On the Database/Network Interface in Large-Scale Publish/Subscribe Systems

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

Abstract

The work performed by a publish/subscribe system can conceptually be divided into subscription processing and notification dissemination. Traditionally, research in the database and networking community has focused on these aspects in isolation. The interface between the database server and the network is an aspect often overlooked by previous research. At one extreme, database servers are directly responsible for notifying individual subscribers; at the other extreme, updates are injected directly into the network, and the network is solely responsible for processing subscriptions and forwarding notifications. These extremes are unsuitable for complex and stateful subscription queries. A primary goal of this paper is to explore the design space between the two extremes, and to devise and present solutions that incorporate both database-side and network-side considerations in order to reduce the communication and server load and maintain scalability of the system. We focus on a broad range of stateful query groups, and present solutions for several of them. Our detailed experiments based on real and synthetic workloads with varying characteristics and link-level network simulation show that by exploiting the query semantics and building an appropriate interface between the database and the network, it is possible to achieve orders of magnitude savings in network traffic at low server-side processing cost for several classes of stateful queries.

1 Introduction

The advent of the Digital Age has enabled data collection on an unprecedented scale. An important problem is how to disseminate relevant information efficiently to users over a network. Users can *poll* data sources for information; however, polling too frequently may be inefficient, while polling less often may miss important updates. The alternative, supported by *publish/subscribe* systems, is to *push* updates to users according to their interests, which are expressed using *subscriptions*. The push model is better suited for ensuring timely update delivery required by many applications involving data monitoring. Examples of such applications abound in various domains: personal (e.g., news alerts and Internet auctions), financial (e.g., trading stock and commodity in real time), security (e.g., distributing critical software patches), military (e.g., disseminating command and surveillance information), etc.

At the conceptual level, the bulk of the work performed by a publish/subscribe system can be roughly divided into two components: (1) *subscription processing*, the task of matching and processing each incoming publish message with the large set of active subscriptions, and (2) *notification dissemination*, the task of notifying, over a network, those subscribers who are interested in the publish message. Previous work from the database research community has focused on efficient subscription processing; notification dissemination is rarely addressed. Most existing work assumes that a server maintains the entire database state and all subscriptions in the system, and is responsible for computing the set of subscribers affected by each incoming publish message. A straightforward way to notify this set of subscribers is to unicast a notification to each of them in turn. When many subscribers need to be notified, this approach will incur a large amount of outbound traffic from the server, and may easily overwhelm the server and its network links. As server-side subscription processing techniques (such as sharing [11] and indexing [16]) continue to mature, the dissemination bottleneck has surfaced in many systems, both research [12] and commercial [17]. Recently, the database community has made some initial efforts [12, 24] in addressing this problem (further discussed in Section 6), but much research is still needed.

On the other hand, the networking research community has always focused on efficient notification dissemination. Notable mechanisms include *multicast* [2] and *content-based networking* [7]. With multicast, the system defines a number of *multicast groups*, each consisting of a set of subscribers, e.g., those who are interested in Google's stock. The network can efficiently disseminate the same message to all members of the group. With content-based networking, the system views each message as a tuple of attribute-value pairs; e.g., attributes in a stock update message may include `SYMBOL`, `RISK`, `PRICE`, `EARNING`, etc. Each subscription is defined as a predicate over the message tuple; e.g., `SYMBOL` $=$ `'GOOG'`, or `RISK` $\in [20, 60]$ (between moderately low to medium risk). The network is typically implemented by an *overlay* of nodes that perform application-level routing. Each overlay node maintains a summary of all subscriptions reachable from each of its outgoing overlay links, and it forwards an incoming message onto an outgoing link if the message matches the corresponding summary. Both mechanisms, however, support only *stateless* subscriptions, i.e., those that can be processed by examining the message itself. For multicast, a message's group id encodes its forwarding directions. For content-based networking, the message tuple contains all the information needed to forward the message.

Traditional publish/subscribe systems have simple subscription languages that support only stateless subscriptions. In many situations, however, users may want updates to be further transformed, correlated, and/or aggregated. For example, with a range-aggregate subscription, a user can track the minimum `PER` (price-to-earning ratio, a popular measure of stock quality) of stocks within a risk range. This subscription is *stateful*, because just by looking at a stock update message, the system cannot always tell whether or how the message would affect the subscription. To meet the needs of these users, we are developing a wide-area publish/subscribe system that supports complex subscription definitions. The examples below discuss the range-min subscription in detail. Through the discussion, we illustrate some of the challenges in supporting such subscriptions, and preview several possible implementation approaches.

Example 1 (Range-min subscriptions) *Consider a publish/subscribe system that monitors the stock market for a large number of traders over a wide-area network. Conceptually, the system provides a database view `STOCK`(`SYMBOL`, `RISK`, `PER`, ...) that continuously tracks the up-to-date information for each stock. Suppose that a user is interested in tracking the minimum `PER` of stocks within a risk range $[x_1, x_2]$ that she is comfortable with. She can define a subscription over `STOCK` using a SQL query: `SELECT MIN(PER) FROM STOCK WHERE` x_1 `<= RISK AND RISK <=` x_2. To simplify discussion, let us focus on updates of `PER`, and assume that each update message has the schema $\langle$`SYMBOL`, `RISK`, `PER`, ...$\rangle$, where `PER` is the new price-to-earning ratio after the update. When such a message arrives, the system needs to notify those users whose subscription query results are affected by the `PER` update.*

The range-min subscription is stateful. To illustrate, consider the current state of `STOCK` shown as a collection of points (labeled by t_i and shown in solid black) in Figure 1; the X-axis plots `RISK`, while

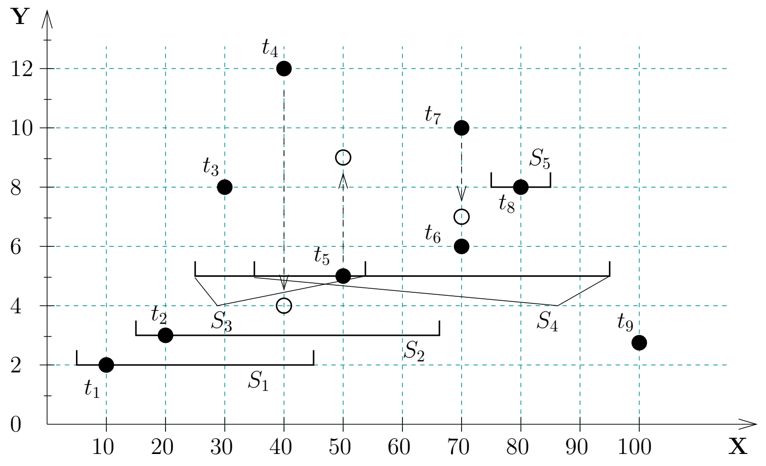


Figure 1: Example `STOCK` table and range-min subscriptions.

the Y-axis plots `PER`. Each range-min subscription (labeled by s_i) is represented as a horizontal interval spanning the risk range of interest, whose height equals the minimum `PER` in that range.

Suppose that an update lowers t_4's `PER` to just below that of t_5 (indicated by a dotted line with arrow). This update should affect subscriptions s_3 and s_4, but not s_1, s_2, or s_5. For s_1 through s_4, their ranges all cover t_4's `RISK`. In order to determine that s_3 and s_4 are affected while s_1 and s_2 are not, the system must be able to compare t_4's new `PER` with the minimum `PER` currently maintained by each of these subscriptions; this latter information is not available in the update message.

A more complicated situation arises when the current minimum `PER` shared by a group of subscriptions is updated higher, potentially exposing them to different new minima. For example, suppose that an update raises t_5's `PER`, as illustrated in Figure 1. As a result of this update, s_3 should be updated with t_3's `PER`, while s_4 should be updated with t_6's `PER`. Neither piece of information is available in t_5's update message. In general, the system must maintain the entire state of the `STOCK` table in order to handle such updates.

Example 2 (Supporting range-min subscriptions)
Following the traditional database-centric approach, we can use a server to maintain all subscriptions and the up-to-date state of the `STOCK` table. Thus, for each incoming stock update message, this server can easily compute which subscriptions are affected and how they need to be updated. However, options for disseminating these notifications are limited. (1) Unicast is the most natural way, but can be inefficient for a large number of affected subscriptions. (2) Content-based networking is difficult to leverage because of an "impedance mismatch": A content-based network performs matching between messages and subscriptions in the network, while in this database-centric approach the server has already computed the list of affected subscriptions; converting this list back to a message for dissemination in a content-based network is not straightforward, and would be a waste of resources because of duplicate processing. (3) Multicast is a possibility, but to do a perfect job, we would need a multicast group for every possible subset of the subscriptions that could be affected by a stock update in the same way. There may be a prohibitively large number of such groups (up to 2^m if every subset from a total of m subscriptions can form a group), rendering multicast infeasible. Even if we restrict the problem to range-min subscriptions alone, it is unclear how to reduce the space of possible groups. For example, in Figure 1, although s_2's risk range contains that of s_3, not every update affecting s_3 would affect s_2, and vice versa.

An alternative is to follow a network-centric approach. Content-based networking is a natural start-

ing point because it supports range subscriptions. We can "relax" a range-min subscription `SELECT MIN(PER) FROM STOCK WHERE` x_1 `<= RISK AND RISK <=` x_2 to a range subscription `SELECT * FROM STOCK WHERE` x_1 `<= RISK AND RISK <=` x_2. The network would then forward to each subscriber every stock update message that falls within her risk range. Each subscriber locally maintains the content of the range subscription from which the range-min subscription can be derived. Note that all stocks in the range must be maintained (not just ones with the minimum `PER`) in order to handle the case when the minimum `PER` rises. Besides this maintenance overhead, a more serious issue is that the relaxation of a stateful subscription into a stateless one can potentially result in much more update traffic. For example, in Figure 1, any `PER` movement of t_4 *above* t_5*'s `PER` has no effect on any subscriptions, but with this approach, all updates of* t_4 *would still be forwarded to* s_1 *through* s_4 *simply because* t_4 *falls into their risk ranges.*

The above examples show that efficient support of stateful subscriptions is a challenging problem for wide-area publish/subscribe systems. On one hand, existing network dissemination mechanisms do not support stateful subscriptions directly. While it is possible to relax a stateful subscription into a stateless one and rely on subscribers to perform additional local post-processing, doing so requires unnecessarily large amounts of local subscription state and high volumes of notifications. On the other hand, while the database-centric approach can easily process stateful subscriptions at a server, disseminating notifications over a wide-area network remains difficult because of the inefficiency of unicasts and the difficulty in interfacing the server with advanced network dissemination mechanisms such as multicast and content-based networking.

In this paper, we argue that the key to the solution lies in properly *interfacing* the database with the network, in order to combine the processing power of database servers and the dissemination power of the network effectively. In general, there is a wide spectrum of possibilities for interfacing the database with the network and for dividing up work between them. These possibilities provide an interesting set of trade-offs in terms of efficiency, scalability, and manageability of the system. To the best of our knowledge, there is no prior work that investigates this spectrum of database/network interaction models comprehensively. This unified perspective from both databases and networking enables us to identify interesting hybrid solutions that outperform approaches that are either database-centric or network-centric. Specifically, we make the following contributions:

- We explore a number of points along the spectrum of possibilities for interfacing database processing and network dissemination, and study their trade-offs. We show that efficient support of stateless subscriptions in a wide-area publish/subscribe system calls for hybrid solutions with novel database/network interfaces. We demonstrate through experiments with synthetic and real stock datasets that our hybrid solutions offer orders-of-magnitude performance improvement over approaches that are either database-centric or network-centric.

