Value-Based Notification Conditions in Large-Scale Publish/Subscribe Systems[*]

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

ABSTRACT

We address the problem of providing scalable support for subscriptions with personalized value-based notification conditions in wide-area publish/subscribe systems. Notification conditions can be fine-tuned by subscribers, allowing precise and flexible control of when events are delivered to the subscribers. For example, a user may specify that she should be notified if and only if the price of a particular stock moves outside a "radius" around her last notified value. Naive techniques for handling notification conditions are not scalable. It is challenging to share subscription processing and notification dissemination of subscriptions with personalized value-based notification conditions, because two subscriptions may see two completely different sequences of notifications even if they specify the same radius. We develop and experimentally evaluate scalable processing and dissemination techniques for these subscriptions. Our approach uses standard network substrates for notification dissemination, and avoids pushing complex application processing into the network. Compared with other alternatives, our approach generates orders of magnitude lower network traffic, and incurs lower server processing cost.

1   Introduction

Today, we are faced with a huge increase in demand for personalized data. Millions of users request data like stock prices to be delivered to their cell phones, desktop clients, or email inboxes. The data needs are potentially different across subscribers. Publish/subscribe systems are a suitable middleware for matching user needs (expressed as *subscriptions*) to incoming *events* generated by data sources (called *publishers*). Traditional topic-based and content-based publish/subscribe systems support *stateless* subscriptions, i.e., those that can be processed by only examining the incoming event itself. Personalized data needs, however, often translate to *stateful* subscriptions. One approach to supporting such subscriptions in traditional systems is to add post-processing logic and state maintenance at subscribers, but this approach is not scalable, as we will show in this paper.

Consider, for example, an application that delivers stock price

updates to a large number of subscriptions. While many subscribers may be interested in the same stock, each subscriber may have a unique data need in the form of a *notification condition*, which precisely specifies when the subscriber wishes to receive a price update. In this paper, we focus on *value-based* notification conditions, where each subscriber is notified of the new price if and only if it differs from the price value last received by the subscriber by no less than a threshold (called *radius* and specified as part of the subscription definition). This problem setting corresponds closely to the well-known problem of *bounded approximate caching* [15]. However, we have the stricter requirement that the subscriber (cache) should not be updated without a radius (bound) violation, and we also address the challenge of scaling to a large number of subscriptions.

Value-based notification conditions offer flexible, personalized control of when value updates are disseminated to subscribers. Besides stock price monitoring, value-based notification conditions are also useful in scenarios such as monitoring auctions and online sales prices, tracking sports scores and vote counts in elections, etc. Subscriptions with these notification conditions can also be used as building blocks for implementing wide-area approximate caching.

Challenges   Support for value-based notification conditions in a publish/subscribe system involves the standard challenge of scalable subscription processing and notification dissemination, in the presence of millions of subscriptions. Naive solutions do not scale: 1) Given an incoming event, if we let a server check all subscriptions in turn and notify each affected one with unicast, processing and dissemination costs can easily overwhelm the server. 2) If we disseminate all events to subscribers and rely on post-processing at each subscriber to enforce notification conditions, there will be excessive network traffic, most of which is unnecessary.

To develop more efficient group processing and dissemination techniques, we are faced with several unique challenges raised by value-based notification conditions. First, each such subscription has a potentially unique per-subscription state that may need to be separately maintained—the last value that was sent to that particular subscription. This state can be different for different subscriptions, even for those with identical radius. Second, there is no simple subsumption relationship among subscriptions based on their radii. In other words, an update that needs to be sent to a subscription with a larger radius should not necessarily be sent to a subscription with a smaller radius (or vice versa). Even subscriptions with the same radius may receive completely different sequences of updates, depending upon when these subscriptions were created. We will explain these subtleties further using examples in Section 2.

Contributions   We develop and evaluate a set of techniques for supporting scalable processing and dissemination for a large number of subscriptions with value-based notification conditions. Com-

---

pared with less sophisticated alternatives, our techniques show a huge performance improvement in server processing, and an order of magnitude lower network cost for disseminating notifications.

