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 in a wide-area publish/subscribe system. Subscriptions are interested in joins over different data sources (tables), with varying interests expressed as range selection predicates 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 across events and subscribers. We develop and experimentally evaluate our scheme, and show that it 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, cell 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, conceptually represented by a table* `Stocks`(`Symbol`, `PER`, ...)*. In particular, the* `PER` *attribute records the price-to-earning ratio, a popular measure of stock quality. Another source of data is analyst reports, represented by* `Reviews`(`Symbol`, `Rating`, ...)*. In general, 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:

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

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

Challenges Continuing with Example 1, we further illustrate the challenges (and opportunities) that arise in supporting select-join subscriptions in a wide-area publish/subscribe system. 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` (e.g., if we track historical information in `Stocks`).

`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 select-join subscription query, in the same manner as *incremental view maintenance* [20]. 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 $s_4 r_2, s_4 r_3, \ldots, s_4 r_{20}$, which is the subset of $\{s_4\} \bowtie$ `Reviews` satisfying the selection conditions of X_1.

The first obvious problem with this approach is that of *large server output size*. The server is burdened by having to enumerate potentially many output tuples. This problem slows down processing and increases server load, and in turn reduces event-processing throughput and increases 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.

- ***Current-content redundancy***. This redundancy arises when some information in the notification message can be readily inferred from the current subscription content. In many application scenarios it is reasonable to assume that subscription clients in a publish/subscribe system maintain the current subscription content. In this case, information that can be inferred from this content does not need to be included in the notification message. The naive approach does not take advantage of this option. 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.

- ***Inter-event redundancy***: This redundancy arises when the 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 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 (e.g., if we maintain only a recent window of history in 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 as much as 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 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 a network of *brokers*, a popular approach in wide-area publish/subscribe systems. Brokers are each responsible for a subset of the subscriptions, and they forward events to each other based on downstream subscription interests. Each event traverses down through a tree of brokers 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, or the general re-dissemination redundancy.

Besides the challenges above, there are important practical considerations. We do not wish to over-complicate the design of broker networks, e.g., by augmenting brokers with hard application state or application-specific processing logic, or general database-like processing capabilities. If possible, we want to reuse common, off-the-shelf network substrates for dissemination, because on large scales, they are more robust and less likely to face deployment and maintenance hurdles than more complex networks.

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 broker network, we base our solution on a simple, well-established type of dissemination substrates called *content-driven networks* (CN) [9].

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 apply a novel *reformulation* scheme to eliminate inter-subscription redundancy and enable the use of CN. The reformulation can be computed efficiently at the server, 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 with different sets of joining tables and range selection conditions (Section 4). The approach is to decompose such subscriptions into two-way select-semijoins and group-process semijoins of the same form; we show how to choose a good decomposition using cost-based optimization. We also extend our schemes to support multi-attribute range selections (Section 6).
- 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*.

Finally, we note that some of our contributions have applications beyond handling join subscriptions. Interestingly, even for simple selection subscriptions, if events contain bulky payloads (e.g., a PNG image attribute in a `Stocks` event containing the company logo, or a text attribute in `Review` carrying the content of the analyst report), it would be more efficient to treat selections as select-joins and apply our techniques. We discuss this technique using an example in Section 6.

The techniques in Section 3.1 can be used for delivering results of continuous queries to remote clients, or maintaining views that are materialized remotely, such that communication is minimized. Compared with classic solutions, one advantage of our techniques is they are designed to scale to huge number of continuous select-join queries and views. The techniques in Sections 3.2 and Section 5 further improve efficiency if we use a broker network for communication.

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 between brokers. This extension is application-agnostic, avoids hard state at brokers, and can be deployed incrementally—i.e., CN* can coexist with CN, and we can gradually replace CN brokers in an existing broker network with CN* brokers, or simply add new CN* brokers to the mix.

2 Preliminaries

2.1 Publish/Subscribe Systems

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.

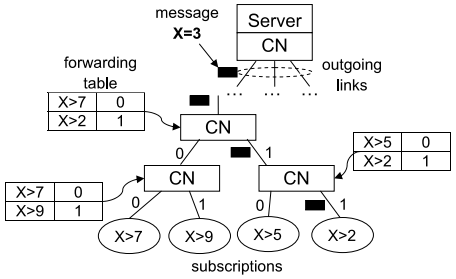

Figure 1: Using a content-based network for publish/subscribe.

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 work, however, we use a publish/subscribe model that supports more powerful database-style subscriptions. We assume that a central server maintain a database. Publishers of data send events to the server. Events are modifications (inserts, deletes, and updates) to the database. A subscription is a query Q over the database. On an incoming event, the database state changes 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 publish/subscribe 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.

To share the costs of notification delivery across subscriptions, we use a network of *brokers*, like most wide-area publish/subscribe systems (as discussed in Section 1). Each broker is assigned a subset of the subscriptions. The brokers collectively forward notification messages among themselves and to the subscribers. The last hop of notification delivery, from a broker to an affected subscription assigned to it, can use any delivery mechanism suitable to the subscription client, e.g., IP unicast, emails, or instant messages.

2.2 Content-Driven Networks

A popular technique for building the broker network is to use a *content-driven network* (CN) [1], a term that we introduced in [9] 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. In this work, we treat CN as a black-box delivery mechanism. Many structured as well as 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. Users can choose an appropriate CN, depending on convenience and the desired level of expressiveness (application-dependent).

Content-based Networks *Content-based networks* [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 well-established and popular systems that use content-based networking as the dissemination layer, e.g., SIENA [5], Gryphon [32], REBECA [31], Hermes [34], PADRES [19], JEDI [15], XNet [7], etc. Thus, content-based networks are easy to configure, deploy and maintain 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 typed attributes following some *schema*. For example, a message M following a schema that

[1] 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., [4, 5, 33]; we want to capture a broader class of systems with different designs and varying degrees of expressiveness.

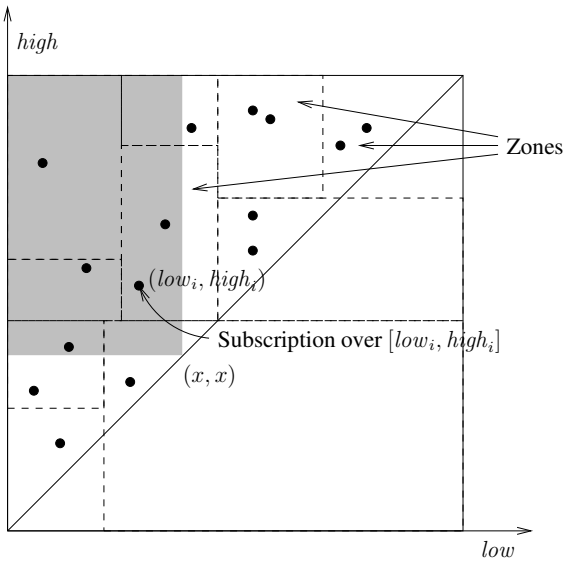

Figure 2: Using a CAN for publish/subscribe.

includes `Symbol` and `Price` as two of its attributes, might look like $\langle$ `Symbol`: '`GOOG`', `Price`: `550.00`, ... $\rangle$. Destinations, on the other hand, are simple predicates over the message attributes. For the example schema above, a destination may be a set of predicates such as `Symbol`$=$ '`GOOG`' $\wedge$ `Price`$\in$ `[500.00, 600.00]`. 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 to each destination, every message matching the predicates declared by that destination. Message delivery is performed in a multi-hop manner over an overlay network of nodes. The delivery function consists of two interrelated subfunctions: routing and forwarding. Routing amounts to establishing flow paths through the network by compiling and positioning local forwarding tables at each node. A forwarding table contains the information necessary for a node to decide to which neighbor node or nodes a given message should be sent; the processing of a message at a node is the forwarding subfunction. Taken together, the forwarding performed at the nodes causes messages to be routed through the network, until they reach all the affected destinations. Content-based networks use sophisticated techniques [5, 32] to build concise forwarding tables at the nodes, and to perform the forwarding function efficiently. Figure 1 shows the routing of a message using forwarding tables, for 4 subscriptions with predicates on an attribute X, installed in the system.

Content-Addressable Networks Another example of CN, which supports a more restrictive subscription language, is a *content-addressable network* (*CAN*) [36], where a message contains d numeric attributes and each destination node implicitly specifies an orthogonal range predicate in the d-dimensional space as its subscription. 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 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 subscriptions can specify range predicates over k attributes. We index each subscription in a $2k$-dimensional CAN as follows. Each attribute used in range selection is mapped to two dimensions in the

CAN space, one for the low end of the range and the other for the high end. A range subscription can then be represented as a point in this space. Figure 2 illustrates a 2-d CAN, where each single-attribute range subscription over the range $[\texttt{A\_LOW}, \texttt{A\_HIGH}]$ is mapped to the point $(\texttt{A\_LOW}, \texttt{A\_HIGH})$ in the CAN space. Note that in general, a subscription could be mapped to 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*—a network node responsible for all the subscriptions in its zone. Partitioning of the CAN space into zones can use load balancing criteria, for example, the number of subscriptions residing in the zone and the number of events reaching the zone. The zone owners correspond to brokers in the publish/subscribe system. In order to use the CAN-based CN for reaching affected subscriptions, we inject a message with an attached description into CN. The description can be interpreted as a *region* in CAN space that we wish to reach. 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* [21] is a publish/subscribe system that uses a CAN-based dissemination layer similar to the one just described, but only supports subscriptions with range predicates, with simple stateless matching of events to subscriptions matching all range predicates.