- We formalize *message and subscription reformulation* as a general mechanism for implementing stateful subscriptions using a dissemination network that supports only stateless subscriptions. Reformulation allows us to keep a simple and clean interface between the database and the network, while at the same time providing a comparable or higher level of efficiency compared with much more complex system configurations that require application-specific extensions to routing. We have developed reformulation techniques for a number of stateful subscriptions types including range aggregation/`DISTINCT` and joins.

- For range-min subscriptions, our reformulation technique is based on the concept of Mar (*Maximum Affected Range*), for which we have also developed new data structures and group processing algorithms. These techniques are also applicable in processing a group of continuous range-min/max queries, which is an interesting problem in its own right.

To recap, the combination of (1) cooperative processing and dissemination by the database and the network, (2) a clean, easy-to-implement database/network interface, and (3) efficient server-side data structures and algorithms together provide an efficient platform for supporting stateful subscriptions over a wide-area network.

The remainder of the technical report is organized as follows. Section 2 discusses various methods for interfacing the database with the network, including database-centric, network-centric, and hybrid approaches. Message/subscription reformulation and Mar are introduced in the context of the hybrid approach (Section 2.4). Section 3 presents efficient server-side indexing and processing techniques in support of the approaches in Section 2. We concentrate on describing how to support range-min subscriptions in this paper, and briefly discuss other stateful subscriptions in Section 4. Section 5 covers additional details of system implementation and presents experimental results. Section 6 surveys related work, and Section 7 concludes the report and discusses several directions for future research.

2 Spectrum of Server/Network Interfaces

This section explores the spectrum of possibilities for interfacing servers with a network in order to support stateful subscriptions efficiently. We start with a brief discussion of the database-centric approach in Section 2.2. Then, Section 2.3 discusses the network-centric approach, together with some background on content-based networking and intuition behind how updates affect subscriptions, which are useful also in later discussions. Section 2.4 describes one of the main results of this paper—a hybrid approach that supports closer cooperation between servers and the network using message/subscription reformulation.

2.1 Preliminaries

The publish/subscribe system we are building offers a conceptual (and possibly virtual) *database* over which users can define *subscriptions* as SQL views. *Publish messages* are updates to the database. If a database update causes the content of a subscription view to change, we say that the database update (publish message) *affects* the subscription; in this case, the system needs to send the subscriber a *notification message* containing the change to the content of the subscription view.

To keep our discussion focused, we concentrate in this section on supporting range-min subscriptions. These subscriptions are useful in many situations where users are interesting in tracking the "best" objects in ranges of their interest, e.g., stocks with the lowest price-to-earning ratios within a risk range, or lowest-priced digital cameras with at least 4.0 megapixels. The various server/network interface approaches and the message/subscription reformulation mechanism that we are going to present later are completely general; however, the actual reformulation technique may vary for different subscription types. We discuss how to handle other subscription types in Section 4.

Before proceeding, we give a classification of database updates based on how they affect range-min subscriptions. To make our discussion concrete, recall from Example 1 the database table

`STOCK(SYMBOL,RISK,PER,...)`

and range-min subscriptions of the form

```
SELECT MIN(PER) FROM STOCK WHERE x1 <= RISK AND RISK <= x2.
```

We call `RISK` the *range attribute* and `PER` the *aggregation attribute*. To simplify discussion in this section, we further restrict ourselves to updates of the aggregation attribute (`PER`) only. Insertions, deletions, and updates to other attributes require fairly straightforward extensions, details of which we defer until Section 3. Let $\Delta(t : x, y_o \to y_n)$ denote an update of a stock t (with risk x) that changes `PER` from y_o to y_n. This update falls into one of the following categories:

- *Ignorable updates.* These are updates that, given the current state of the database, cannot possibly affect any subscriptions. In our running example, $\Delta(t : x, y_o \to y_n)$ is ignorable if there exists another stock t' with the same risk x and a `PER` no higher than both y_o and y_n. For example, the update of t_7 in Figure 1 is ignorable because of t_6.
- *Non-ignorable updates.* These are updates that may affect some subscription (i.e. they are not ignorable). They are further classified into two types:
  - *Bad updates.* These are non-ignorable updates whose effects on affected subscriptions cannot be determined from the content of the update itself; additional information from the database is required. In our running example, a non-ignorable update $\Delta(t : x, y_o \to y_n)$ is bad if it might "expose" a new minimum `PER` in some risk range. The update of t_5 in Figure 1 is an example of a bad update because it exposes both t_3 and t_6. The effect of a bad update cannot be inferred from the content of the update alone; additional information from the database is required.
  - *Good updates.* A good update is a non-ignorable update that is not bad, i.e., its effect on any affected subscription can be determined from the content of the update itself. In our running example, a non-ignorable update $\Delta(t : x, y_o \to y_n)$ is good if no other minimum `PER` is exposed due to that update. The update of t_4 in Figure 1 is an example of a good update.

To recap, a decreasing update can be ignorable or good, whereas an increasing update can be ignorable, bad, or occasionally good. Note that this classification scheme does not take into account what subscriptions are currently in the system. Such information can be exploited for more efficiency (e.g., if there are no subscriptions, all updates are effectively ignorable), but doing so also incurs some extra overhead; we discuss this point further in Section 2.4.1.

2.2 Database-Centric Approaches

In this set of approaches, we follow the traditional database-centric view of publish/subscribe—of first computing the updates to each subscription, and then disseminating these updates. We assume that a single server maintains the database state and keeps track of all subscriptions. For each publish message, we can efficiently compute all subscription updates in time sublinear in the size of the database and the number of subscriptions, using our group-processing techniques (presented in Section 3). The approaches below differ mainly in how subscription updates are disseminated.

2.2.1 S-UN: Server with Unicast Network

With this approach, which we call S-UN, the server unicasts a subscription update message to each affected subscription. For our running example, the message has a constant size, and simply contains the new minimum `PER` for the subscription. The problem with this approach is that when many subscriptions are affected, there will be a large amount of traffic overall, and the server can easily become a bottleneck of dissemination.

If multiple affected subscriptions are hosted by the same node in the network, an additional optimization is for the server to combine multiple messages into one. This technique, which we call *message aggregation*, reduces the number of messages. However, the size of a combined massage would no longer be constant; instead, it becomes linear in the number of affected subscriptions at the node. The reason is that the message needs to list the affected subscriptions (because not all subscriptions at the node may be affected) and possibly multiple subscription updates (because different subscriptions may be affected differently by the same database update, as shown in Example 1).

2.2.2 S-MN: Server with Multicast Network

Multicast [2] is an efficient mechanism for disseminating messages to a group of network destinations. Ideally, we would define a multicast group for each subset of the subscriptions. After the server computes all subscription updates, it checks to see which subset of the affected subscriptions share the same update message, and sends out this message to the multicast group consisting precisely of these subscriptions. However, both IP multicast [2] and application-level multicast [8] techniques do not handle the need for large number of groups (up to 2^M for M subscriptions).

In this paper we resort to *hierarchical application-level multicast*. This method builds a tree rooted at the server spanning all nodes hosting subscriptions, with a moderate fan-out c. Each non-leaf node and its children together form a multicast network with 2^c multicast groups, each supporting efficient application-level multicast from the node to a subset of its children. Thus, this method avoids the problem of having too many multicast groups by breaking down the dissemination task into a hierarchy of much smaller multicasts with group number capped at 2^c. The cost of doing so is that the update message sent out by the server must list the set of affected subscriptions; otherwise, a non-leaf node would not be able to tell which children to forward the message to.

We call the above approach S-MN. The problems with using multicast for publish/subscribe (large number of groups and large message size) have also been identified by other work [4, 22, 24]. Possible solutions are: (1) Reduce number of groups [22] by approximating group membership in which case post-processing and additional state are needed at subscribers. We have considered some of these techniques, but the details are beyond the scope of this paper. (2) Use compact, "semantic" descriptions of affected subscriptions [4, 24] to avoid large update messages. This approach gives a content-based network (that still does not handle stateful subscriptions), which we consider next.

2.3 Network-Centric Approaches

At the other end of the spectrum, we have approaches that avoid the use of servers altogether by making the network handle as much subscription processing as possible. As discussed in Section 1, a natural starting point is a content-based network [7], which supports stateless subscriptions defined as predicates over the content of each message. Information about subscriptions is reflected in the distributed routing state of the

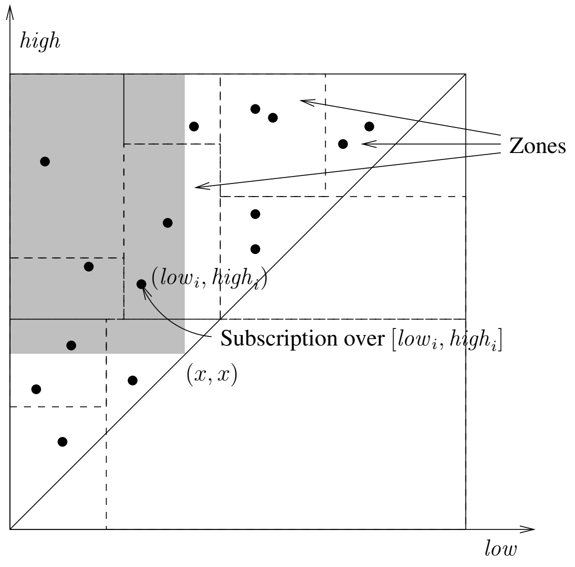


Figure 2: Space of range subscriptions.

network, which allows an update to be forwarded to affected subscriptions without intervention of servers. We now explore how to extend basic content-based networking to support stateful subscriptions.

Background on Content-Based Networking Content-based networks can be implemented using various techniques. Since the section focuses on range and range-min subscriptions, we discuss an implementation based on a *Content Addressable Network* (*CAN*) [25] similar to *Meghdoot* [15]. Meghdoot is designed to support only range subscriptions; we later demonstrate how to extend it in different ways to support range-min subscriptions. At the heart of Meghdoot is a CAN constructed as follows. Each attribute used in range selection (e.g., `RISK` in our running example) 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 $[low_i, high_i]$ is mapped to the point $(low_i, high_i)$. The space is partitioned into rectangular *zones*, each with a *zone owner*—an network node responsible for all the subscriptions in its zone. Each zone has knowledge of only its neighbors and can route messages only to them. Routing is carried out in multiple hops until the destination is reached.

Meghdoot notifies affected range subscriptions as follows. Consider an update to a tuple whose range attribute value is x. The affected range subscriptions are precisely those in the upper-left quadrant rooted at the point (x, x) (shown as a shaded region in Figure 2). Subscriptions inside this quadrant are affected because their ranges contain x; subscriptions outside this quadrant are unaffected because their ranges do not contain x. Hence, Meghdoot first routes the update message towards the point (x, x) which is called the *event point*; the zone owner of that point then forwards this message to the neighboring zone owners in the affected quadrant, which in turn forward the message upward and leftward to their neighbors, and so on. CAN provides a convenient substrate to implement this forwarding algorithm. Further details of this algorithm can be found in [15].