We advocate a clean interface between the server and the network in a publish/subscribe system. All our techniques leverage standard or readily available network substrates, e.g.: IP unicast, *content-based networks* [3], *distributed hash tables* such as [19], etc. Accordingly, our solutions can be deployed easily and quickly, without worrying about pushing complex application processing logic inside the network.

We support notification conditions with arbitrary and different radii. Subscriptions can be added and removed dynamically, and we handle the case where subscriptions with identical radius may need to notified differently because their different creation times can lead to different per-subscription states. We adhere to a strict interpretation of notification conditions; i.e., a subscriber is notified if and only if the incoming event causes a radius violation.

We also propose two extensions, including 1) a more flexible form of notification condition specifying when a subscription must be notified, when it must not be notified, and when it may be optionally notified; 2) relative notification conditions, where notification is needed when the difference between the current and the last-notified values, measured in relative terms such as percent change, exceeds the prescribed threshold.

Outline The rest of this paper is organized as follows. In Section 2, we describe our problem setting and semantics of value-based notification conditions precisely; we also warm up to our main discussion by presenting a series of less sophisticated solutions. Sections 3–6 present our solutions in a step-by-step manner. In Section 3, we tackle the first subproblem of handling subscriptions with different radii, assuming that no two subscriptions have the same radius. In fact, this solution works without the assumption, if we slightly relax the semantics of subscription creation to ensure that subscriptions with the same radius see an identical sequence of notifications. Without relaxing the subscription creation semantics, however, we need the solution to the second subproblem, described in Section 4, which handles subscriptions with same radius but different views of the data. Next, Section 5 describes how to combine the solutions to the subproblems in a single system, which provides full support for value-based notification conditions. Finally, Section 6 presents extensions of our techniques for flexible may-notify and must-notify conditions, and for relative notification conditions. In Section 7, we present a thorough experimental evaluation of our techniques. We discuss related work in Section 8 and conclude in Section 9.

2 Preliminaries

2.1 Semantics of Value-Based Notification

Consider a total of N subscriptions interested in tracking the value of the same data item over time. We denote the value of the data item at time t by $d(t)$. When a subscription is created, it is notified with the data value at the time of creation. Each subscription S_i specifies a *radius*, denoted by r_i. We notify the subscription with the current value of the data item if and only it has drifted by no less than r_i from the value last received by S_i.

Formally, at time t, the *center* of a subscription S_i, or its *last-notified value*, denoted $c_i(t)$, is the last value received by S_i at or prior to t. We call $(c_i(t) - r_i, c_i(t) + r_i)$ the *subscription interval* of S_i at t. The system maintains the invariant $d(t) \in (c_i(t) - r_i, c_i(t) + r_i)$ for any i and t (since the creation of S_i). If an update event causes $d(t)$ to fall outside S_i's subscription interval, we notify S_i, thereby *recentering* its subscription interval at $d(t)$.
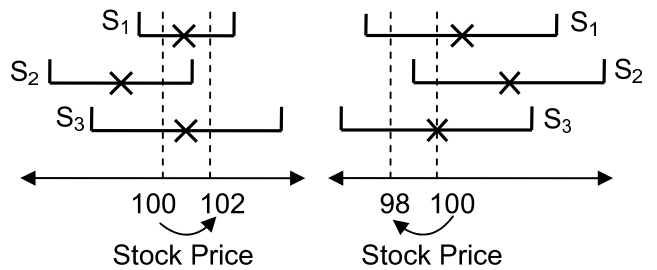

Figure 1: Example subscriptions.

The following two examples illustrate some of the intricacies that arise in supporting valued-based notification conditions.

Example 1 (Subscriptions with different radii). *Suppose there are three subscriptions interested in tracking the price of a stock, with radii 2, 4, and 6 respectively. Figure 1 (left) plots these subscriptions as intervals $(c_i(t) - r_i, c_i(t) + r_i)$, with their centers indicated by crosses. The only obvious constraint on these intervals is that all of them must be stabbed by $d(t)$. In the figure, for instance, a stock price change from 100 to 102 will affect the subscription with radius 4 but not the subscriptions with radius 2 or 6. Hence, there is no simple subsumption relationship implied by the lengths of subscription radii.*