Expressiveness: If subscriptions are mapped to the CAN space based on their range predicates, a hypercube region in d-dimensional CAN space can succinctly 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 range attribute of a subscription. More complex region definitions can express arbitrary containment predicates succinctly.

For example, a CN message $\langle\, \texttt{UL\_X}\colon a, \texttt{UL\_Y}\colon b, \texttt{LR\_X}\colon c, \texttt{LR\_Y}\colon c \,\rangle$ might be interpreted as a rectangular region with upper-left coordinate (a, b) and lower-right coordinate (c, c), shown shaded in Figure 2. This description identifies every subscription whose range predicate (1) is contained within the range $[c, c]$, and (2) contains the range $[a, b]$. For comparison, the same subscription (over the range $[\texttt{A\_LOW}, \texttt{A\_HIGH}]$), in a content-based network, would be represented as the predicate $[\texttt{A\_LOW}, \texttt{A\_HIGH}] \subseteq [\texttt{UL\_X}, \texttt{UL\_Y}] \wedge [\texttt{LR\_X}, \texttt{LR\_Y}] \subseteq [\texttt{A\_LOW}, \texttt{A\_HIGH}]$.

Other CN instances Many dissemination mechanisms fall under the CN umbrella, including multicast networks (e.g., [6, 33]) and distributed indexes such as prefix hash trees [12], P-trees [14], BATON [26], SD-Rtrees [18], etc. However, these substrates have considerably lower expressiveness. For example, a multicast network supporting multiple multicast groups can be seen a 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. Distributed one-dimensional range search indexes (e.g., [12]) are CN, because we can regard a node responsible for data item s as interested in all range search messages satisfying the predicate $(S_L \leq s) \wedge (s \leq S_R)$, where S_L and S_R are the two message attributes corresponding to the left and right endpoints of the search range.

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

2.3 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. We assume that subscriptions return all attributes in the result (i.e., no projections), and there are no self-joins.

We will start with a simple scenario, formalized in the example below, where all subscriptions have the same two-way join signature and one single-attribute range selection for each table. In Section 4, we show how to extend our solution in Section 3 to a general mix of multi-way joins. The extension to multi-attribute range selections is covered in Section 6.

Example 2 (Two-Way 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* payloads*. 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 in this scenario.

2.4 Simple Solutions

We now present several simple solutions that can be used in existing systems to support select-join subscriptions. The parallels between these solutions and prior related work is elaborated in Section 8.

Enumeration of Direct 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 [13] 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 (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 are handled by CN. 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 (Ref-J and Ref-J$^+$) *Reformulation* [8] 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 predicate 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 predicate $\mathtt{A} \in I_i^A \wedge \mathtt{C} \in I_i^C$.

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.[2] Also, Ref-J does not avoid current-content, or more generally, inter-event and re-transmission redundancies.

Another inefficiency with Ref-J, caused by delayed evaluation of the selection conditions (in CN instead of at the server), is that Ref-J may send out 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 Two-Way Select-Joins

We now present our solutions for the scenario described in Example 2, where all subscriptions are two-way 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 will ignore the dissemination aspect, and focus on how to efficiently compute, at the server, the minimal amount of information to send to each subscription for every event. Then, in Section 3.2, we show how to retool the solution in Section 3.1 so that we can further leverage CN for efficient dissemination.

3.1 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:

$$X_i^R = \sigma_{A \in I_i^A} R \ltimes_B \sigma_{C \in I_i^C} S, \text{ and } X_i^S = \sigma_{C \in I_i^C} S \ltimes_B \sigma_{A \in I_i^A} R.$$

We call them *R-semijoin* and *S-semijoin* respectively. Semijoins are a well-known method in distributed

[2] Result representation redundancy could be overcome by requiring brokers to batch messages and compress outgoing message batches during message forwarding, but this requires modifying CN to add complex application-specific logic, which we want to avoid.

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 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 either 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 subscription 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 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 a t_R-affected R-semijoin does not imply the corresponding S-semijoin is t_R-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. However, we will see that there is a simple technique for finding such S-semijoins without having to compute or materialize their contents at the server. 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 are omitted for simplicity.

Once the changes have been computed, we assume that the server unicasts them to each subscription. We call this scheme as *Enum-SJ*. We will see how to exploit 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 t_R-affected R-semijoins. A number of techniques exist in the scalable continuous query processing literature, e.g. NiagaraCQ [13]. 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 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 t_R-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 substantially higher than the actual number of t_R-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 clusteredness 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.

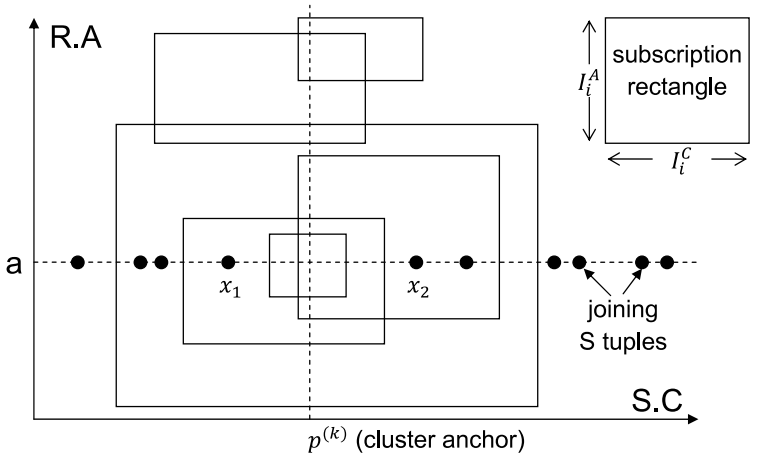

Figure 3: Processing Enum-SJ using SSI for cluster $\mathcal{X}^{(k)}$.

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 $R.C$. 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 have been developed in [1] to maintain the partitions such that τ is close to the minimum possible. Figure 3 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 3 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).

This procedure returns exactly the set of t_R-affected R-semijoins. 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 t_R-affected iff its rectangle contains at least one of these points. Therefore, any t_R-affected subscriptions 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 Let k_R be the number of t_R-affected R-semijoins. Let $g = O(\sqrt{\max_{1 \le k \le \tau} \left|\mathcal{X}^{(k)}\right|})$ 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. As it turns out, testing exposure is not as hard it seems.

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

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

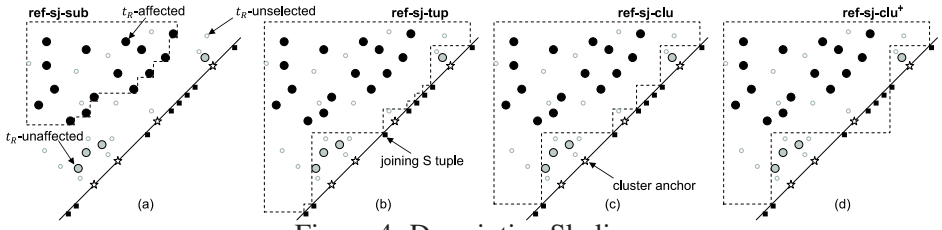

Figure 4: Descriptive Skylines

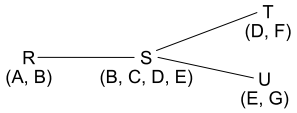

Figure 5: Example join graph.

- $a^+(t_R) = \min\{t_R.A \mid t \in R \wedge t.B = t_R.B \wedge t.A \geq t_R.A\}$, *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 $R(B, A)$. This lookup directly retrieves $a^-(t_R)$, while $a^+(t_R)$ is its immediate right neighbor in the B-tree. Next, we process each subscription cluster $\mathcal{X}^k$ in turn, in the same loop where we compute t_R-affected R-semijoins (Section 3.1.1). For each t_R-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 t_R-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 the 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 t_R-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 simple traverse the B-tree leaves towards left and right starting from this point, stopping when we encounter a tuple with $S.C \notin I_i^C$.

Complexity Let s be the number of S tuples that join with t_R. Recall that k_R denotes the number of t_R-affected R-semijoins. Let k_S ($\leq k_R$) denote the number of t_R-affected S-semijoins (i.e., those with at least one exposed joining S tuple), and let $\bar{s}'$ ($\leq s$) be the average number of joining S tuples exposed to each t_R-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 Scalable Dissemination of Select-Semijoins

Enum-SJ suffers from inter-subscription redundancy because it unicasts notifications to each subscription. In this section, we show how to leverage CN to share 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 t_R-affected R-semijoins efficiently. Can we characterize this set succinctly without enumerating its 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 4 (a).