2.3.1 CN: Serverless Content-Based Network

The straightforward way to support stateful subscriptions using a content-based network, as already discussed in Example 2, is to "relax" them into stateless subscriptions directly supported by the network. We call this approach CN. For our running example, Meghdoot can be used to support the stateless `RISK`-range subscriptions relaxed from the stateful min-`PER`-over-`RISK`-range subscriptions. Each subscriber locally maintains the content of the stateless subscription queries, and uses post-processing to derive updates to the stateful subscriptions.

The advantage of CN is its simplicity: We only need to extend the capability of the subscribers; the network substrate remains unchanged. The system does not require a central server, thereby removing a potential bottleneck. The disadvantages of CN are obvious too. All updates within the `RISK` range are sent to the subscriber, even though most of them may be ignorable in practice. Also, to cope with bad updates, each subscriber must maintain all stocks within its `RISK` range, which is rather costly.

2.3.2 CN$^+$: CN with Additional Routing Logic

We can improve the efficiency of CN by exploiting additional information in the database state maintained for each subscription. The key observation regarding range-min subscriptions is the following (recall the notation from the beginning of Section 2):

> **(Subsumption property)** *If an aggregate attribute update $\Delta(t : x, y_o \rightarrow y_n)$ does not affect a range-min subscription with range $[x_1, x_2] \ni x$, then the update cannot affect any range-min subscription with range $[x'_1, x'_2] \supseteq [x_1, x_2]$.*

This observation allows us to cut off forwarding of update messages early: As soon as we hit an unaffected subscription corresponding to point (x_1, x_2) in the CAN space, we can exclude the upper-left quadrant rooted at (x_1, x_2) from further forwarding. It is easy to verify the correctness of this observation. The minimum value of the aggregate attribute in the larger range $[x'_1, x'_2]$ is the minimum among the following three quantities: (1) the minimum in $[x'_1, x_1)$, (2) the minimum in $[x_1, x_2]$, and (3) the minimum in $(x_2, x'_2]$. Quantities (1) and (2) cannot change because the update falls in $[x_1, x_2]$. Thus, if the minimum in $[x_1, x_2]$ is not affected, the minimum in the larger range cannot be affected either.

We can stop forwarding even more effectively with the following observation, which is stronger and slightly more subtle:

> **(Cutoff property)** *Suppose that update $\Delta(t : x, y_o \rightarrow y_n)$ does not affect a range-min subscription with range $[x_1, x_2] \ni x$ because of another tuple t' with range attribute value $x' \in [x_1, x_2]$ and aggregate attribute $y' \leq \min(y_o, y_n)$. Then the update cannot affect any range-min subscription with range $[x'_1, x'_2] \supseteq [\min(x, x'), \max(x, x')]$.*

For intuition, consider the example in Figure 1. Update of t_4 does not affect subscription s_2 because of stock t_2. The presence of t_2 "protects" any range-min subscription whose range includes both t_4 and t_2 (e.g., s_1) from being affected by the update of t_4. This property allows us to cut off forwarding of update messages early, instead of forwarding them all the way towards the upper-left corner of the CAN space as is done in CN.

We are now ready to present the CN$^+$ approach using our running example. For each range-min subscription, CN$^+$ maintains a stock tuple with the minimum `PER` in subscription's `RISK` range. `PER` updates (assuming for now that they are ignorable or decreasing) are handled as follows. As in Meghdoot, a PER update to stock with RISK x is first routed to point (x, x) in the 2-d CAN space. The zone owner of this point tags the message with a *cutoff point* (c_1, c_2), initially set to $(-\infty, \infty)$. This message is routed upward and leftward to the zone owners in the upper-left quadrant rooted at (x, x), just as in Meghdoot, but with several differences. First, the message is not forwarded outside the rectangle spanned by (x, x) and the cutoff point. A zone owner does not need to process subscriptions outside the same rectangle because they cannot be affected. For each subscription inside the rectangle, we check its currently maintained minimum PER to see if it is affected. If yes, it is updated. Otherwise, we refine the cutoff point in the message based on the RISK value x' of the stock with the minimum PER: If $x' < x$, we raise c_1 to x'; if $x' > x$, we lower c_2 to x'. The cutoff property discussed earlier is the basis for this refinement.

It is possible for CN$^+$ to handle non-ignorable increasing PER updates with reasonable efficiency, but the details are very messy and we omit them here. On a high level, we distribute the entire database state along the diagonal of the 2-d CAN space, which supports computation of any new minima exposed by bad updates. We have also developed optimizations for sharing this computation among multiple affected subscriptions.

CN$^+$ has a big performance advantage over CN. An ignorable update is detected and stopped very early, as it will not be forwarded beyond the subscription with the smallest range containing it. CN$^+$ also attempts to cut off forwarding of non-ignorable updates as early as possible. The disadvantage of CN$^+$ is its complexity. CN$^+$ pushes a significant amount of application-specific routing logic into the content-based network layer, and the specialized routing algorithms require access to additional state including the contents of subscriptions and the database. The resulting system is difficult to implement and maintain because of the lack of a clean interface separating the network from the database.

2.4 Hybrid Approaches

Our goal in this section is to develop techniques that offer the same or higher level of efficiency as CN$^+$, but without complicating the network substrate with application-specific routing algorithms. To achieve this goal, we need to rethink the traditional responsibilities of servers in a publish/subscribe system, and divide the work carefully between servers and the network. We seek to maximally exploit the capability of a content-based network within the confines of its standard interface. Recall that a content-based network supports subscriptions defined as predicates over the content of each message. In this section, we show that with our message/subscription reformulation techniques, we can support stateful range-min subscriptions efficiently using stateless subscriptions of the form “the data rectangle in the message contains the point of interest.” Such subscriptions are a standard feature in most content-based networks, e.g., [7]; in particular, we show that Meghdoot can handle these subscriptions very efficiently with minimal extension. While this section focuses on range-min subscriptions, we note that message/subscription reformulation is a general mechanism; reformulation techniques for other types of subscriptions will be discussed in Section 4.

2.4.1 S-CN: Server with Content-Based Network

Under this approach, which we term S-CN, a central server maintains the database state and is responsible for generating notification messages and injecting them into a content-based network for dissemination. Interestingly, the server does not need to know the set of subscriptions, which makes S-CN particularly attractive when subscriber anonymity is desired, or when it is expensive for a server to maintain a large, dynamic set of subscribers.

Message/Subscription Reformulation The key idea is for the server to reformulate each publish message into zero or more notification messages whose contents carry additional information derived from the current database state. This additional information effectively removes the dependency of stateful subscriptions on the database state. When stateful subscriptions register with the content-based network, they are first reformulated into stateless subscriptions (without any knowledge of the database state) to work with the reformulated notification message format.

To illustrate the general reformulation mechanism, let us consider a very naive reformulation technique as a warm-up exercise. The server can simply embed the entire database state into each notification message. Doing so obviously makes all stateful subscriptions stateless, but it incurs too much overhead, and may exceed the capability of most content-based networks as they may not support full SQL queries over the message content. How to do better than this naive technique requires non-trivial understanding of different subscription types.

Mar**-Based Reformulation** It turns out that for range-min subscriptions, there exists an efficient and effective reformulation based on Mar (for *Maximum Affected Range*), which intuitively captures an update's "extent of influence" on range-min subscriptions. Informally, using our running example, the Mar of a stock t is the maximum `RISK` range in which t has the minimum `PER` and is the only stock with this `PER`. We formally define Mar below, where a point (x, y) represents a tuple with range attribute value x and aggregate attribute value y:

Definition 1 (Maximum affected range) $\mathsf{Mar}(x_0, y_0)$, *the* Mar *of point* (x_0, y_0) *with respected to a set of distinct points* P*, is the maximum range* $(x_l, x_r) \ni x_0$ *for which there exists no point* $(x, y) \in P$ *such that* $x_l < x < x_r$ *and* $y \leq y_0$*. Let* $\mathsf{Mar}(x_0, y_0) = \emptyset$ *if no such range exists; i.e.,* $\exists(x_0, y) \in P : y \leq y_0$.

The Mar *of an update* $\delta = \Delta(t : x, y_o \to y_n)$*, denoted* $\mathsf{Mar}(\delta)$*, is the union of* $\mathsf{Mar}(x, y_o)$ *and* $\mathsf{Mar}(x, y_n)$*, both of which are defined with respect to the set of points representing all tuples in the relation other than* t.

For example, Figure 3 shows $\mathsf{Mar}(x_0, y_0)$ with respect to a set of points (shown as solid black dots). Basically, $\mathsf{Mar}(x_0, y_0)$ is an open interval between two points: the first one to the left of x_0 and the first one to right of x_0, both with height less than or equal to y_0. As another example, in Figure 1, the Mar of the t_5 update is the range $(20, 100)$. We show in Section 3 how to compute Mar efficiently (in time logarithmic in the size of the database). The following results (proofs omitted due to space constraints) establish the utility of Mar in range-min subscription processing:

Theorem 1 *A range-min subscription with range* $[x_1, x_2]$ *is affected by an update* δ *if and only if* $x \in [x_1, x_2] \subseteq \mathsf{Mar}(\delta)$.

Corollary 1 (Update classification) *Consider an update* $\Delta(t : x, y_o \to y_n)$*. (1) If* $\mathsf{Mar}(t) = \emptyset$*, the update is ignorable. (2) If* $\mathsf{Mar}(t) \neq \emptyset$*, and* $\mathsf{Mar}(x, y_o) \subseteq \mathsf{Mar}(x, y_n)$ *(with respect to the set of points representing all tuples other than* t*), then the update is good, and the new minimum for any affected range-min subscription is* y_n*. (3) Otherwise, the update is bad.*

Corollary 1 provides the tests for the server in S-CN to run in order to classify each incoming database update. Furthermore, this corollary leads immediately to the following reformulation techniques:

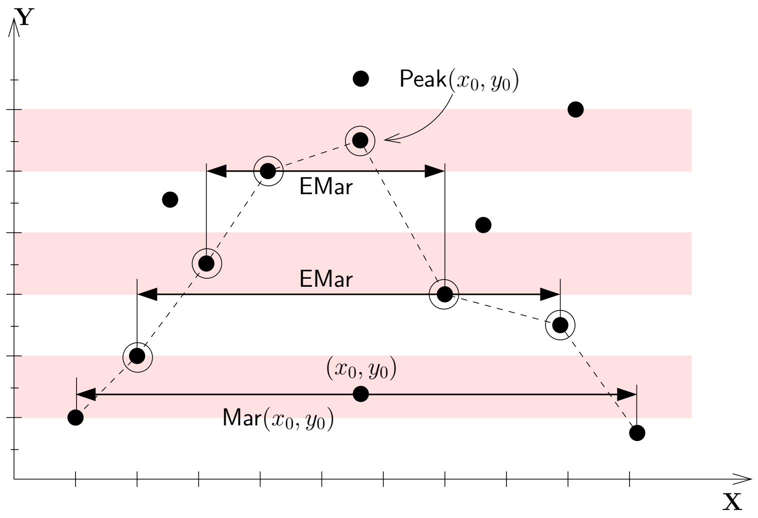


Figure 3: Mar and Hull.

- **(Message format)** Each database update is reformulated into zero or more notification messages of the form
$$\langle \texttt{NEW\_MIN}, \texttt{INNER\_L}, \texttt{INNER\_R}, \texttt{OUTER\_L}, \texttt{OUTER\_R} \rangle$$
and injected into the network.