Example 2 (Subscriptions with same radius). *Consider three subscriptions with the same radius, say 4. The subscriptions were created when the stock price was 101, 103, and 100 respectively, as depicted in Figure 1 (right). If the current stock price is 100 and it changes to 98, we would notify S_2 but not S_1 or S_3. In fact, it is not difficult to devise a sequence of update events such that the sequences of notifications received by these three subscriptions are completely disjoint from each other. For example, each update event can always drop the value just below the largest (but above the second largest) left endpoint of the current subscription intervals; this sequence will cause the subscriptions to be notified in a round-robin fashion, one at a time.*

2.2 Notification Dissemination

For each incoming event, the publish/subscribe system identifies the subset of subscriptions that need to be notified, and uses a network substrate to deliver notification messages to the subscribers over a wide-area network. We employ a network of *brokers*, each of which is responsible for a subset of the subscriptions. A broker forwards notifications to relevant subscriptions under its responsibility using whatever end-user delivery mechanism is suitable (e.g., IP, emails, instant messages). In this paper, we primarily concern ourselves with the network substrate spanning all brokers, and ignore the "last-hop" delivery because it mainly depends on subscribers' needs and is orthogonal to the design of the rest of the system. The most basic network substrate we can employ is the Internet, which supports IP unicast from a central subscription server to any individual broker.

We also consider a class of networks which we call *content-driven networks (CN)*.[1] Messages in CN do not specify any physical destination address; instead, they contain a list of attribute-value pairs. Destinations declare their interests with the CN as boolean

---

[1]Terms such as *content-based routing*, *content-based networking*, and *semantic multicast* capture similar concepts. We choose not to use these terms because they are often associated with specific projects and systems, e.g., [3, 2, 18]; we want to capture a broader class of systems with different designs and varying degrees of expressiveness.

predicates over contents of individual messages. The CN automatically routes messages based on their contents to destinations interested in them. Many networks can be classified as CN. In the following, we give a number of examples, with varying degrees of expressiveness in the predicates they support.

- A multicast network supporting multiple multicast groups can be seen a CN whose messages carry a group id attribute; destination interests, implied by group memberships, can be regarded as message predicates that select particular group ids.

- In a d-dimensional *content-addressable network (CAN)* [19], messages contain d numeric attributes, and CAN nodes correspond to orthogonal range predicates in the d-dimensional space. CAN automatically routes each message to the CAN node whose predicate is satisfied by the message content.

- A *prefix hash tree (PHT)* [6] is an overlay network that supports one-dimensional range searches in the domain of binary strings. Each PHT node corresponds to a binary string. PHT automatically routes each range search message to all nodes within the range. To see why PHT is a CN, we can regard a PHT node corresponding to string s as interested in all range search messages satisfying the predicate $(S_L \leq s) \wedge (s \leq S_R)$, where S_L and S_R are the two message attributes corresponding to the left and right endpoints of the search range.

- *Content-based network* [3], perhaps the most general incarnation of CN, supports messages with arbitrary attributes and destinations with interests expressed as arbitrary boolean predicates involving message attributes.

CN can directly support notification dissemination for stateless subscriptions (as long as the predicates defining the subscriptions are supported by the particular network used). We can simply inject each event into the CN, and it will deliver the event to all destinations interested in this event in an efficient manner.[2] Compared with unicast-based notification dissemination, CN offers better scalability, and incurs much less network traffic when lots of subscribers need to be notified.

However, CN does not directly support stateful subscriptions such as those with value-based notification conditions. The reason is that for stateful subscriptions, we cannot determine, just by examining the content of an incoming event message and a subscription definition, whether the subscription is affected by the event (i.e., whether the subscription needs to be notified because of the event). For subscriptions with value-based notification conditions, the missing information needed to make this determination is the current center of the subscription. Despite this difficulty, we want to develop techniques for handling stateful subscriptions using CN for notification dissemination, in order to leverage CN's scalability and in-network predicate matching capabilities. Our previous work [4] considered other types of stateful subscriptions (e.g., range-min) without notification conditions; this paper focuses on subscriptions with personalized notification conditions.