Consider just the subset of R-semijoins whose I_i^A ranges contains a; we call these the t_R-*selected* R-semijoins. Obviously, every t_R-affected R-semijoin must first be t_R-selected. Among the t_R-selected R-semijoins, it turns out that the t_R-affected ones can be separated from the t_R-unaffected ones by a skyline, as formally 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* $x_i \leq x_j$ *and* $y_i \geq y_j$*. The* subspace covered *by this skyline is the union of northwest quadrants of all skyline points, i.e.,* $\{(x, y) \mid \exists i : x \leq x_i \wedge y \geq y_i\}$.

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 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 t_R-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 t_R-selected R-semijoin X_i^R is covered by the skyline, it must lie to the northwest of some t_R-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 local selection condition must also satisfy X_i^R's too. Hence, X_i^R must also be t_R-affected. 2) This property follows directly from the construction of P'. □

For example, Figure 4 (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 following message:

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

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

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

We call this general approach Ref-SJ, for reformulation with semijoins. Compared with the Enum-SJ 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, corresponding to different versions of Ref-SJ. We now examine several techniques for computing one. Several factors influence our choice: 1) we want to compute the skyline efficiently; 2) we want to reduce the number of skyline points, because it affects the size of the reformulated message; 3) we want to reduce the area of the subspace covered by the skyline, because a larger area may translate to slightly higher dissemination costs for some CN implementations (e.g., those based on CAN).

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 4 (b). While every easy to compute, the number of skyline points may be large and can even exceed k_R.

Cluster-Based Skyline (Ref-SJ-Clu) Here, 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 t_R-affected R-semijoins in this cluster is exactly the set of t_R-selected R-semijoins in this cluster that lie to the northwest of (x_1, x_1) or (x_2, x_2). The justification follows the same line of reasoning as the SSI-based algorithm in Section 3.1.1. Briefly, the other joining S tuples which were a part of the description in Ref-SJ-Tup are unnecessary. To see why, note that every t_R-affected R-semijoin is known to lie to the northwest of some cluster (by definition of a cluster), as well as to the northwest of some point in the joining-tuple skyline. Thus, every t_R-affected R-semijoin would be covered by (i.e., lie to the northwest of) at least one of (x_1, x_1) and (x_2, x_2).

When all clusters have been processed, the point set P specifies a descriptive skyline with up to 2τ points, as illustrated by Figure 4 (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 t_R-selected and t_R-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 4 (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 t_R-selected and t_R-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 $O(\tau)$ R-tree lookups. Specifically, we check whether a triangle is disposable as follows. 1) 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. 2) Suppose there are one more cluster anchors between q_j and q_{j+1}. For each cluster anchor $p^{(k)}$, we count n_1, the number of t_R-affected R-semijoins in cluster $\mathcal{X}^{(k)}$, by looking up (q_j, a) and (q_{j+1}, a) in the cluster R-tree, as in Section 3.1.1. We also count n_2, the number of t_R-selected R-semijoins in $\mathcal{X}^{(k)}$, by looking up $(p^{(k)}, a)$ in the cluster R-tree. It is not difficult to see that the triangle is disposable iff $n_1 = n_2$ for all cluster anchors between q_j and q_{j+1}. 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 find such skylines, but the running time would become $O(k_R^2)$. We leave a more

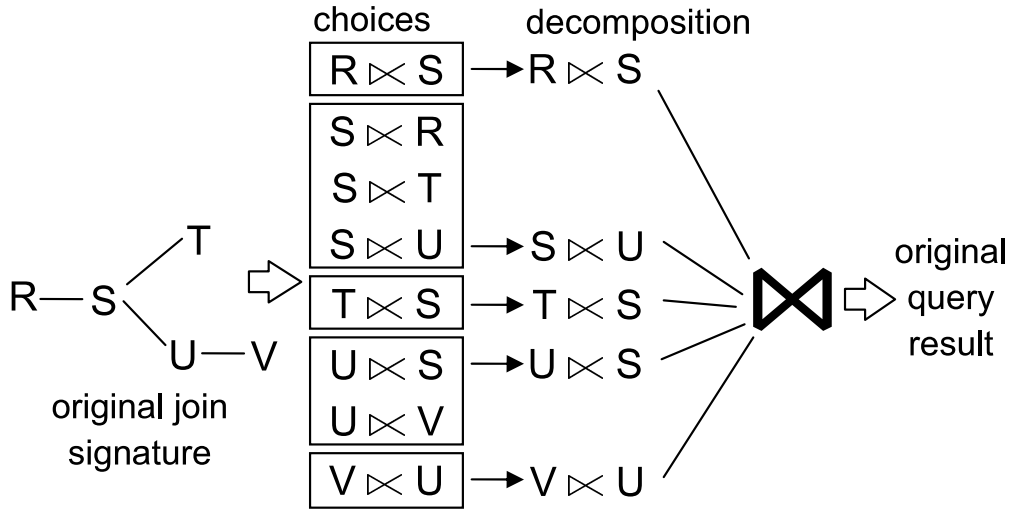

Figure 6: Processing multi-way join using decomposition.

detailed investigation of this alternative as future work, since 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 t_R-affected S-semijoins, by iterating through subscription clusters. For each cluster $\mathcal{X}^{(k)}$, we find the t_R-affected R-semijoins using two R-tree lookups, as in Section 3.1.1. For every t_R-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 following reformulated CN message:

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

Accordingly, an S-semijoin is reformulated simply as a predicate 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 exactly s', the number of S tuples exposed to at least one t_R-affected S-semijoin. The total message size is $O(s'h_S)$. The running time of the algorithm is $O(\log |R| + \tau(\log |S| + g) + k_R \log |k_S| + k_S \log |S|)$.

4 Multi-Way Select-Joins

In this section, we consider how to handle a general mix of select-join queries over multiple tables. Recall from Section 2 that we represent the schema as a join graph, and each select-join query has a join signature that corresponds to a connected subgraph of the join graph. The join graph has tables as nodes and join attributes as edges. For simplicity, we assume that each table has a single selection attribute, and defer the extension to multiple attributes to Section 6. Each table has as many join attributes as incident edges. We present our approach first, and then briefly discuss other alternatives and the associated tradeoffs.

4.1 Overview

An n-way select-join query over n input tables is decomposed into n two-way select-semijoins (one for each input table). Note that each two-way select-semijoin provides the content for one 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 predicates on R and S in the original subscription. We call this two-way select semijoin an R^S-semijoin. An R^S-semijoin is identical in definition to the R-semijoin introduced in the two-way case; the superscript S serves to disambiguate between the various possible neighboring tables in the join graph that R could join with. For example, in the join graph of Figure 6, a subscription over the entire join graph might be decomposed into 5 binary semijoins: R^S-semijoin, S^U-semijoin, T^S-semijoin, U^S-semijoin, and V^U-semijoin. Note that there are three choices for table S—S^R-semijoin, S^T-semijoin, and S^U-semijoin (one corresponding to each edge incident on S). Likewise, there are two choices for table U—U^S-semijoin and U^V-semijoin.

The client can reconstruct the original select-join using these binary-semijoins by simply performing an n-way natural join over the contents them. To see why, note that the client receives input table tuples that participate in some two-way select-join result. Since we know that every multi-way select-join result has an embedded two-way select-join result, the destination can use the semijoin decomposition to answer the original select-join subscription. Note that no local selection (filtering) is necessary at the client, because the binary semijoins already handle these predicates. The process of decomposing a join signature and reconstructing the original query result using the binary semijoins is shown in Figure 6.

One consequence of binary semijoin decomposition is that not every semi-join result is necessarily a part of the multi-way select-join result, which means that the query could receive some false positives. Nevertheless, in cases with many join results (which is where simpler methods suffer the most) the number of tuples passing the binary semijoin is usually much smaller. We consider the alternative of directly extending semijoins (without decomposition) to the multi-way case, at the end of this section.

4.2 Optimizing Decomposition

Each source table has to choose a neighboring table (in the join signature) to semijoin with. The optimal choice that minimizes the number of tuples that need to be sent to the client, depends on several factors. For some intuition, consider the cost of choosing R^S-semijoin. 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 upon 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" neighbor S that is expected to generate the smallest number of t_R-affected R^S-semijoins. 2) On insertion of a tuple t_S into table S, each of the joining R tuples need to be sent to all t_S-affected R^S-semijoins. The associated cost depends upon the probability of insertion into table t_S, the expected number of joining R tuples (since each joining R tuple may need to be sent out), 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 for each table is independent.