- **(Subscriptions)** Each range-min subscription over range $[x_1, x_2]$ is reformulated into a predicate
$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R})$$
over the notification message. Upon receiving a message matching the reformulated predicate, a subscriber simply updates the minimum to NEW_MIN.

- **(Ignorable updates)** They are simply discarded by the server.

- **(Good updates)** Each good update $\Delta(t : x, y_o \rightarrow y_n)$ is reformulated as $\langle y_n, x, x, x_1, x_2 \rangle$, where $(x_1, x_2) = \mathsf{Mar}(x, y_n)$, computed with respect to the set of points representing all tuples other than t. For example, the good update $\Delta(t_4 : 40, 12 \rightarrow 4)$ in Figure 1 is reformulated as $\langle 4, 40, 40, 20, 100 \rangle$.

Reformulating Bad Updates As we have seen in Example 1, a bad update (such as the rise of t_5's PER in Figure 1) is tough to handle because it "exposes" different new minima for different subscriptions. Interestingly, with the help of Mar and the concept of *upper hull* introduced below, we can capture all effects of a bad update on affected subscriptions succinctly and precisely, and in a way that allows the server in S-CN to encode them in the same format as the reformulated notification messages for good updates.

Definition 2 (Upper hull) *Consider point (x_0, y_0) and a set P of points. Suppose $[x_l, x_r] = \mathsf{Mar}(x_0, y_0) \neq \emptyset$. $\mathsf{Hull}(x_0, y_0)$, the* upper hull *of point (x_0, y_0) with respect to P, is the set of points consisting of the following:*

- *The* peak, *denoted $\mathsf{Peak}(x_0, y_0)$, is the point $(x_0, y) \in P$ where y is the smallest possible. Let the peak be (x_0, ∞) if no such point exists, i.e., P has no point with X-coordinate of x_0.*

- *The* left upper hull*, denoted* $\mathsf{LHull}(x_0, y_0)$*, is the set of all points* $(x', y') \in P$ *where* $x_l < x' < x_0$*, and there exists no other point* $(x, y) \in P$ *such that* $(x' \leq x < x_0) \wedge (y \leq y')$*.*

- *The* right upper hull*, denoted* $\mathsf{RHull}(x_0, y_0)$*, is the set of all points* $(x', y') \in P$ *where* $x_0 < x' < x_r$*, and there exists no other point* $(x, y) \in P$ *such that* $(x_0 < x \leq x') \wedge (y \leq y')$*.*

For example, Figure 3 circles the points in $\mathsf{Hull}(x_0, y_0)$. Basically, $\mathsf{Hull}(x_0, y_0)$ consists of the two "skylines" [23] that we observe by looking towards left and right from (x_0, y_0). As it turns out, each point $(x', y') \in \mathsf{Hull}(x_0, y_0)$ corresponds to a new minimum that would be exposed by the removal of (x_0, y_0). Intuitively, using $\mathsf{Mar}(x_0, y_0)$, we can capture the set of subscriptions that will have y' as their new minimum. This observation is formalized by the following theorem:

Theorem 2 *Consider a bad update* $\Delta(t : x, y_o \to y_n)$*. Let* P *be the set of points representing the set of tuples after the update has been applied. A range-min subscription with range* $[x_1, x_2]$ *is affected by the update if and only if there exists a point* $(x', y') \in \mathsf{Hull}(x, y_o)$ *(with respect to P) such that* $[\min(x, x'), \max(x, x')] \subseteq [x_1, x_2] \subseteq \mathsf{EMar}(x', y')$*, where* $\mathsf{EMar}(x', y')$ *is the* Exposed Maximum Affected Range *of* (x', y') *with respect to* $P - \{(x', y')\}$*. Furthermore, the new minimum for this affected subscription is* y'*.*

Note that we have used EMar instead of Mar in the above theorem. The two concept are identical except the special case where two points in P have the same Y-coordinate. The difference between EMar and Mar is rather minor and does not affect the exposition in this section. Theorem 2 provides the basis for the following technique for reformulating bad updates:

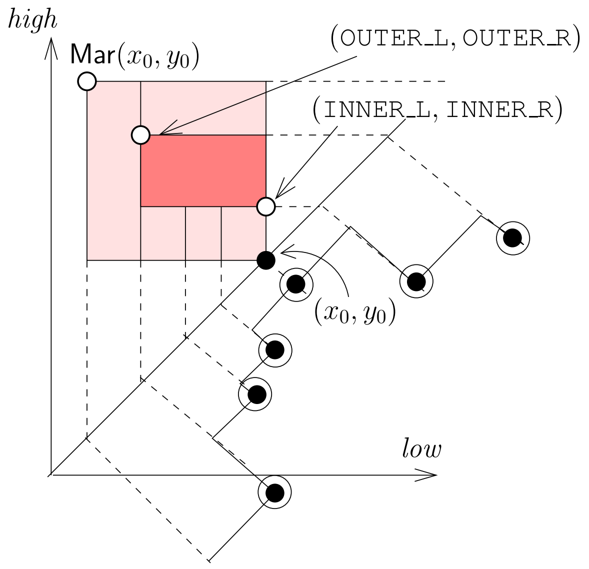
- **(Bad updates)** Given a bad update $\Delta(t : x, y_o \to y_n)$, for each point $(x', y') \in \mathsf{Hull}(x, y_o)$, the server generates a notification message $\langle y', \min(x, x'), \max(x, x'), x_1, x_2 \rangle$, where $(x_1, x_2) = \mathsf{EMar}(x', y')$ (see Theorem 2 for what point sets Hull and EMar are computed with respect to).

Both the notification message format and the behavior of reformulated subscriptions are consistent with those for good updates. The only difference is that the server generates more than one notification message per bad update. As an example, the bad update $\Delta(t_5 : 50, 5 \to 9)$ in Figure 1 is reformulated as 3 notification messages: $\langle 9, 50, 50, 30, 70 \rangle$, $\langle 8, 30, 50, 20, 70 \rangle$, and $\langle 6, 50, 70, 20, 100 \rangle$.

Disseminating Reformulated Messages Recall that S-CN reformulate a range-min subscription over range $[x_1, x_2]$ into the following predicate over reformulated notification messages:

$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R}).$$

We now illustrate how S-CN can disseminate messages to such subscriptions efficiently using Meghdoot with minimal extension. As in Section 2.3, we can picture each subscription as a point (x_1, x_2) in a 2-d CAN space. Each reformulated notification message can be seen as specifying two opposing corners $(\texttt{INNER\_L}, \texttt{INNER\_R})$ and $(\texttt{OUTER\_L}, \texttt{OUTER\_R})$ of a rectangle in this space (as shown in Figure 4). This message matches precisely those subscriptions that fall within the rectangle. Meghdoot already knows how to disseminate a message from a point along the diagonal of the CAN space to its upper-left quadrant. To support dissemination to a rectangular region, we simply need to (1) start the Meghdoot forwarding algorithm from the lower-right corner $(\texttt{INNER\_L}, \texttt{INNER\_R})$, and (2) stop forwarding once the message goes beyond either `OUTER_L` or `OUTER_R`. Note that this extension to Meghdoot is needed only because Meghdoot is a very specialized content-based network designed to support only range subscriptions. Many content-based networks (e.g., [7]) allow subscriptions to be general predicates over message content, and therefore should work with S-CN without additional extension.



$high$
$\mathsf{Mar}(x_0, y_0)$
$(\texttt{OUTER\_L}, \texttt{OUTER\_R})$
$(\texttt{INNER\_L}, \texttt{INNER\_R})$
(x_0, y_0)
low

Figure 4: S-CN routing.

It is interesting to visualize the messages reformulated from good and bad updates as rectangles in the CAN space. A good update is reformulated into a single rectangle, with its lower-right corner corresponding to an 0-length range containing just the update position, and its upper-left corner corresponding to the Mar of the update. A bad update is reformulated into a collection of non-overlapping rectangles, whose union is a big rectangle spanning the position and Mar of the update (Figure 4).

As noted at the beginning of Section 2, we detect ignorable updates without the knowledge of the active subscriptions in the system. Thus, some non-ignorable updates may turn out to be effectively ignorable because certain ranges may not be covered by subscriptions. As a simple optimization, the server in S-CN can maintain the ranges of active subscriptions in the system, and perform a check before injecting a notification message into the network. Doing so would incur extra maintenance overhead; on the other hand, S-CN can still provide some protection of subscriber anonymity, because the server only needs to know the subscription definitions, but not who or where the subscribers are.

2.4.2 DS-CN: Distributing the Server in S-CN

We can replace the central server in S-CN with multiple servers that together maintain the database in a distributed manner, resulting in an approach we call DS-CN. The idea is to leverage the network substrate not only for disseminating notifications, but also for distributing the database state. Consider again our running example. Using CAN/Meghdoot as the network substrate, DS-CN maps a stock with RISK x to a point (x, x) on the diagonal of the CAN space. This stock would be maintained by the zone owner responsible for the corresponding point in the CAN space. In addition, each zone owner along the diagonal maintains pointers to its two immediate neighboring zone owners (left and right) along the diagonal. When a PER update $\delta = \Delta(t : x, y_o \rightarrow y_n)$ enters the system, DS-CN routes it to the zone owner responsible for (x, x). The zone owner then initiates two linear, distributed traversals starting from x: One traversal follows the left zone-owner pointers, scanning stocks in decreasing order of RISK until one (other than t) with PER no greater than $\min(y_o, y_n)$ is reached; the other traversal follows the right zone-owner pointers,

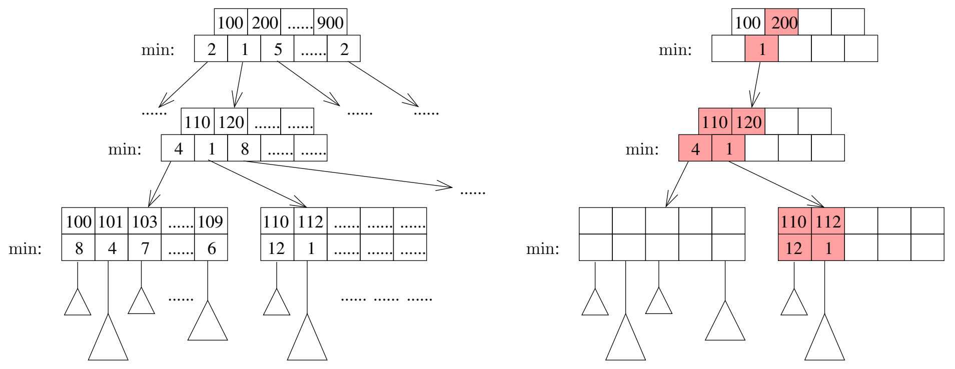


Figure 5: Example of an A2B-tree.

scanning stocks in increasing order of RISK using the same stopping condition. When both traversals stop, DS-CN will have examined all stocks in $\mathsf{Mar}(\delta)$, which provide enough information to reformulate the update (detailed omitted). The rest proceeds in the exact same way as S-CN.