Besides unicast and CN, another possibility is to push application state and processing logic into the network substrate, so that it supports stateful subscriptions directly. Systems such as *SMILE* [12] take this approach. We, on the other hand, consciously take the simpler approach of relying on standard network substrates that do not need to support application state, thereby preserving a

---

[2]Note that in CAN and PHT, overlay nodes cannot specify their own interests; their predicates are chosen by the network. In these cases, to support subscriptions with arbitrary interests, we need to map subscriptions to appropriate overlay nodes, which serve as brokers responsible for forwarding events to them. We will see an example in Section 2.3.
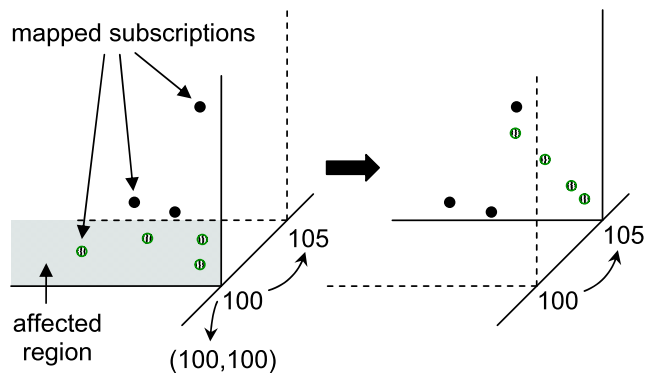

Figure 2: Value-based notification conditions over CAN.

clean, untangled interface between application and network. We believe this approach makes our system easier to deploy and maintain on a very large scale.

2.3 Alternative Approaches

Before describing our approach in detail, we warm up by presenting two alternatives that are more obvious and less sophisticated, followed by a discussion of some existing solutions [21, 20].

B-tree(lr)+Unicast: Server-Based Processing with Unicast Dissemination In a subscription server, we can store all subscriptions for the same data item in a standard B-tree by indexing the left and right endpoints of each current subscription interval (hence the name *B-tree(lr)*). In other words, at the current time t, the B-tree stores the points $c_i(t) - r_i$ and $c_i(t) + r_i$ for each subscription S_i. Suppose an update event $d(t') \rightarrow d(t)$ arrives, where t' is the time of the last update and $d(t')$ denotes the value before the current update. We look up $d(t)$ in the B-tree, and traverse leaf nodes towards $d(t')$. The subscriptions whose endpoints we encounter when traversing the leaf nodes are precisely those needing notification. To see why, note that if $d(t) > d(t')$, then these subscriptions are precisely those with $c_i(t') + r_i \leq d(t)$; if $d(t) < d(t')$, then these subscriptions are precisely those with $d(t) \leq c_i(t') - r_i$; in either case the new value has fallen out of the subscription interval.

Once identified, each affected subscription S_i is notified using unicast from the server. Since the notification recenters the subscription to $c_i(t) = d(t)$, to maintain the B-tree, we need to remove the old endpoints of S_i and insert the new ones $d(t) - r_i$ and $d(t) + r_i$. Recall that N denotes the total number of subscriptions. Let k denote the number of subscription affected by the incoming event. The cost of B-tree lookup and maintenance for the incoming event is $O(k \log_B N)$ I/Os, where B is the block size.

There are two problems with this approach. First, besides lookup, every incoming event requires deleting and reinserting each affected entry, which can have a high overhead if there are many affected subscriptions. Second, this approach enumerates the list of affected subscriptions and unicasts to each of them in turn. Because the server needs to send out these messages, it will experience heavy node stress and cause long notification delays for subscriptions that appear late in that list. The resulting network traffic will also be high. The server can batch all unicast messages destined to the same broker into one to save some overhead, but that one message still needs to enumerate all affected subscriptions hosted by the broker, so batching cannot avoid this problem fundamentally.

CN(lr): Serverless Processing and Dissemination by CN Although a value-based notification condition is stateful, we can make it stateless by instantiating the condition using the current subscription center. Specifically, each subscription S_i is defined by the predicate $(D \leq c_i(t) - r_i) \vee (c_i(t) + r_i \leq D)$, where D is the at-

tribute in the update event message that stores the new value of the data item. This technique allows us to use a content-driven network to automatically deliver an update event to affected subscriptions, without using any server. We call this approach *CN(lr)* because it is based on CN and it instantiates each subscription as a predicate involving the left and right endpoints of the current subscription interval.