We cost each decomposition as follows. The total cost is the sum over all *binary semijoin groups*. A binary semijoin group (say R^S) contains only decomposed semijoins of the same form (R^S-semijoins). The cost of a binary semijoin 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 binary semijoin group: 1) The first approach is based on periodic simulation feedback. We use a random sample of subscriptions and

events, and simulate the processing and dissemination for each binary semijoin group, assuming that all possible R^S-semijoins have been assigned to this group. The output cost is divided by the number of R^S-semijoins to give a per-semijoin cost. This approach is general and can adapt to the actual subscription and event workload, but simulation requires more work. 2) The second approach is to use a cost model. We have developed a simple cost model (described next) that assumes a uniform distribution of subscriptions to keep complexity low, and costs each group based on parameters including the probability of insertion into each table and the expected number of joining tuples on an insertion.

We follow a greedy algorithm to find the decomposition with lowest overall cost. We repeat the following for each node R in the join graph. We find the neighbor S with lowest cost $c(R^S)$, and make all subscriptions containing R and S pick this semijoin for R. If there are unassigned subscriptions containing R, the process is repeated using the group $R^{S'}$ which has the next lowest per-semijoin cost. This is repeated until all subscriptions containing R have been assigned to some group. Given the nature of our cost model, it is easy to see that this greedy algorithm finds the optimal decomposition. In experiments, we find that optimization based on cost model gives good results in scenarios that we tested.

Optimality of Greedy The greedy decomposition gives the optimal choice under our cost model. We prove this 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.

4.3 Cost Model

We now describe a simple cost model for the select-semijoin $\sigma_{p_R} R \ltimes_B \sigma_{p_S} S$ between the two tables R and S, with the CAN-style CN introduced in Section 2. Assume that (1) subscriptions are uniformly distributed over their domain, in terms of their range selection predicates, and (2) local selection attribute values of events are uniformly distributed over their respective domains.

Assume that CN uses a balanced zone partitioning scheme that assigns approximately the same number of subscriptions to each zone. We can approximate the CAN space to be a uniform grid, restricted to the upper-left of the diagonal (called the *routing area*). Dissemination cost to a region depends on the number of zones covered during routing. Hence, the cost of dissemination to some region in CAN space is proportional to the fraction of the routing area covered by the region. In case of a two-way join, the CAN space is a four-dimensional space. Two dimensions correspond to the left and right endpoints of $R.A$ attribute ranges. The projection of the CAN space onto these two dimensions is called the *R-space*. The remaining two dimensions of the CAN space correspond to the left and right endpoints of $S.C$ attribute ranges, and this space is referred to as the S-space.

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.
- j_S: expected number of S tuples having the same join attribute.

The cost of the select-semijoin has two components.

Cost due to insertion into R On an insertion into R, the affected fraction of routing area in the R-space is expected to be $1/3$, which we next show. Consider the CAN space to be a square with diagonal of unit length. If the insertion into R has $R.A = x$, the routing area is the portion of the space to the north-west of point (x, x) in the CAN space. This routing area is a rectangle with sides $\frac{x}{\sqrt{2}}$ and $(\frac{1}{\sqrt{2}} - \frac{x}{\sqrt{2}})$. Note that the total routing area is simply the upper-triangle of area $1/4$. Since x can lie equally anywhere between 0 and 1, the expected affected fraction is:

$$\int_0^1 4\frac{x}{\sqrt{2}}\left(\frac{1}{\sqrt{2}} - \frac{x}{\sqrt{2}}\right)dx = \frac{1}{3}$$

Next, in S-space, the affected routing area lies to the northwest of the j_S points. With a uniform distribution of joining S tuples along the diagonal, the affected fraction of the routing area in S-space is expected to be $\frac{j_S}{j_S+1}$. Since the insertion into R occurs with probability p_R, we can approximate the total insertion cost c_R as:

$$c_R \propto p_R \times \frac{1}{3} \times \frac{j_S}{j_S + 1}$$

Cost due to insertion into S On an insertion into S, we need to send j_R messages, one for each joining tuple. For a message corresponding to some joining R tuple with $R.A = a$, the affected fraction of routing area in the R-space is expected to be $1/3$ as before, since the point a could lie anywhere along the diagonal with equal probability, and the affected routing area lies to the northwest of the point a. In S-space, the affected routing area lies to the northwest of (c, c) and southeast of (c^-, c^+), where $S.C = c$ and c^- and c^+ are the immediate neighbors to the left and right of c. Since after insertion there are j_S points uniformly distributed along the unit length diagonal, the distance from c to c^- or c^+ is expected to be $\frac{1}{j_S}$. Thus, the affected area of routing area in R-space is expected to be $\frac{1}{2j_S^2}$, implying that the affected fraction of routing area in S-space is expected to be $\frac{2}{j_S^2}$, since the total routing area is again $\frac{1}{4}$. Since the insertion into S occurs with probability p_S, we can approximate the total insertion cost c_S as:

$$c_S \propto p_S \times j_R \times \frac{1}{3} \times \frac{2}{j_S^2}$$

Total cost The total cost of R^S-semijoin is the weighted sum of the above costs, i.e., total cost $c(R^S) = \alpha c_R + \beta c_S$, where α and β are constants.

4.4 Alternative Approaches

First, we could use select-join relaxation (Rel-Sel) to convert each query into a single-table select. All subscriptions selecting over some table are processed and disseminated together. This scheme may suffer from many unnecessary notifications due to relaxation. Second, we could use simple reformulation (Ref-J) to disseminate all join result tuples for each join signature. But, the output size problem is exacerbated because the join result can be quite large if many tuples satisfy the predicates. Also, join results need to be generated and output separately for each join signature. However, if the join is very selective, this may be a viable solution. Moreover, using CN* (discussed in Section 5) for routing can help overcome the re-dissemination redundancy.

4.5 Directly Extending Two-way Joins [3]

If we assume that the join signature graphs are acyclic, we can directly extend the reformulation procedure for two-way select-joins and directly consider longer semijoins instead of binary semijoins. We can group-process all join queries having the same join signature (graph $\mathcal{S}$). For each table R, let $\mathcal{Q} = \mathcal{S} - \{R\}$ be the remainder of the join signature excluding node R. We define $R^{\mathcal{Q}}$-semijoin as the semijoin $R \ltimes (\mathcal{Q})$ (selects are not shown). The reformulation of $R^{\mathcal{Q}}$-semijoin is described next. We first define a *partial join result*.

Definition 2 (Partial Join Result)**.** *The partial join result of a connected subgraph $\mathcal{G}$ of a query graph is the result of the natural join computed only over $\mathcal{G}$ with no selection predicates applied, and projecting only the selection predicate attributes. Thus, the partial join result of a subgraph with n nodes can be visualized as a set of points in an n-dimensional space.*

Insertion into R On the insertion of a tuple t_R into table R, the reformulation for $R^{\mathcal{Q}}$-semijoin consists of a conjunction of k predicates, one for the subgraph on each outgoing edge from R. The predicate for each such subgraph is similar to that derived for R-semijoins in the multi-attribute selection case (see Section 6.3), with the difference that the descriptive skyline in this case is computed with respect to the partial join result tuples of that subgraph, which join with t_R.

Insertion into S Table S represents any of the other tables in the join signature. We first determine which tuples of R are exposed to some subscription as a result of the insertion of t_S into table S. For each exposed R tuple, say t_R, we send t_R with the following reformulation. As before, we have a conjunction of predicates, one for the query subgraph on each outgoing edge from R. For the subgraphs that do not contain S, the predicate is exactly as described in the previous paragraph. For the subgraph containing S, we need to ensure that we are describing only those subscriptions that have never previously received t_R due to some other joining partial join result tuple. We first compute the cluster-based descriptive skyline $\mathcal{D}$ over all the *new* partial join result tuples, i.e., partial join result tuples that contain t_S and join with t_R. For each point in $\mathcal{D}$, we compute the skyline envelope of that point with respect to the *old* partial join result tuples that join with t_R. The predicate for this query subgraph is that the subscription should contain at least one point in $\mathcal{D}$ while not containing any of the corresponding skyline envelope points.

Discussion While the direct extension is most precise, the higher dimensionality creates practical issues: (1) there are more attributes in the message, and the predicates over the attributes are more complex. (2) it is difficult to process events efficiently at the server because high-dimensional indexes are not as efficient. In contrast, binary semijoins lose some filtering power of multi-way joins, but are simple to implement and efficient in practice. This is similar to the use of low-dimensional index structures in databases to process queries involving predicates in more dimensions. Another advantage of binary semijoin decomposition is that we do not have to group-process the queries in each join signature separately. Instead, we can group-process all queries choosing a particular binary semijoin.

5 Reducing Re-Dissemination Redundancy

The techniques outlined in Sections 3 and 4 avoid result representation, current-content, and inter-subscription redundancies which cover a very significant portion of network cost. However, they still cannot 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, there is no sharing of content across

[3] Due to its complexity, the reader can skip the details of the extension and directly read the discussion at the end of the subsection. We do not suggest using this direct extension, but include it for completeness.