The advantage of DS-CN over S-CN is that there is no bottleneck of a central server. Publish messages no longer need to rendezvous at the same server, and reformulated notification messages are now sent out from different servers. Furthermore, Meghdoot's zone-splitting algorithm [15] can be easily adapted to split diagonal zones that hold too many database state, thereby proving effective load balancing. Note that the two uses of the network substrate by DS-CN—for disseminating notifications and for distributing database state—are completely orthogonal and do not need to interfere with each other; in fact, we can use different overlay networks for the two purposes.

3 Server-Side Data Structures and Algorithms

3.1 Server-Side Processing in S-CN

3.1.1 Computing Mar with A2B-Tree

We begin this section by discussing the data structure and algorithm for computing Mar, which is a primitive used by S-CN for both update classification and message reformulation. Later in Section 3.2 we will also show how to apply the same techniques in database-centric approaches to compute notification messages. In the following, we denote the range attribute by X and the aggregate attribute by Y.

A straightforward way to compute Mar is to use a B-tree on X. Given an update $\delta = \Delta(t : x, y_o \to y_n)$, we look up the tuple(s) with $X = x$ in the B-tree. Starting from these tuples, we scan the index leaves left and right in a manner similar to DS-CN (but on one server). The number of I/Os incurred is $O(\log_B N + L/B)$, where N is the size of the database table, L is the number of tuples in $\mathsf{Mar}(\delta)$, and B is the block size of the B-tree. This method would not scale with large affected ranges.

We propose *A2B-tree* (*augmented* 2*-tier B-tree*) for computing Mar. The upper tier is a B-tree on X. Each leaf index entry of this upper-tier B-tree points to a lower-tier B-tree indexing all tuples with the same X value, using Y as the index key. Furthermore, each upper-tier index entry (which points to either a child in the upper tier or the root of lower-tier B-tree) is augmented with an extra min field maintaining the minimum Y value found in the subtree rooted at this index entry. An example A2B-tree is shown in left part of Figure 5. The space taken by this index is $O(N/B)$, linear in the size of the table. Both the height of the upper tier and the height of the lower tier are bounded by $O(\log_B N)$, so the combined height is also $O(\log_B N)$.

A2B-tree supports lookups and updates in $O(\log_B N)$ I/Os. Lookup follows the standard B-tree procedure, first using X as the search key through the upper tier, and then using Y as the search key through the lower tier. Insertion and deletion extend the standard B-tree procedures with maintenance of $\min$ fields.

Our A2B-tree can be easily maintained by standard B-tree maintenance procedures with slight modifications to adjust $\min$ fields accordingly. For example, when inserting a tuple with $Y = y$, for each upper-tier index entry on the path from the root to the lower-tier B-tree containing the insertion point, we update $\min$ to y if it is currently greater than y. Deletion is slightly trickier than insert because it may remove the tuple who is supplying the $\min$ value for some upper-tier index entry. We proceed in a bottom-up manner, first updating $\min$ of the upper-tier leaf index entry (the new value can be easily obtained from the leftmost leaf of the corresponding lower-tier B-tree), and then updating each entry long the path from left to root until the root, or an entry which does not need to be updated, whichever comes earlier. Similarly, in split and merge, $\min$ filed should also be maintained. For each entry, new $\min$ can be computed in constant time from its children and there are $O(\log n)$ entries need to be modified, therefore, maintaining the auxiliary data does not increase the time complexity of standard B-tree maintenance procedures. Therefore, the I/O cost is bounded by $O(\log_B N)$ in maintenance.

Conceptually, we use following simple example to show how Mar is computed. If we want to compute right end of Mar given point $(x_0, y_0) = (102, 3)$ in the left part of Figure 5. At top level, we first identify the subtree containing $x_0 = 102$, which is associated with key 200. Since the min field of that entry is 1, smaller than $y_0 = 3$, we need to go one level down by following the link to subtree. At the second level, we first identify the subtree containing x_0 again (subtree with key 110 in example), however the min field of this entry is 4, greater than y_0, hence we are sure we cannot find right end of Mar in this subtree since every point in which has greater Y value than y_0. Thus we move rightward to locate the first entry which min field no greater than y_0, which is subtree with key 120 and min field 1, and we go further down to the third level, which is leaf level of our A2B-tree. We repeat the traverse process above to find right end of Mar is $(112, 1)$. The shaded squares in right part of Figure 5 are the entries we visit during the process. We simply return ∞ if we do not find any point in above process.

Following algorithm FINDR$_<(\mathcal{T}, x, y)$ queries right boundary of MAR of a given tuple (x, y) on an augmented B-tree $\mathcal{T}$, assuming domain of x is $(-\infty, +\infty)$ and each node has B entries at most. Sarting from root of $\mathcal{T}$, it traverses downward along the path from root to where x is located, until reaching an entry whose $\min$ field is greater or equal to y. Then it calls subroutine SEARCHR$_<$ to continue the search rightward. Note to compute FINDR$_\leq(\mathcal{T}, x, y)$, we just need to replace $\leq$ with $<$ in line 7 and replace FINDR$_<$ and SEARCHR$_<$ with FINDR$_\leq$ and SEARCHR$_\leq$ respectively.

Since at every level of the upper-tier of A2B-Tree, the algorithm visits 2 nodes at most, as a result its time complexity is $O(\log_B N)$, because it is bounded by twice of height of the augmented B-tree. Computation of MAR serves as a building block in next sections where we will introduce efficient computation and dissemination of a large number of select-aggregation subscriptions given an incoming event.

```
FINDR$_<(\mathcal{T}, x, y)$
1: $m \leftarrow +\infty$, $node \leftarrow \mathcal{T}.root$;
2: **if** $node = null$ **then** {tree is empty}
3:     **return** $m$;
```

```
4: $i \leftarrow 1$;
5: **while** $i \leq B$ and $node.key[i] < x$ **do** {locate index $i$ of subtree by key $x$}
6:     $i \leftarrow i + 1$;
7: **if** $y \leq node.min[i]$ **then**
8:     $m_{r2} \leftarrow \textsc{SearchR}_{<}(node, y, i + 1)$;
9: **else**
10:    **if** *leaf*[node] **then** {node is a leaf}
11:        $m \leftarrow node.key[i]$;
12:    **else** {node is an internal node}
13:        $n \leftarrow \textsc{FindR}_{<}(node.child[i], x, y)$
14:        **if** $n \neq +\infty$ **then**
15:            $m \leftarrow n$;
16:        **else**
17:            $m \leftarrow \textsc{SearchR}_{<}(node, y, i + 1)$;
18: **return** $m$;
```

$\textsc{SearchR}_{<}(node, y, index)$ {Replace the second $\leq$ with $<$ in line 2 to compute $\textsc{SearchR}_{\leq}(\mathcal{T}, x, y)$.}

```
1: $i \leftarrow index$;
2: **while** $i \leq B$ and $y \leq node.min[i]$ **do**
3:     $i \leftarrow i + 1$;
4: **if** $i = B + 1$ **then**
5:     $m \leftarrow +\infty$;
6: **else**
7:     **if** *leaf*[node] **then** {node is a leaf}
8:         $m \leftarrow node.key[i]$;
9:     **else**
10:        $m \leftarrow \textsc{SearchR}_{<}(node.child[i], y, 0)$
11: **return** $m$;
```

3.1.2 Computing Hull and EMar

S-CN needs to compute Hull and EMar in order to reformulate bad updates. Computation of Hull(x_0, y_0) begins with identifying the peak, which is easily located by looking up x_0 in the A2B-tree $\mathcal{T}$. Starting from the peak (x_0, y_p), we call $\textsc{FindR}_{<}(\mathcal{T}, x_0, y_p, \emptyset)$, which returns the X-coordinate of the first point (x_1, y_1) in RHull; y_1 can be readily obtained from the min field of the leaf index entry for x_1. Then, for $i \geq 1$, we repeatedly call $\textsc{FindR}_{<}(\mathcal{T}, x_i, y_i, \emptyset)$ to locate the next point (x_{i+1}, y_{i+1}) in RHull until $y_{i+1} \leq y_0$. The last point is discarded. LHull(x_0, y_0) is computed analogously. Since the cost of each FindR or FindL is $O(log_B N)$, the total cost of computing Hull(x_0, y_0) is $O(k \log_B N)$, where k is the number of points in the

Hull. All index nodes visited in the process belong to the minimum spanning tree containing all points in Hull; we can further ensure that we never visit the same node more than once (details omitted), although this optimization does not change the asymptotic complexity.

Recall from Section 2.4.1 that S-CN reformulates a bad update using EMar for each point in Hull. While we could compute each EMar individually with a procedure similar to Mar computation, it is much more efficient to use the already computed Hull. The key (illustrated in Figure 3) is that for Peak, EMar begins at the rightmost point in LHull and ends at the leftmost point in RHull; for each point (x_i, y_i) in LHull, EMar(x_i, y_i) begins at its left neighbor in LHull, and ends at the first point (x', y') in RHull with $y' < y_i$;[1] for each point (x_i, y_i) in RHull, EMar(x_i, y_i) begins at the last point (x', y') in LHull with $y' \leq yi$, and ends at (x_i, y_i)'s right neighbor in RHull.[2] We exploit the fact that LHull and RHull computation naturally produces points in sorted order. Using a merge-like procedure on LHull and RHull, we can compute EMar for all points in Hull in time $O(k)$, where k is the number of points in Hull, which is also the number of reformulated messages sent out by S-CN for the bad update.

3.1.3 Handling Insertion, Deletion, and Other Updates

So far, for simplicity of presentation, we have only discussed updates of the aggregate attribute. We now briefly present how to support other types of modifications. For insertion and deletion, all results in Section 2 continue to hold if we view an insertion as an update $\Delta(t : x, \infty \to y_n)$, and a deletion as $\Delta(t : x, y_o \to \infty)$. These tuples involving ∞ are of course never stored explicitly; we simply assume their existence for convenience. Updates that change neither X nor Y are ignored. For a complex update $\Delta(t : x_o \to x_n, y_o \to y_n)$ that change both X and Y, we could treat it as a deletion of (x_o, y_o) followed by an insertion of (x_n, y_n). With this simple method, however, it is possible for a subscription to receive two notification messages (one due to deletion and one due to insertion). We have developed techniques to ensure that each subscription gets at most one notification for each publish message, by efficiently consolidating the effects of a complex update. We omit details here due to space constraints. This approach has complexity $O(k \log_B N)$, where k is the minimum number of unique notification massages required for a complex update.

3.2 Server-Side Processing in S-UN and S-MN

We briefly discuss how to use the A2B-tree and concepts of Mar and Hull to compute unicast and multicast notification messages efficiently for S-UN and S-MN. Given an update, a naive method is to examine every subscription in turn and compute how the update affects it; the A2B-tree can be used to recompute in $O(\log_B N)$ time the new minimum in any range when the old minimum is lowered or deleted. This method takes $O(M \log_B N)$ time, where M is the number of subscriptions; it does not scale to a large number of subscriptions. We can do much better by leveraging the techniques in Section 3.1 for S-CN. Using the same algorithms and an A2B-tree for the database table, we compute reformulated messages for each incoming

[1]Note that EMar differs from Mar when multiple points can share the same Y-coordinate (as mentioned in Section 2.4.1). Mar(x, y) extends to the first point whose Y is *less than or equal to* y, because for a good update, we do not want to notify a subscription whose minimum is already y. In contrast, EMar(x_i, y_i) in this case extends to the first point with Y *strictly less* than y_i. The reason is that for a bad update, we want to send out as few messages as possible to cover all exposed minima; therefore, we want each EMar to cover as much of the exposed range as possible.