As a concrete example, we show how to implement this approach using a specific CN: a two-dimensional CAN [19]. Each point (x, y) in this two-dimensional coordinate space represents an interval $[x, y]$. Each subscription S_i is mapped to the point $(c_i(t) - r_i, c_i(t) + r_i)$, as shown in Figure 2. Note that if a point p_1 is located to the northwest of another point p_2, then p_1's interval contains p_2's interval. Hence, all subscriptions in Figure 2 (left) are to the northwest of point $(100, 100)$, where 100 is the current data value. Suppose that an update event changes the value to 105. The affected subscriptions are those not containing the new value, shown in the shaded region.

CAN can be easily extended to support routing of messages to any designated hyperrectangle in its coordinate space, as is done in *Meghdoot* [11]. Specifically, we partition the two-dimensional space into zones based on load-balancing criteria, and assign each zone to a CAN node. Each CAN node has knowledge of only its neighbors and can route messages only to them. We use CAN nodes as brokers, each of which is responsible for all subscriptions that fall within its assigned zone. When an update event $100 \rightarrow 105$ arrives, we first send it to $(100, 100)$ using standard CAN routing, and then forward it north and west through brokers until the shaded region is completely covered.

The problem here, however, is that each affected subscription S_i will be recentered and therefore must change it coordinate to $(d(t) - r_i, d(t) + r_i)$. This movement of subscription points is illustrated in Figure 2 (right). Unfortunately, this operation is expensive, because these subscriptions will change brokers and potentially trigger load rebalancing.

Although there are other alternatives to CAN as the CN substrate (e.g., PHT and content-based network), the efficiency problem caused by subscription recentering is universal. Even for CN substrates that do not require subscriptions to switch brokers (e.g., content-based network), changes of subscription intervals must be propagated through the network to update routing tables, which is also expensive. The fundamental problem is that while we have reformulated a stateful subscription to use a stateless definition, this definition depends on a dynamic property of the subscription (current center, in this case). Whenever this property changes, we may incur a significant overhead.

Existing Solutions   Shah et al. [21, 20] have investigated the problem of disseminating dynamic data in a network with the goal of meeting coherency constraints. The goal is to maximize *fidelity*, which is the percentage of time that the coherency constraints are met. Translated to the publish/subscribe setting, the only semantic requirement is that the value $c_i(t)$ at the subscriber S_i should lie within $(d(t) - r_i, d(t) + r_i)$. However, a subscription may receive an update even if the updated value does not fall outside the current interval. Furthermore, a subscription may even receive a value that never existed, as long as the resulting interval (recentered at this value) still contains the true value of the data item.

Their solution uses a customized dissemination tree with smaller radii serving larger ones. An event is disseminated starting at the root, using one of the following techniques:

- At the server, determine the largest radius affected (r_{max}) and send the event to all S_i such that $r_i \leq r_{max}$ [21].

- *Distributed dissemination*, where a parent S_i sends an event to its child S_j if $|c_j(t) - c_i(t)| > r_j - r_i$ [21]. Both this technique and the first one may result in unnecessary notifications which is forbidden by our stricter notification semantics.

- Enforce *dependent ordering* by sending *pseudo-values* [20]. In this case, subscriptions may receive values that have never been published, which violates our publish/subscribe semantics.

Our precise notification semantics in Section 2.1 require that S_i should not receive any updates that do not fall outside the current subscription intervals. Furthermore, a subscription should only receive "true" values, i.e., values that have been produced by the publisher. In this paper, we show that we can handle the precise semantics efficiently. We also discuss, in Section 6, how to relax the semantics to allow additional may-notify (loose) conditions in subscriptions. Another notable difference is that we use only standard network substrates, instead of using a customized dissemination tree that introduces complexity by adding application-specific logic inside the network,

3   Subscriptions with Different Radii

As the first step towards a complete solution, we assume in this section that all subscriptions with the same radius have identical centers. Under this assumption, by assigning subscriptions with the same radius to the same broker, we can effectively treat them as a single subscription, so all subscriptions now have unique radii.