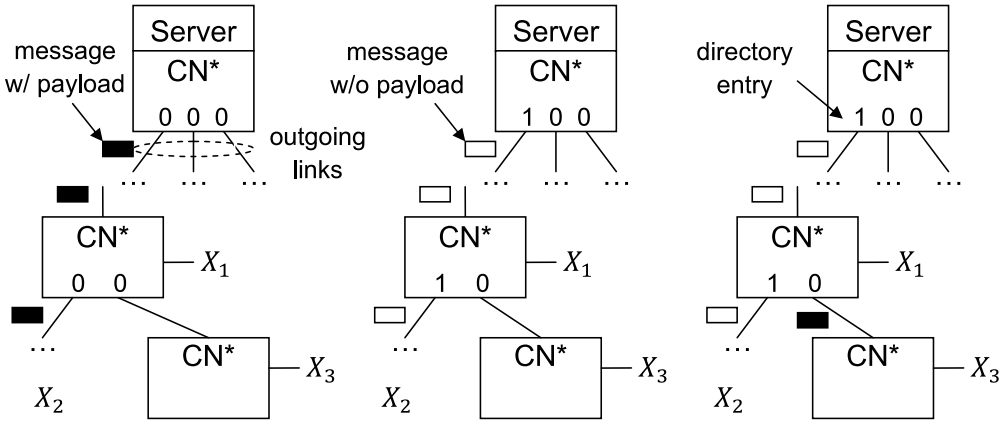

Figure 7: Payload dissemination using CN*.

events, because the updates delivered to a broker are not available for sharing in future. 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; we will see an example in Section 6. Hence, we opt for a general, CN-based solution that attempts to avoid retransmitting the same payload through the same link. We aim to maintain the clean interface of CN, and avoid introducing hard state or complex processing at brokers.

One straightforward solution to the problem is to use caching. The idea is to convert each payload into an ID reference, and push the messages without payload to the subscription. Upon receiving the message, the subscription generates a request to the source over an overlay tree that implements caching, so as to serve future requests before reaching the server. However, this scheme fundamentally changes the push-style dissemination of publish/subscribe to pull, which may not be good enough for some applications. Moreover, the scheme can add considerable latency and new payloads can cause a high amount of additional pull traffic.

Solution Overview 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. Assume that each payload can be identified by an ID specified by the server (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 can avoid re-dissemination as follows: if a payload has been sent over an outgoing link, we send a lighter version of the message without payload, otherwise the message is sent with payload. The broker responsible for a subscription adds the payload to the message (if necessary) before finally forwarding it to the subscription. We call this approach *CN**.

Consider the example in Figure 7. On the first incoming event, subscriptions X_1 and X_2, both interested in this event, receive the same payload through the broker network (Figure 7 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), the messages can be sent with the payload (Figure 7 center). Finally, if another event causes a third subscription X_3 to need the same payload, we would transmit the payload only along the links that never previously transmitted it; other links would transmit without the payload (Figure 7 right).

One important feature of this extension, which we call CN*, is that both directory and repository are soft state. We cannot assume that each CN* node remembers the entire history. Soft state also aids in failure-handling since we do not have to recover any state related to the payload. However, this means that we must

cope with cases where the directory and/or repository entries are deleted. We next describe the details of our approach.

Operational Details The standard CN message (with attribute-value pairs) is augmented with a CN* part indicating which attributes collectively form the Payload, an ID identifying the payload (or a pointer to a standard CN attribute that can serve as ID), and a Source, which records the closest encountered CN* node along the dissemination path, which has a copy of Payload (initially, Source is set to the server).

The payload directory is organized as entries of the form [ID, Bitmap]. Bitmap is a bitmap with one bit for each outgoing link, that indicates whether the payload has been previously sent along that link. The payload repository is organized as entries of the form [ID, Payload], and stores the payload content. A B-tree indexed on ID is used for quick access to a particular entry.

Assume for now that entries in the directory and repository are never purged. During dissemination, the server injects a message m as usual into CN*. The CN* router checks the directory to see if the corresponding entry is present. If not, it adds a new entry with Bitmap set to all zeros. It also adds m.Payload, if available in m, to the payload repository. Link matching is then done just as in regular CN, to determine the set of matching outgoing links. For each matching link, if the payload was previously sent over that link (indicated by a 1 in the bitmap entry), the router sends the message without the payload. Otherwise, the bit is set to 1 and the message is sent with the payload. Every CN* node along the path performs similar checks. A CN* node that has the payload in its repository sets m.Source to its own address before forwarding m.

Directory and Repository Maintenance To limit space usage, CN* can periodically purge entries from the repository and/or directory. The choice of which entry to purge could be based, for example, on a least-recently-used (LRU) scheme. Further, if a directory entry is 1 for all outgoing links, it is usually acceptable to purge the entry from the repository. We next discuss the purging of repository and directory entries separately.

Handling Purged Repository Entries: Assume that a message m corresponding to a purged repository entry arrives at node n. An inconsistency occurs if the payload needs to be sent along some outgoing link, but is not present in the repository or the message. Under this scenario, n contacts m.Source to get the payload (if m.Source dropped this entry in the interim period, n can always contact the server as a fallback mechanism). This scheme incurs no more payload traffic (one hop) than directly carrying the payload from m.Source. The increase in notification latency for this subtree is one roundtrip between n and m.Source.

Handling Purged Directory Entries: The directory occupies very little space (a 512kB directory can hold tens of thousands of entries), hence it may need to purge entries very infrequently. When n receives a message corresponding to a purged directory entry, it simply processes the message assuming that all bitmap entries are 0. Thus, depending on the frequency of purging directory entries, CN* may rarely send a payload multiple times along a link.

Forwarding Algorithm Algorithm 1 shows the CN* forwarding procedure. In line 3, R-QUERY retrieves the corresponding directory entry (a new entry is created if necessary). If the payload is present in the incoming message m, R-QUERY also adds the payload to the repository. R-QUERY may also update access statistics depending on the purging scheme is use. Line 4 identifies the matching interfaces are identified just as in CN. In lines 7—12, we check the bitmap field for each affected interface. If the payload was previously sent over that interface, it is not sent again. Otherwise, GETPAYLOAD retrieves the payload from either the repository entry, the closest provider source, or (in the worst case) the server. The retrieved payload is added

to the message before disseminating. If the repository caches the payload, line 13 updates the Source field in the outgoing message to the machine's network address. Finally, the message is disseminated along the outgoing link (line 14).

Algorithm 1: Forwarding algorithm for CN*.

```
1  FORWARD(message m) begin
2      p ← ∅ ;                                              // placeholder for payload
3      X ← R-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
               // payload already sent previously
8              m'.Payload ← ∅;
9          else
10             if m'.Payload = ∅ then
                   // add payload to message
11                 if p = ∅ then p = GETPAYLOAD(X, m'.Source);
12                 m'.Payload ← p;
           // update source address
13         if X.Payload ≠ ∅ then m'.Source ← local address;
14         DISSEMINATE(m', i) ;                             // send message along interface i
15 end
```

Co-existence with Regular CN By design, CN* nodes can co-exist with regular CN nodes. This facilitates incremental deployment and adoption. Regular CN nodes simply ignore the additional attributes in the message. Our algorithms operate correctly in the presence of CN nodes. If a CN node delivers a message m with empty payload to the broker application (for example, because an upstream CN* node assumed that the payload was previously sent), it can simply contact m.Source to retrieve the payload. Another alternative is for CN* to always send the payload if the downstream node is CN.

We have developed an efficient technique for CN* to detect that a next hop neighbor is CNṪo achieve this, when a payload is sent along an link i, the corresponding bit in the directory entry is not immediately set. Instead, it is set only if this node receives a special acknowledgment from the next hop (indicating that it is a CN* router). Thus, in case the next hop is a plain CN router, the bit would never get set and the payload would always gets sent along that link. The acknowledgment does not lie on the critical path from the server to the destination, and thus does not impact notification latency.

Comparison with Caching 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, as discussed earlier in this section, 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 Discussion and Extensions

6.1 Select with Payload

A stateless selection subscription, supported by traditional publish/subscribe systems, may sometimes benefit from being expressed as a join. For example, consider an event schema of store products Items(ID, Rating, Photo). Subscriptions may be interested in all the information for different ranges of rating. The problem is that every rating insertion for the same product would have to carry the Photo attribute repeatedly, and deliver it to all subscriptions stabbed by the new rating. If we represent each subscription as a select-join over two tables (ID, Rating) and (ID, Photo), our techniques give benefits at two levels:

- Semijoin reformulation itself benefits select with payload, by eliminating current-content redundancy. This happens automatically due to the reformulation scheme, and does not have to be treated as a special case. In the example above, the Photo attribute would be sent to a subscription only when the subscription is exposed to a particulat stock for the first time (due to a joining Rating within its range of interest). Subsequent Rating tuples would not be disseminated to such a subscription.

- 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 broker network. In this case, the Photo attribute would not be re-disseminated on the common path if it were retained by CN* in the node repositories.