[2]Note the slight asymmetry in defining EMar for LHull and RHull points ($y' < y_i$ vs. $y' \leq y_i$). In the special case where a point in LHull has the same Y-coordinate as a point in RHull, we need this asymmetry to ensure the property that every subscription affected by a bad update receives exactly one notification message (i.e., no redundant or missing notifications).

update as in S-CN. Instead of disseminating these messages, however, we treat each message as a query identifying the set of affected subscriptions. As in Figure 2, we view the range-min subscriptions as points in a 2-d space, and store them using a 2-d index structure that supports rectangular queries. An external-memory kd-tree, for example, would support such a query in $O(\sqrt{M/B}+J/B)$ time, where J is the number of subscriptions in the query rectangle. S-UN generates one unicast message for each such subscription; S-MN generates one multicast message that encodes this set of subscriptions. Overall, for an update with k reformulated messages in S-CN affecting a total of A out of M subscriptions, the server-side processing costs incurred by S-UN and S-MN are $O(k \log_B N + k\sqrt{M/B} + A/B)$, compared with $O(k \log_B N)$ for S-CN.

4 Other Subscription Types

In this section, we briefly discuss how to handle other subscriptions with a hybrid approach such as S-CN, using our general message/subscription reformulation mechanism.

Range-max subscriptions can be handled by the same techniques as range-min. Range-count/sum/average subscriptions are easier to handle: We simply reformulate them into range subscriptions without aggregation; publish messages do not need to be reformulated (though obviously irrelevant updates can be ignored, e.g., those updating neither range nor aggregation attributes). Unlike range-min/max, relaxing these range-aggregation subscriptions would not result in excessive traffic, because relevant updates that fall within a subscription range generally do affect the subscription.

A range-`DISTINCT` subscription tracks the set of distinct values of an attribute Y for tuples whose range attribute X fall within some range. Simply relaxing this subscription into a range subscription may generate a lot of unnecessary traffic if there are many duplicates. In this case, we can extend the concept of Mar as follows: Mar of an insertion (or deletion) is the maximum X range that contains the insertion (or deletion) point and no other tuples with the same Y value, as shown in left part of of Figure 6. An insertion (or deletion) is reformulated into a message containing X and Y values and the Mar, if it is not empty. A range-`DISTINCT` subscription is reformulated into a stateless selection subscription that checks if the subscription range contains the X value and is also contained by the Mar.

Select-join subscriptions are also stateful. Given a publish message that applies an update δR to table R, its effect on subscription $\sigma_p(\sigma_{p_R} R \bowtie \sigma_{p_S} S)$ is $\sigma_p(\sigma_{p_R}\{\delta R\} \bowtie \sigma_{p_S} S)$, which requires accessing state (content of table S) not in the original update message. Following the message/subscription reformulation approach, a server maintaining the database state can reformulate each δR into a series of notification messages, each containing a result tuple in $\{\delta R\} \bowtie S$. Meanwhile, the select-join subscription $\sigma_p(\sigma_{p_R} R \bowtie \sigma_{p_S} S)$ is reformulated into a stateless subscription that checks condition $p \wedge p_R \wedge p_S$ over reformulated messages.

Last but not least, we have extended our techniques for 1-d range aggregation/`DISTINCT` subscriptions to the 2-d case, where each subscription can specify orthogonal range conditions for two attributes. The shaded area in right part of Figure 6 is an example of MAR^{2d} for a tuple r. The contour of shaded area is determined by a set of tuples in 2d range selection space X_1, X_2 such that for each of them r', $r'.y \leq r < y$ and there does not exist a tuple r'' in the shaded area such that $r''.y \leq r.y$. It is not hard to prove that a subscription will be affected by update of r if and only if it is stabbed by r and its orthogonal range selection condition is fully contained within MAR^{2d}. We leave for the future work the data structure and algorithms computing MAR^{2d} and hull in two dimension which both can be computed similarly to one dimension.

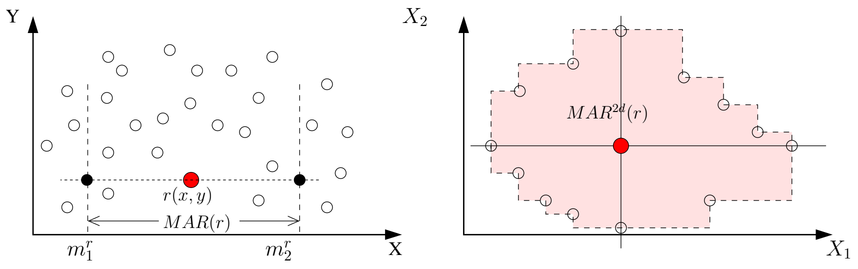


Figure 6: Generalized MAR.

In a publish/subscribe system that uses a single network substrate to support more than one types of subscriptions, we use an additional `TYPE` field in reformulated messages to distinguish those intended for different types of subscriptions. A reformulated subscription would also include an extra condition that selects notification messages with the appropriate `TYPE` value.

5 Evaluation

5.1 Implementation Details

On the server side, we implemented the algorithms computing Mar and Hull, and the processing techniques introduced in Section 3. To compare with S-CN, we also implemented the query processing component for unicast and message grouping component for multicast. The server module supports well-defined network interfaces to a regular network for unicast and multicast, and CAN for S-CN. On the network side, we have implemented a network simulator for a large-scale publish/subscribe system. The first phase of network simulation generates application-level routing details that are used by a second phase which can accept any topology generated using INET [10], an Internet-like network topology generator. This phase performs a link-level simulation (timing is not simulated) of the network topology. We support a number of dissemination styles, as discussed next.

Unicast from a centralized server. First, the server determines the set of subscribers affected by an update. The network simulator sends a single hop unicast message to each subscriber, carrying the new answer to that subscription. The route follows the shortest path over the underlying IP substrate from the server to the destination.

Multicast from a centralized server. For each exposed answer, the server determines the set of destinations that need to receive a multicast message with that answer.

A message containing this new answer tuple has to be multicast to the set of destinations. The network simulator uses this application-level data to perform multicast as described next. Given a set of N possible destinations, it would take 2^N groups to be able to directly multicast to any subset of recipients. We use application-level multicast in our simulator, as IP multicast is not widely supported and has severe limitations in terms of number of groups. We implement an efficient hierarchical multicast approach that limits the number of groups that any single node needs to be aware of.

The hierarchical multicast that we have designed and implemented is inspired by NICE [5] and works as follows. Consider a network with N nodes. In addition to its IP address, each node can be identified by its position within the network, described as a set of coordinates in a geometric space such as the one proposed by *Global Network Positioning* (*GNP*) [21]. GNP assigns coordinates to nodes such that their geometric distances in the GNP space approximate their actual network distances. We then use a geographic clustering algorithm such as k-means to form a tree of nodes as follows. We define a limit c on the number of children at any node in the tree. Clustering is performed on a set of n nodes to give $min(c, n/c)$ sub-clusters of nodes. Each sub-cluster has a leader, and the leader of the original cluster is the parent of these sub-cluster leaders in the tree. The tree is rooted at the server which acts as the leader of the highest level cluster consisting of all nodes in the network. Each cluster C with leader L_C is further clustered in the same manner and L_C is the parent of the leaders of each of these sub-clusters. Clustering is no longer performed on a cluster if it has no more than c nodes. This gives a tree rooted at the server, with each non-leaf node having at most c children, and the tree has a total of N leaves. Each leader L with k children forms a multicast group for each of the 2^k possible subsets of the k children. It maintains a routing table that provides, for each multicast group, the application-level multicast tree that is constructed rooted at L and reaching all the members of that multicast group. The application-level multicast tree can be constructed in a number of ways - we use a greedy offline algorithm to construct a bottleneck bandwidth tree; this was used in Bullet [18] as a good offline technique to compare with dynamic application-level multicast techniques. Note that the number of group IDs at any node is bounded by 2^c and hence, by limiting c we can ensure that no node needs to be aware of a large number of multicast groups.

When a node L receives a message along with a set S of recipient nodes for that message, it can compute the subset of its children that need to receive the message. All it needs to compute this is a bitmap of all the nodes, formed by a left-to-right ordering of the nodes in the tree under node L. At every level, the nodes in each sub-cluster form a contiguous chunk in the bitmap. By looking at the bit positions that need to be reached in its chunk of the bitmap, a node can determine the subset of its children to which the message needs to be forwarded. It can then look up the multicast group ID for that subset and forward the message along the application-level multicast tree described earlier. This process is repeated until the message reaches all its intended recipients. Note that the message reaching a node N needs to encode the exact set of destination subscribers located below N as this is needed to determine the set of affected children at that node.

CAN routing with range predicates, with Mar, and with additional state (no server). We use the Meghdoot simulator [15] in order to evaluate the basic CNapproach with just range predicates (described in Section 2.3), without specialized techniques to handle more complicated queries. The simulator is augmented with our link-level simulator for more accurate routing statistics. In order to support the sophisticated techniques used in S-CN (such as Mar) and DS-CN (such as diagonal two-way routing), we implemented appropriate extensions to the simulator.

5.2 Evaluation Metrics

We use both server- and network-side metrics for evaluation. On the server-side, we track processing time, which is measured as the period between the time at which an update arrives at the server and the time at which the server completes generation of all outgoing messages for dissemination. On the network-side, we track, for each event: (1) *Number of overlay message hops*: This is the total number of messages sent between overlay nodes, in order to process and disseminate that event. (2) *Number of IP message hops*: This the number of hops over IP-level links, during dissemination of an event. An overlay hop may traverse a number of IP-level links on its path. (3) *Network traffic*: We define network traffic as the total number of bytes that need to be transferred between overlay nodes during dissemination. (4) *Maximum node stress (MNS)*: We define node stress as the number of overlay messages originating from a node. MNS for an event is the highest node stress among all nodes while processing that event.

5.3 Workload

We use normal distributions to derive 1-d range-min subscriptions. To model a *hot* range which interests more subscribers, the position of the subscription interval is normally distributed around center of domain of local selection attribute. The interval length also follows a normal distribution. This subscription workload is used in association with both synthetic and real update workloads.

Our synthetic update workload consists initially of a database that contains between $10,000$ and $100,000$ tuples uniformly distributed in the domain. We generate $200,000$ events (long enough to reach stable measurements), each being an update of the aggregate attribute `PER`, and collect the measurements of each update. All experimental parameters are summarized in Table 1. We vary one or more of these parameters to perform experiments with varying database size, number of subscriptions, percentage of ignorable updates, and the average number of subscriptions affected.

We update a tuple by increasing or decreasing its output attribute using a random walk model. The step depends on the current value and updates are independent of each other. Each random variable (tuple) in this model is actually an irreducible, finite, and aperiodic Markov chain; hence there exists a stationary distribution for the value of each tuple. Consequently, update statistics such as the number of subscriptions affected by an update will be stationary over time. Our simulation is long enough to ensure convergence. Detailed proof of convergence of our update model is omitted for brevity. To make the workload more realistic, we also introduce a small percentage of *spikes*. A spike is an update where the `PER` drops suddenly (affecting a large number of subscriptions) and then bounces back to its old value. We also use a real update workload based on stock data from Yahoo! Finance [14]. More details are provided in Section 5.5.6.

