, email, IM, etc.) might be different for different clients. We now examine the effect of considering the last hop, assuming direct unicast from overlay nodes to clients. We assume that the same approach (join, semijoin, or relaxation) is applied until the end subscriber.[3] We see from Figure 20 that considering the last hop makes semijoin much more attractive than before. Enum-SJ incurs the lowest total traffic because it avoids the overlay network completely, at the cost of very high server stress. Ref-SJ-Clu+ incurs only slightly higher cost, with the important benefit of sharing dissemination using the overlay network.

Additional References

[26] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 2001.

[27] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002.

[28] R. Chand and P. A. Felber. A scalable protocol for content-based routing in overlay networks. In *NCA*, 2003.

[29] Y. Chawathe et al. A case study in building layered DHT applications. In *SIGCOMM*, 2005.

[30] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *WebDB*, 2004.

[31] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, 2001.

[32] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtree: A scalable distributed Rtree. In *ICDE*, 2007.

[33] G. Mühl. Generic constraints for content-based publish/subscribe. In *CoopIS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, 2001.

[34] L. Opyrchal et al. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[35] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

[36] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW*, 2002.

[3] Hybrid schemes where semijoin or relaxation is applied inside the overlay network, but precise join results are sent to subscribers, are also possible but are omitted for simplicity.