6.2 Multi-Attribute Join Conditions

Consider the join between tables R and S, sharing d join attributes $(B_1, \ldots, B_d)$. Handling multiple join attributes is easy, as we can conceptually treat the set of joining attributes as a single composite attribute. The only change to our algorithms is to use B-trees indexing the composite key $(B_1, \ldots, B_d, A)$ and $(B_1, \ldots, B_d, C)$, instead of (B, A) and (B, C) respectively.

6.3 Multi-Attribute Selection Conditions

Assume that R and S have d_R and d_S selection attributes respectively. On insertion of an R tuple t_R, we can extend our semijoin reformulation techniques as follows. The joining S tuples can be mapped as a set of points ($\mathcal{J}_\mathcal{S}$) in a d_S-dimensional space. A subscription is a hypercube with $d_R + d_S$ dimensions, one for each selection attribute range.

Handling R-Semijoins The reformulation scheme is the same as before, but each point in the descriptive skyline has $2d_S$ dimensions. Thus, computing the descriptive skyline is more complicated. The joining-tuple skyline (see Section 3.2) is directly derived from $\mathcal{J}_\mathcal{S}$ — each point $(c_1, \ldots, c_{d_S})$ in $\mathcal{J}_\mathcal{S}$ is converted into a skyline point 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. We first define a *skyline envelope*.

Definition 3 (Skyline Envelope)**.** *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 minimal hypercube spanning both points z and y.*

For each cluster of subscriptions that are stabbed by the same point p in this space, we only need to add the skyline envelope of p with respect to the the joining S tuples in this space, i.e., Sky$(\mathcal{J}_\mathcal{S}, p)$, to the

Method	Delivery	Technique	Comment
Select	CN/CN*	Rel-Sel	Inject events in CN/CN* w/o joining (Sec. 2.4)
Select-Join	Unicast	Enum-J	Compute join results for each sub (Sec. 2.4)
	CN/CN*	Ref-J, -J$^+$	Inject join results in CN/CN* (Sec.2.4)
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.

cluster-based skyline (again, converting coordinates to empty ranges).

Handling S-Semijoins For each joining S tuple t_S, we consider the set of previously joining R tuples ($\mathcal{J}_\mathcal{R}$) as points in a d_R-dimensional space. We can compute the skyline envelope of t_S with respect to the points in $\mathcal{J}_\mathcal{R}$, i.e., $\mathsf{Sky}(\mathcal{J}_\mathcal{R}, t_S)$. t_S then needs to reach every subscription that, apart from satisfying the selection conditions on t_S and t_R, is not stabbed by any of the envelope points. The latter condition ensures that we do not reach subscriptions that are t_R-unaffected (i.e., have t_S in their view due to joining with some other selected R tuple).

Discussion While the extension described above is precise, it may be slightly less efficient in practice due to the lack of efficient high-dimensional indexing and skyline computation structures and the lower degree of user-interest clustering expected at higher dimensions. Similar to the techniques adopted in traditional databases, one solution to tackle the problem of higher dimensions is to choose a single selection attribute (per table) for group-processing and group-disseminating. The remaining selection attributes are applied by subscriptions as a post-processing step. The tradeoffs in choosing the best selection attribute in this case are similar to that of choosing the best binary semijoin decomposition in the multi-way join case (we need to choose the overall most selective and effective filtering attribute for each table). We leave a detailed analysis of this alternative as future work.

7 Evaluation

7.1 Setup, Metrics, and Workload

Server Setup At the server, we implemented all our novel schemes of Enum-SJ, Ref-SJ-Tup, Ref-SJ-Clu, Ref-SJ-Clu$^+$, and Ref-SJ-Sub. For comparison, we also implemented Enum-J, Rel-Sel, Ref-J, and Ref-J$^+$. Refer to Table 1 for a summary of solutions. 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 with a write speed of ˜70 Mbps, which is roughly similar to a dedicated OC1 optical (˜52 Mbps) connection to the Internet. Our data structures use main memory use main memory for optimal performance. The experiments were 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 be analyzed using our link-level simulator which uses a 20,000 node topology produced by INET [11], a generator of Internet-like network topologies. A subset of 1000 nodes act as brokers. In this paper, we focus only on measurements between overlay nodes because we have observed that IP-level costs generally follow similar trends as node-level costs.

The vanilla CN we chose to implement is a CAN [36]-based overlay network that uses a semantic routing space, with subscriptions mapped as points in the space based on their interests. The semantic space is divided into *zones* based on load balancing criteria, with one broker assigned to each zone. Each zone

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.

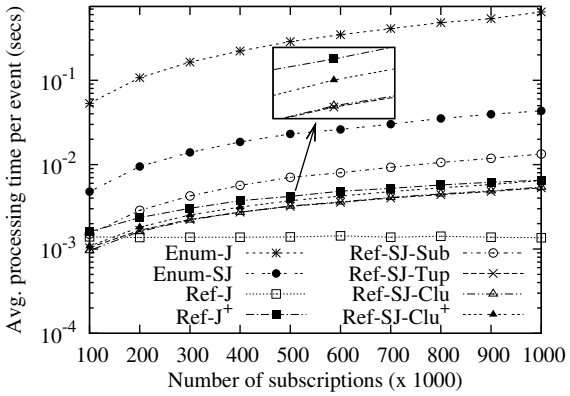

Figure 8: Processing time; increasing number of subscriptions.

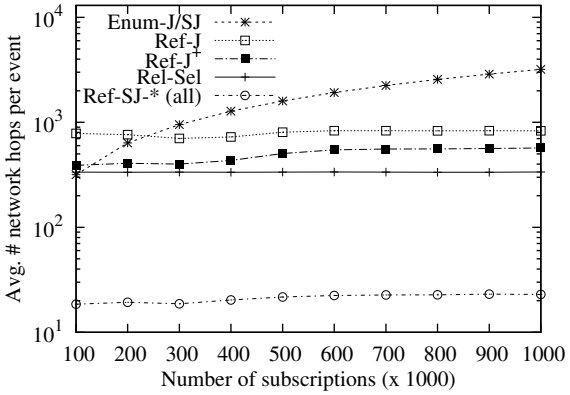

Figure 9: Network hops; increasing number of subscriptions.

# 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).

is responsible for a region of the semantic space, and subscriptions in that region are assigned to the corresponding broker. Zones have knowledge of only their neighbors, and routing proceeds in a multi-hop manner until the region of interest described by the injected message is covered. A similar CN has been used in several other publish/subscribe systems, e.g., [21, 8]. We also implement and report results using our CN* extension. Each CN* node maintains a directory and repository following the LRU replacement

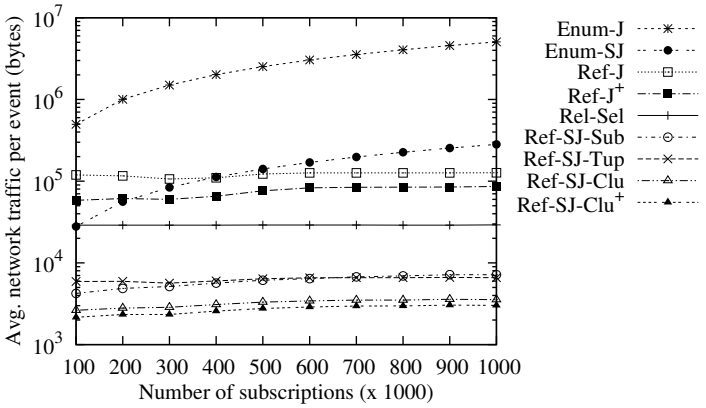

Figure 10: Network traffic; increasing number of subscriptions.

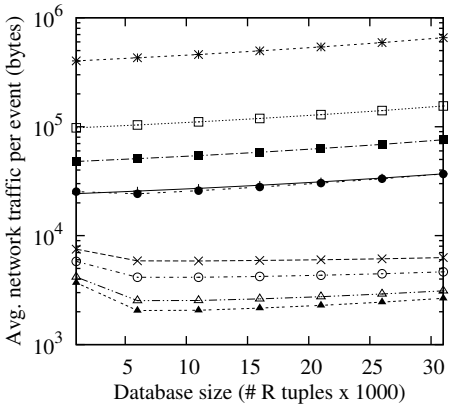

Figure 11: Network traffic; increasing database size.

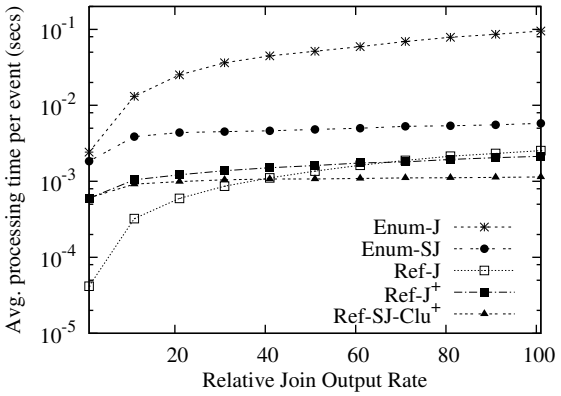

Figure 12: Processing time; increasing relative join output rate (RJOR).