5.4 Experimental setup

We perform detailed link-level simulation of a $20,000$-node INET topology. Of these, 1000 nodes are chosen as the end nodes participating in an overlay network. Events are generated by publishers assumed to be randomly scattered throughout the network.

We assume that publishers are distributed throughout the network. We do not model the message hop from the publisher to the server/event point, because this cost would be incurred in all systems. If the publisher is not a part of the CAN, it could contact some peer as an entry point into the CAN. Similarly, subscriptions could be distributed throughout the network, but each subscription chooses a peer in the overlay network as its gateway to the CAN. We use the following technique to choose a gateway in our experiments. In CAN-based techniques, the zone owner is chosen as the gateway for all subscriptions that map to that zone. The same mapping is used in all the compared approaches. We assume that subscriptions reside at the gateway and do not model the propagation of messages from the gateway to the end subscriber. In case of multicast, the network-side metrics for each group ID at each node in the hierarchy are precomputed to speed up the simulation. In addition, c is fixed at 10 so that no more than 1024 multicast trees need to be stored at any node in the system.

5.5 Experiments and results

5.5.1 Demonstration of scalability

In this set of experiments, we compare the techniques in terms of their ability to scale to large numbers of tuples and subscriptions. On the server side, we compare the average processing time per update for

parameter	value
number of overlay nodes	1000
number of physical nodes	$20,000$
domain of range select attribute	$[0, 10, 000]$
domain of aggregate attribute	$[0, 10, 000]$
number of subscriptions	$100k - 1M$
midpoint of selection range	$N(5000, 1500)$
length of selection range	$N(1000, 1000)$
number of tuples in initial DB	$10k - 100k$
number of simulation events	$200, 000$
percentage of spiked events	0.5%

Table 1: Summary of all parameters used in experiments.

db size	$20K$	$40K$	$60K$	$80K$	$100K$
S-CN	2.10	2.15	2.18	2.20	2.22
Unicast	726	671	678	655	654

Table 2: Number of outgoing messages from server, varying database size

# subs.	$200K$	$400K$	$600K$	$800K$	$1M$
S-CN	2.08	2.08	2.08	2.08	2.08
Unicast	303	616	909	1218	1441

Table 3: Number of outgoing messages from server, varying num. of subscriptions

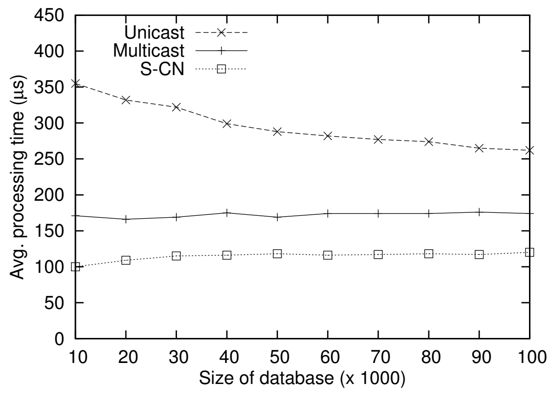


Figure 7: Avg. processing time; increasing database size.

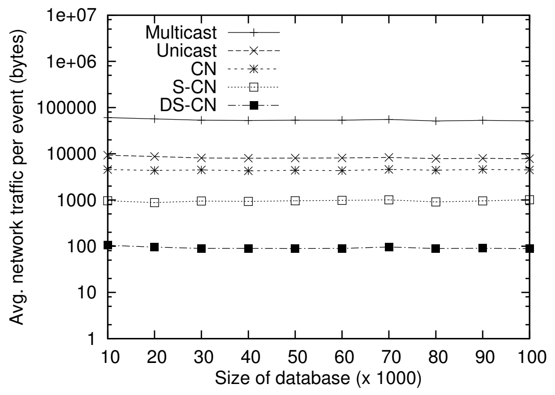


Figure 8: Avg. network traffic; increasing database size.

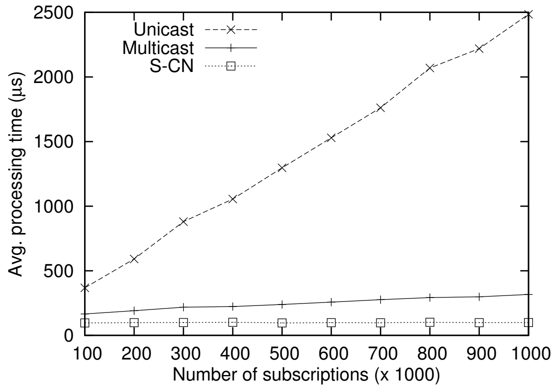


Figure 9: Avg. processing time; increasing number of subscriptions.

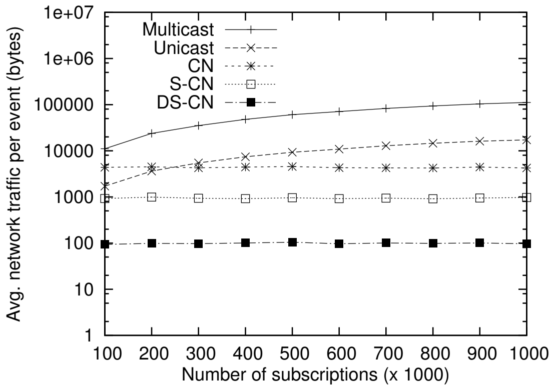
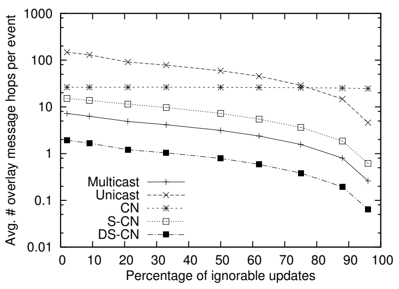


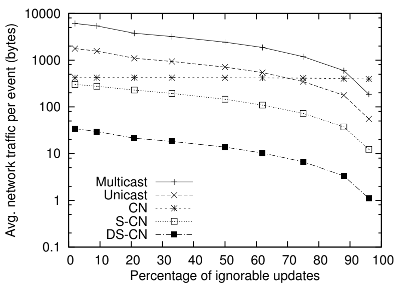
Figure 10: Avg. network traffic; increasing number of subscriptions.

S-CN, unicast and multicast (recall that CN and DS-CN do not have a central server). On the network side, we compare the average network traffic (bytes) generated per event. All updates are non-ignorable and the percentage of spikes is 0.5%.

We first vary the database size from $10,000$ to $100,000$ tuples, for $500,000$ subscriptions. Figures 7 and 8 show the server- and network-side costs respectively. On the network side, CN, S-CN, and DS-CN perform much better than unicast and multicast, and the average network traffic is independent of database size. Note that we have shown network traffic in log scale on the y-axis because of the order of magnitude difference between the various approaches. Among these three approaches, DS-CN performs the best in terms of traffic as the diagonal traversal usually takes very few hops. S-CN is next, the main factor for performing worse than DS-CN is the cost of routing from the server to the event point. This is followed by CN. Among central server approaches, S-CN achieves the lowest processing time, and its processing time increases only by 20% when the database size increases by a factor of 10. The processing time of multicast is roughly 50-70% higher than S-CN due to the cost of identifying all affected subscriptions. The trend for multicast is flat because the slight increase in processing time over increasing database size is compensated by a decreasing number of affected subscriptions: a larger database gives a subscription more resistance against update. However, the network-side performance of multicast is the worst because the set of affected subscriptions needs to be encoded in each message, even though the number of multicast messages may be small. Unicast performs badly on both server- and network-side due to the high cost of assembling and disseminating outgoing messages for each affected subscription. The processing time of unicast also benefits from a larger database.

Next, we keep the database size at $10,000$ tuples and vary the number of subscriptions from $100,000$ to 1 million. Figures 9 and 10 show scalability on the server and network side respectively. S-CN is completely independent of subscriptions; thus, both its processing time and network traffic is approximately constant. The three approaches of CN, S-CN, and DS-CN perform well with traffic never rising above $1kB$. All these approaches are independent of the number of subscriptions, which makes them scalable. The processing time of multicast increases over number of subscriptions. Unicast also increases but it performs much worse due to the overhead of memory allocation for constructing objects corresponding to outgoing messages (there are many outgoing messages for unicast). Even if we can optimize it using bulk memory allocation, it cannot beat multicast, which is worse than S-CN. On the network side, multicast is good in terms of number of overlay hops (not shown) but it performs badly in terms of total traffic generated.





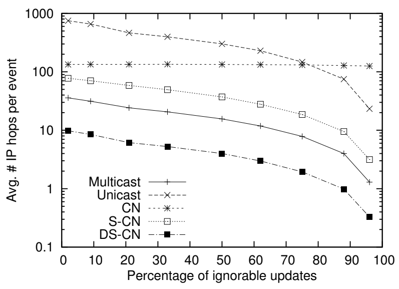


Figure 11: Performance of various approaches, varying the percentage of ignorable updates.

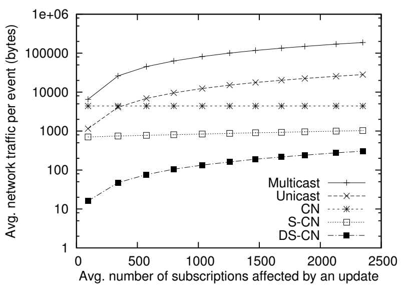


Figure 12: Traffic vs. # affected subs.

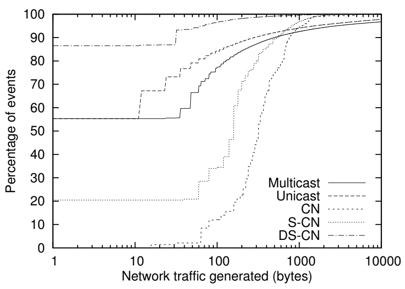


Figure 13: CDF of network traffic.

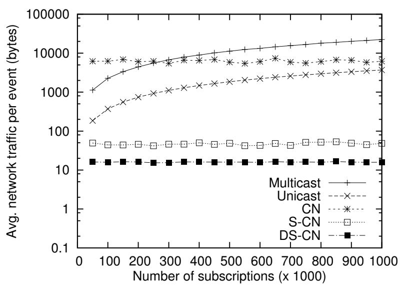


Figure 14: Result of real workload.

Tables 2 and 3 compare the number of outgoing messages (from server) of S-CN and unicast for the two sets of experiments. In all cases, S-CN saves more than 99% of outgoing messages. We also evaluated CN^{+} for the portion of workload with decreasing updates. We found that early stopping is effective and results in traffic reduction of more than 98% compared to CN. Early stopping in CN^{+} (for decreasing updates) was around 2% less effective in terms of traffic than the Mar-based stop conditions in S-CN. However, handling increasing updates in CN^{+} is complex (as discussed earlier) and would cause more traffic. We do not advocate CN^{+} because it pushes a significant amount of complex application-specific routing logic into the network layer.

5.5.2 Varying percentage of ignorable updates

We next demonstrate the effect of increasing the percentage of ignorable updates I. To better control the parameter, we use a subscription distribution where the midpoint of the selection range is taken from a normal distribution $N(5000, 10000)$ and the length is taken from a normal distribution $N(50, 10)$. There are no spikes introduced. Figure 11 shows results for three network metrics. The left figure shows the average number of overlay hops per event, while the middle figure shows the average network traffic (in bytes) per event, for all approaches. The serverless CN approach is independent of I because we cannot truncate ignorable updates. As I is increased, the performance of all the approaches except CN improves and, for high values of I, CN becomes the worst approach. For high values of I, serverless approaches are less desirable as they are unable to detect and truncate ignorable events. However, DS-CN still performs well because in most cases where an update is ignorable, it can be detected at the zone owner containing the event point itself. Overall, the best performance is achieved by DS-CN. Multicast performs quite well in terms of number of overlay messages, with S-CN following up in third position. However, as the middle figure shows, multicast is the worst in terms of total network traffic (due to message size). The right figure

Approach	Max. node stress
Unicast	$10,282$
Multicast	3
S-CN	23
DS-CN	25
CN	8

Table 4: Maximum node stress

shows the average number of IP-level hops per event, for each of the various approaches. The trends are similar to those of overlay hops. In further experiments, due to space reasons, we show only the average network traffic (in bytes) for various approaches; the other metrics were found to follow similar trends as described here, in all our experiments. In summary, the single server-based S-CN and the distributed DS-CN perform the best as the messages encode the affected subscriptions very compactly and messages are sent only to those CAN regions which could be affected by the event.

5.5.3 Varying number of affected subscriptions

We increase the average number of subscriptions affected by a non-ignorable update by controlling the percentage of spikes which affect a large number of subscriptions. All updates are non-ignorable. The number of subscriptions is $500,000$ and the database size is $50,000$ tuples. Figure 12 shows the performance of each of the approaches in terms of average network traffic (in bytes). From the figure, we see that unicast and multicast worsen in performance linearly with increase in average number of affected subscriptions. CN is independent of this parameter and hence shows a flat line. The network traffic generated by S-CN increases very slightly across the workloads as seen from the figure. This is due to larger average Mar as a result of increasing percentage of spikes. Finally, DS-CN shows an increasing trend; the reason is that since spikes have a large Mar, the diagonal traversal generates more traffic. In real workloads, such updates with large Mar are extremely rare.

5.5.4 Maximum node stress for various approaches

We compared the maximum node stress (MNS) for a workload with 21% ignorable updates, $500,000$ subscriptions, and $10,000$ database tuples. There were no spiked events. The highest MNS (across events) for unicast was found to be extremely high $(10,282)$ at the server. From the CDF of MNS over events (not shown), we found that although more than half the events generate no message, a fair percentage of events generate very high node stress at the server. This demonstrates the main disadvantage of unicast: lack of scalability for large-scale subscriptions. The highest MNS of S-CN and DS-CN were found to be just 23 and 25 respectively. In S-CN, 21% of events had 0 MNS, while in DS-CN, nearly 87% of events had 0 MNS. In both approaches, over 99% of events had a MNS less than 10. Multicast had a highest MNS of just 3 because the multicast trees are extremely skinny. Finally, CN had a highest MNS of 8. Although nearly every event has a non-zero MNS in CN, most events have a small MNS, for example, over 94% of events have a MNS of no more than 3.

5.5.5 Distribution of network traffic

We plot the CDF (over events) of network traffic for all approaches, in Figure 13. There are $500,000$ subscriptions and $10,000$ database tuples. The percentage of ignorable updates is 21%. From the CDF, we see that S-CN, CN, and DS-CN perform well with average network traffic generated by an event less than $1kB$ for nearly all events. DS-CN is seen to be the most efficient as nearly 87% of events do not generate any traffic. S-CN also performs well but only the ignorable updates generate no traffic (other events need to be routed to the event point). However, nearly 97% of events generate traffic of less than $1kB$. The performance of CN is worse. Around 94% of events generate less than $1kB$ traffic, but more than half the events generate 320 bytes or more traffic. Unicast and multicast perform poorly overall as expected. However, these approaches perform well for events that affect few or no subscriptions: as a result, more than 77% of messages cause network traffic of fewer than 100 bytes. But, the performance degrades rather rapidly for the rest of the events, making these approaches very bad overall. Traffic of upto $680kB$ is seen for some events that affect a large number of subscriptions.

5.5.6 Experiments on a real workload

In order to evaluate our techniques on a real trace, we obtained information for 3053 stocks from Yahoo! Finance [14]. We gathered data for earnings per stock (`EPS`) for each of the stocks. In addition, computed the average recommendation over the past month (`RECO`) for each stock. `RECO` varies from 1.0 (strong buy) to 5.0 (strong sell). We collected open and close price data over the course of 60 days, and used `EPS` to compute `PER`for each price. We thus obtained a trace of events, each being an update of `PER` with `RECO` constant. The trace had $338,415$ events. 11.7% of the events were non-ignorable events. Note that although this trace has only two events per day, real-time stock prices over the course of the day would follow a similar update trend, with several thousand updates generated every few seconds. This would need efficient database/network coordination to scale to large subscription sets.

We generated $500,000$ subscriptions; each subscription requests the minimum `PER` over a specified `RECO` range. This is a meaningful query because stocks with lower `PER` are intuitively better. Moreover, people may desire stocks rated at different ranges of `RECO`.

Figure 14 shows the average network traffic per event as we increase the number of subscriptions. We see that S-CN and DS-CN generate orders of magnitude lesser traffic than CN, unicast, and multicast. Both unicast and multicast do not scale well; their performances degrade linearly with increase in number of subscriptions. CN shows constant but bad performance. S-CN and DS-CN perform very well and are both independent of number of subscriptions. They generate less than 100 bytes of network traffic per event on the average, with maximum node stress never rising above 10.

6 Related Work

Publish/subscribe systems. Recently, research efforts have been focused on content-based publish subscribe systems which provide fine granularity and flexibility. These can be broadly characterized as systems where publishers publish events following a particular predefined schema, and subscribers express their interests as *profiles* [12] which are predicates over the schema. A large number of such systems have been built in recent years, e.g., SIFT [28] (for text documents), ONYX [12] (for filtering and transformation of XML messages), and the wide-area event notification service [6]. In all these systems, subscriptions are stateless filters defined over individual messages, so they cannot express queries of interest across different messages or over the event history. Profiles are not powerful enough to accommodate stateful SQL-style subscription requirements. ONYX supports on-the-fly transformation of an XML message according to a subset of XQuery; but filtering and transformations are still limited to individual messages. A few continuous query systems also support rich query languages. Unfortunately, these do not address the problem of efficiently delivering updates over a network. ONYX has begun addressing this problem; however, the focus of ONYX on supporting transformation of XML messages is different from our goal of supporting more general stateful SQL subscriptions that cannot be processed on individual messages.

Database-side processing. Continuous query systems [19, 11] and stream processing systems [1] can be regarded as a form of publish/subscribe system where continuous queries over streams correspond to our subscriptions. These systems provide automatic notification whenever a continuous query result changes.

The idea of group processing has been identified and used in trigger processing and continuous query processing systems [16, 11]. Work on scalable database trigger processing [16] focuses on exploiting common patterns in triggering conditions (like our notification conditions). Work on scalable continuous query processing (e.g., [11, 20]) focuses on exploiting common patterns in continuous queries (like our subscription queries). In particular, predicate and query indexing techniques have been developed in [16, 11, 13] to speed up group processing. The upper hull computed for dissemination is similar to computing dynamic skyline in [23]. Their algorithm is based on regular R-tree, which cannot offer the same guaranteed performance as our A2B-tree.

Network dissemination. A server needs to deliver notifications to affected subscribers over a network. The problem of efficient message delivery has long been tackled in networking and distributed systems research. The traditional delivery mechanism is based on client polling. The next generation of delivery mechanism uses real push techniques based on group-based multicast protocols, e.g., IP multicast. Multicast provides a perfect interface for channel-based subscription services. IP multicast has also been exploited in building publish/subscribe systems that support more general filter-style subscriptions [22]. Because of slow adoption of IP multicast, there have been proposals for supporting application-level multicast using an overlay network (e.g., [8]). Oftentimes, they use an overlay network called distributed hash tables, which provide a convenient hash table abstraction over the participating overlay nodes. The problems with multicast were discussed in Section 2.2.2. Although we have included multicast as a comparison, the problem with most multicast techniques is that of number of groups. In order to implement publish/subscribe, we need a large number of groups (one for each possible subscriber set) and these techniques do not scale to large number of groups. Group reduction techniques [22] require the end subscriber to have the ability to filter out unwanted messages. The group size problem is seen in our experimental results with our implementation of multicast, where we avoided the need for large number of groups by encoding the set of subscriptions as part of the message.

An alternative dissemination interface is content-based networking [7, 3, 9]. A content-based network can be used to implement a publish/subscribe system supporting filter subscriptions. A number of such systems have been developed (e.g., [27, 26, 15]). SemCast [24] proposes a number of techniques for efficient dissemination including the use of dynamic statistics. However, subscriptions in all these systems are limited to stateless filters. Nevertheless, they can still be used by our system as the messaging layer for delivering notifications once they are computed. We use message and subscription reformulation to enable traditional content-based networks to handle stateful queries.

7 Conclusions

We approach the construction of large-scale publish/subscribe systems by viewing the problem from the perspective of the interface between the database and the network. Different techniques vary in the degree of database/network cooperation; some are more suitable than others for certain types of queries and/or workloads. The tradeoffs are illustrated by the following table, which compares techniques based on how they handle stateful subscriptions.

Tech-nique	Network side			Server side		Implemen-tation cost
	Traffic	MNS	State (subs)	Processing	State	
S-UN	Very high	High	None	Medium	High	Low
S-MN	Very high	Low	None	Medium	High	Low
CN	High	Low	High	None	None	Medium
CN^+	Medium	Low	Medium	None	None	High
S-CN	Low	Low	Low	Low	Low	Medium
DS-CN	Low	Low	Low	None	None	Medium

It is clear that each technique has its strengths and weaknesses. For example, although unicast does not require state at subscriptions, the update traffic is very high. CN^+ introduces application- specific logic into the network and needs per-subscription state. S-CN and DS-CN perform well overall, with dramatic reduction in traffic at low server-side processing cost. We showed that simply converting a stateful subscription to a stateless one does not yield a scalable solution. Our message and subscription reformulation mechanisms are better because they can efficiently embed state information into messages. It is possible for a normal content-based network (which can handle only stateless subscriptions) to handle several classes of stateful subscriptions efficiently: the key is to transform events into a semantic description of affected subscriptions, and subscriptions into a predicate over the semantic description. We demonstrated this using several query classes including range aggregation, distinct, and join. We experimentally validated our techniques for range aggregation, and showed that it is possible to achieve orders-of-magnitude improvement over a naive transformation of the stateful subscription to a stateless one.

References

[1] Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 2003.

[2] L. Aguilar. Datagram routing for internet multicasting. In *SIGCOMM*, 1984.

[3] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish- subscribe systems. In *ICDCS*, 1999.

[5] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM*, pages 205–217, 2002.

[6] 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.

[7] 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.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communication*, 2002.

[9] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, 2003.

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

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

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

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

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

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

[16] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE*, 1999.

[17] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR DB system. In *ICDE*, 2003.

[18] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[19] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

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

[21] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.

[22] 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.

[23] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 2005.

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

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

[26] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.

[27] P. Triantafillou and I. Aekaterinidis. Content-based publish/subscribe systems over structured P2P networks. In *DEBS*, 2004.

[28] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM TODS*, 1999.