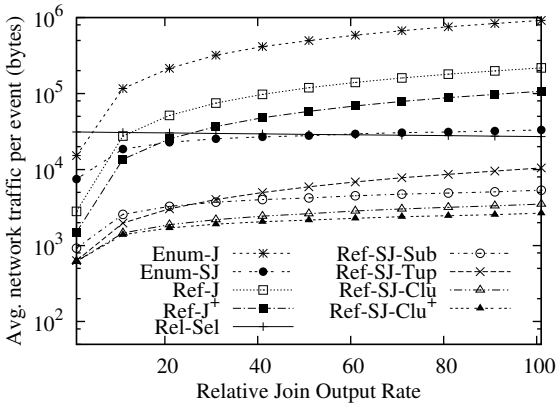

Figure 13: Network traffic; increasing RJOR.

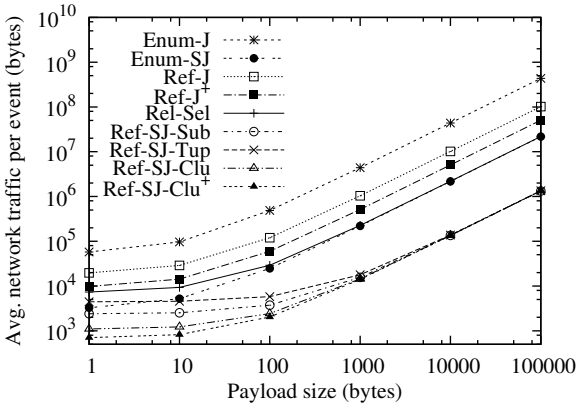

Figure 14: Network traffic; increasing payload size.

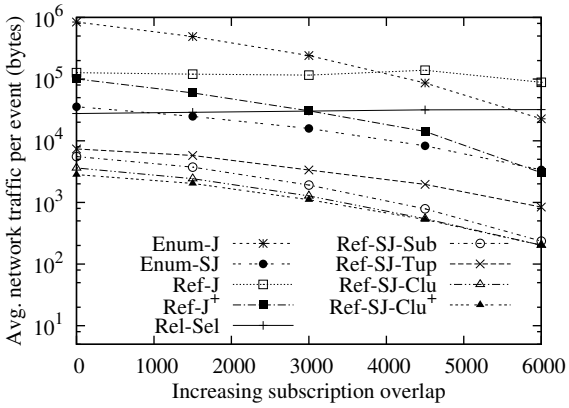

Figure 15: Network traffic; increasing subscription overlap.

policy.

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 bytes transferred between overlay nodes. 2) *Number of overlay message hops* per event, which measures the total number of messages sent between nodes. 3) *Node stress* per event, which measures load on a node. In this paper, we report byte stress, which denotes the total traffic originating from a node. 4) *Hop latency*, which measures the number of overlay hops for an event update to completely reach a subscription. Hop latency roughly corresponds to subscription notification latency, assuming uniform network delays between nodes.

Workload For two-way select-joins $R \bowtie S$ (see Example 1), 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 the summary of parameters. Each subscription uses normal distributions distributions N_1 and N_2 to generate the centers of ranges over $R.A$ and $S.C$ respectively. 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

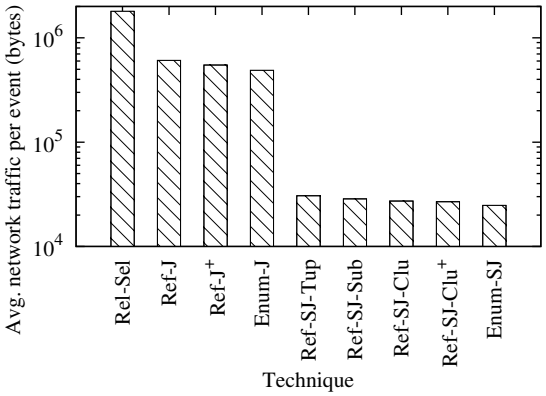

Figure 16: Network traffic; considering last hop.

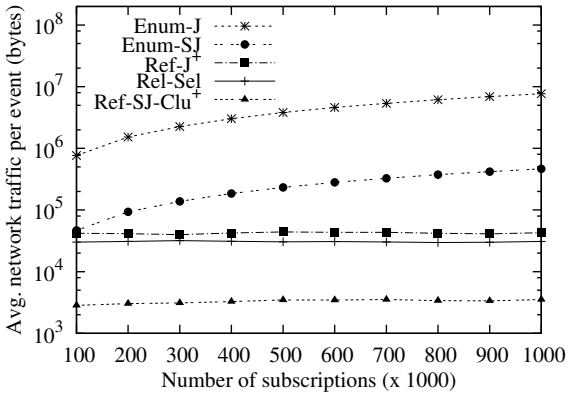

Figure 17: Network traffic; real event workload.

distribution with parameter 0.8. R tuples are inserted for each unique join attribute value, and 70% of R tuple insertions produce at least one join result [4]. 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 [38] (Section 7.2 has for details). Finally, the workloads for our general mix of n-way join subscriptions are described in Section 7.4.

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.

7.2 Two-Way Select-Joins, Unmodified CN

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

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 in all the results.

The results for server processing time are shown in Figure 8. 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 sends minimal data for a particular subscription, but it still suffers from inter-subscription redundancy. The simple reformulations (Ref-J and Ref-J$^+$) perform better, 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 very efficient. 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$^+$ introduce very less overhead over Ref-SJ-Tup.

In terms of overlay message hops (Figure 9), our techniques are at least an order of magnitude better. All semijoin reformulation techniques use the same number of hops (they differ only in the size of the skyline).

[4] This parameter was derived after examining our real stock event workload for the fraction of stocks having at least one rating.

Subscription relaxation (Rel-Sel) does worse due to tuples being unnecessarily sent to brokers. The server-based techniques degrade linearly with increasing number of subscriptions. Ref-J and Ref-J$^+$ also incur a large number of hops.

Network traffic (Figure 10 for Enum-J is extremely high as expected. Enum-SJ is much better, but degrades quickly with number of subscriptions. The simple reformulations (Ref-J and Ref-J$^+$) are better due to sharing of costs over CN, but they also 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 semijoin reformulations avoid unnecessary dissemination while sharing costs across subscriptions. Ref-SJ-Clu$^+$ incurs the lowest traffic overall, at least an order of magnitude lower than simpler schemes. Ref-SJ-Clu$^+$ also gives the best skyline compression (see Table 4), while Ref-SJ-Sub degrades faster with number of subscriptions. Finally, server stress (see Table 3) is lowest for CN based approaches as they enable sharing of dissemination. Ref-SJ-Clu$^+$ is the best as expected.

Varying Database Size We now see the effect of increasing the number of R tuples in the database (older tuples are deleted when new ones are inserted, to keep the table size constant). Figure 11 shows the average network traffic. All schemes have a slightly increasing trend because a larger R table implies fewer deletes, which are cheaper to disseminate. 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, we are still able to easily outperform other approaches.

Varying Relative Join Output Rate Relative join output rate (RJOR) is the number of join result tuples generated for each inserted event. We control RJOR by varying the number of tuples in table S. Figure 12 shows the server processing cost for increasing RJOR. We see that Ref-J is very good at low RJOR, but quickly degrades due to output size. Ref-SJ-Clu$^+$ is scales well with increasing RJOR. Figure 13 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 simpler approaches.

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 14. 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 in 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 of Subscriptions We keep the number of subscriptions constant at 100k and increase the amount of overlap of subscriptions by reducing the standard deviation of the distribution from which subscription range centers (for $R.A$ and $S.C$) are drawn. We set the standard deviation of $R.A$ to $13000-2x$ and that of $S.C$ to $7500 - x$, where x is varied from 0 to 6000. We plot the performance of various approaches in terms of network traffic, with increasing overlap of subscriptions (increasing x), in Figure 15. As we increase the overlap, fewer subscriptions are affected by an update because of the concentration of interests in narrow regions of space. Ref-J is unaffected by overlap since it sends joining tuples regardless of subscriptions. On the other hand, Ref-J$^+$, which takes subscriptions into account, shows lower network

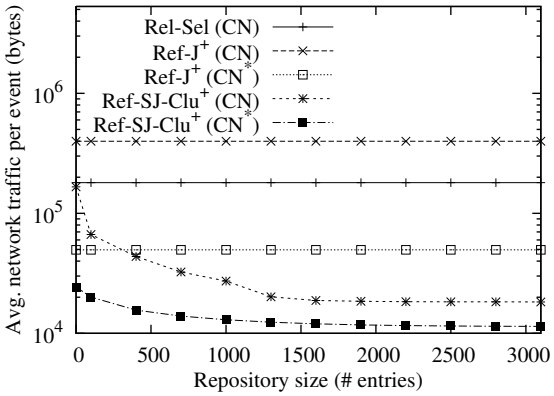

Figure 18: Network traffic; increasing repository size.

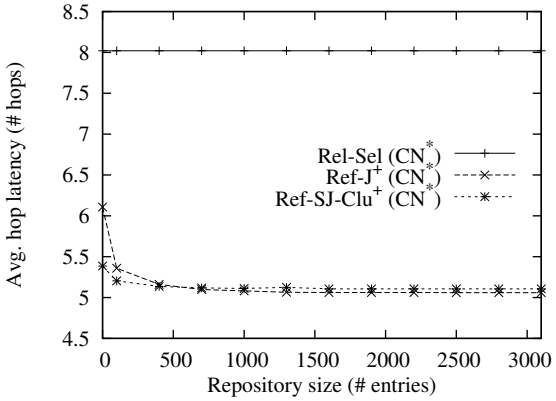

Figure 19: Hop latency; increasing repository size.

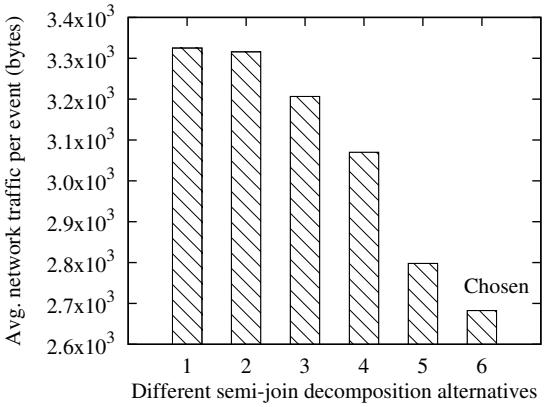

Figure 20: Network traffic; multi-way join mix.

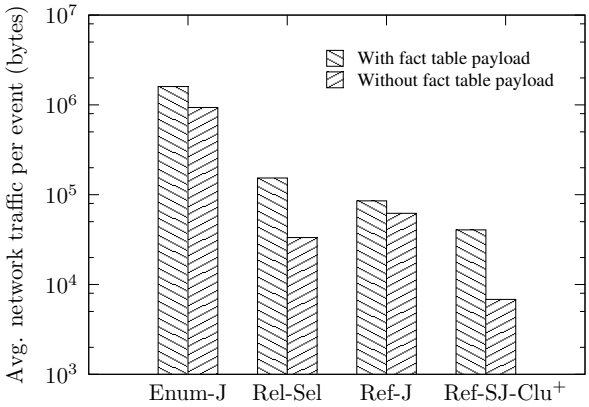

Figure 21: Network traffic; star schema mix.

traffic as the overlap increases, due to fewer affected subscriptions. Enum-J, Enum-SJ, and the semijoin reformulation schemes also see a reduced traffic with increasing overlap due to the same reason. Among reformulation-based approaches, the performance improvement is least for Ref-SJ-Tup since the descriptive skyline is computed 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 lots of subscriptions.

Considering Last Hop We have ignored the "last hop" from broker to 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 brokers to clients. We assume that the same approach (join, semijoin, or relaxation) is applied until the end subscriber.[5] We see from Figure 16 that considering the last hop makes semijoin much more attractive than before. Enum-SJ incurs the lowest total traffic because it avoids the network of brokers 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 broker network.

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

Results of Real Workload We gather real data from Yahoo! Finance [38] 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 $[0, 100\text{k}]$ 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 17 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). Ref-SJ-Clu$^+$ is an order of magnitude better than all other schemes.

7.3 Results of Adding CN*

We now consider the additional benefits derived by using CN* which can reduce re-dissemination redundancy, thus improving performance. We only show Ref-SJ-Clu$^+$, and Ref-J$^+$ (with and without CN*) since these were the best reformulation-based approaches. We also show Rel-Sel for comparison (Enum-J and Enum-SJ do not use CN*).

Effect on Traffic We set the payload directory to be $512kB$, and vary the repository from 0 to 3100 entries. Both R and S tuples carry payloads of 1000 bytes. The R table size is kept very low (500 entries) to expose the worst case performance of Ref-SJ-Clu$^+$. Figure 18 shows the network traffic. We see that CN* is able to avoid re-dissemination redundancy for Ref-SJ-Clu$^+$ and Ref-J$^+$, even at low repository sizes. Ref-J$^+$ benefits more since it disseminates more unnecessary data, leaving a greater scope for CN* to reduce costs. Ref-SJ-Clu$^+$ is better overall. Note that even with 0 repository size, the directory reduces cost by having only affected brokers pull data from the server. Finally, we found that a repository of just 400 entries can reduce server stress by 7 times for Ref-J$^+$ and 3 times for Ref-SJ-Clu$^+$ compared to using CN.

Effect on Hop Latency Figure 19 shows the average hop latency across all subscriptions and events. R table size is very low (200 entries) to further 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 much higher hop depth, since a message needs to reach many more subscribers dispersed over a larger number of brokers.

7.4 Multi-Way Select-Joins

We use the join graph in Figure 5, and 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 selection predicates). The subscriptions and events are derived using normal distributions similar to the case of two-way joins. We experiment with two event schema.

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 were found to be prohibitively expensive due to the large number of join results for the multi-way joins. Rel-Sel was found to generate 23kB traffic per event, around 9 times worse than the optimal Ref-SJ-Clu$^+$ semijoin decomposition. In Figure 20, we compare the costs of different semijoin decompositions (without CN*).Our cost model orders the semijoin group costs as: $c(S^R) > c(S^U) > c(S^T)$. The greedy assignment of query groups is optimal in this case, and gives the lowest traffic. Some other random assignments are also shown. The optimal decomposition is around 20% better than the worst decomposition.

Star Schema We experiment with the popular star schema. Here, there are 3 dimension tables (R, T, and U) with one fact table (S). The query graph is the same as before. We model 1000 entries in each

dimension table, and 25000 entries in the fact table. The dimension tables each have a payload of 100 bytes. The star schema is an extreme case where each insertion into the fact table produces exactly one join result. Thus, RJOR is very low, and Ref-J can do well. Figure 21 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 necessary. Ref-J has to send out complete join results. The advantage is lesser 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). Nevertheless, we find that Ref-SJ-Clu$^+$ outperforms other techniques (even without CN*).

8 Related Work

Continuous Query Systems Continuous query systems (e.g., [29, 13, 16]) can be regarded as a form of publish/subscribe, where continuous queries over streams correspond to our subscriptions. NiagaraCQ [13] supports select-join processing at a server. CACQ [30] group-processes filters, and supports dynamic reordering of joins and filters. PSoup [10] 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 [22] 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., [17, 27, 8, 9, 19].). *SMILE* [27] supports SQL queries, while *PADRES* [19] 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 [8, 9], we have used reformulation to support complex queries over CN. However, [8] focuses on range aggregation, while [9] tackles subscriptions with value-based notification conditions.

Distributed Joins Distributed join processing systems, where state is distributed across overlay nodes, correspond to Rel-Sel if selects are applied first. PIER [23] supports SQL queries (including joins) over DHTs, but targets one-time queries and does not optimize for multiple subscriptions. Idreos et al. [25] 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, 37] to reduce communication in distributed databases. Work on view maintenance (e.g., [28, 24, 35]) 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.

9 Conclusions

A publish/subscribe system needs to optimize both subscription processing and result dissemination, particularly for complex queries such as joins. In this paper, we aimed to develop an end-to-end solution to support a large mix of multi-way select-join subscriptions. Towards this goal, we identified several key redundancies in traditional techniques of supporting such subscriptions. Our novel semijoin-based reformulation schemes reduce these redundancies and outperform standard techniques by orders of magnitude. The schemes are easy to deploy and maintain, yet ensure very high efficiency. We also proposed a new extension (CN^*) to content-driven networks, that further reduces redundancy of disseminating bulky payloads. Extensive experiments on real and synthetic workloads with two-way and multi-way select-joins validated the benefit of our schemes, and showed orders of magnitude improvement compared to 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*, 6(4):602–625, 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] 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.

[6] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002. URL `citeseer.ist.psu.edu/castro02scribe.html`.

[7] R. Chand and P. A. Felber. A scalable protocol for content-based routing in overlay networks. In *NCA '03: Proceedings of the Second IEEE International Symposium on Network Computing and Applications*, page 123, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1938-5.

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

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

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

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

[12] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *SIGCOMM*, 2005.

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

[14] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB*, pages 25–30, 2004.

[15] G. Cugola, E. D. Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the

development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.950318.

[16] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.

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

[18] C. du Mouza, W. Litwin, and P. Rigaux. Sd-rtree: A scalable distributed rtree. In *ICDE*, pages 296–305. IEEE, 2007.

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

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

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

[22] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.

[23] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.

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

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

[26] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

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

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

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

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

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

[32] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

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

[34] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1588-6.

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

[36] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.

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

[38] Yahoo! Finance. http://finance.yahoo.com.