This assumption is actually not as restrictive as it may appear at first glance. We can ensure that this assumption holds by enforcing the simple rule that a new subscription created at time t of the same radius r_i as some existing subscription S_i is sent an initial data value of $c_i(t)$ instead of $d(t)$. This alternative semantics of subscription creation may be acceptable for some applications because 1) $c_i(t)$ was a true value of the data item at some point in the past, and 2) the distance between the true value $d(t)$ and the value $c_i(t)$ now seen by the new subscription is guaranteed to be less than r_i. Nevertheless, for applications that require more precise semantics, we will tackle the general case of multiple subscriptions with the same radius having different current centers in Sections 4 and 5.

3.1   Scan(r)+CN(r): Towards Radii Indexing

The approaches introduced in Section 2 have a glaring problem with indexing subscriptions using properties that are dynamic (for both server processing and network dissemination). The solution that comes to mind is to index subscriptions by their radii, as the radii do not change. However, it is unclear how to process an incoming event efficiently using such an index, because whether an event affects a subscription depends not only on its radius but also its current center, which is no longer indexed.

As a first step towards indexing by radii, we propose a simple technique called *Scan(r)*, which can produce not only a list of affected subscriptions (intended for unicast), but also a semantic description of affected subscriptions (intended for CN). Scan(r) maintains all subscription in a sorted order of increasing radii. Upon receiving an event, Scan(r) makes a linear scan and identify the subscriptions affected by the event. As it proceeds, Scan(r) can easily produce as its output a list of m *affected radius ranges* of the form [R_LOW, R_HIGH], which indicates that all subscriptions with radius in this range is affected by the event. In Example 1, for instance, event $100 \rightarrow 102$ produces a single affected radius range $[4, 4]$. Scan(r) takes $O(N/B)$ I/Os, and does not need expensive updates to recenter subscriptions, as radii remain unchanged.

On the network side, we can "index" subscriptions by radii as well. Each subscription S_i is identified by its radius r_i, or more

precisely, by predicate R_LOW $\leq r_i \leq$ R_HIGH. We use a content-driven network to manage these subscriptions, and call this approach CN(r). Because radius is a static subscription property, CN(r) does not have CN(lr)'s problem of huge reorganization costs during subscription recentering. The list of m affected radius ranges, computed by Scan(r), works perfectly with CN(r). We inject each affected radius range as a message with attributes R_LOW and R_HIGH (in addition to the new data value) into CN(r), and CN(r) will automatically route it all subscriptions with predicates matching this message. Any incarnation of CN that supports range lookups (e.g., PHT or content-based network) can be plugged in.

This trick is an example of *reformulation* [4], the idea of reformulating events as messages concisely describing subsets of affected subscriptions so that stateful subscriptions can be handled by network substrates that only directly support stateless subscriptions. In our case, Scan(r) uses the subscription state it maintains to reformulate an incoming event into messages that can be directly handled by CN(r). Note that for CN(r) to outperform unicast, we need m, the number of affected radius ranges, to be much less than k, the number of affected subscriptions. In our experiments (Section 7), we have found m to be at least an order of magnitude smaller than k. The intuition behind this observation is that subscriptions with similar radii tend to be affected in batches. Even if two adjacent radii are separated by an update that cuts across them, there is a high likelihood that a future update will bring their centers very close together.

3.2 B^A-tree(r): Augmented B-Tree on Radii

We now present a data structure called B^A-*tree(r)* capable of computing affected radius ranges much more efficiently than Scan(r). Each incoming event $d(t') \to d(t)$ is processed by B^A-tree(r) as a *delta* $\Delta_i = d(t) - d(t')$. We start with a B-tree that indexes all subscriptions by radii. The tree topology is static in the absence of subscription insertions and deletions. We augment the B-tree with additional fields that allow us to track the subscriptions centers. To maintain this information efficiently, we update a subtree only when some, but not all, subscriptions in the subtree need to move. If all or no subscriptions in the subtree are affected, the entire subtree can be marked as updated simply by updating the root of the subtree. To support this nontrivial bookkeeping, we augment each index entry i with 8 following fields. An example B^A-tree(r) is shown in Figure 3.

- $r_{\min}[i]$ and $r_{\max}[i]$ track the smallest/largest subscription radius in the subtree rooted at i. A leaf node entry has $r_{\min}[i] = r_{\max}[i] = r$, where r is the radius of the subscription indexed by this entry.

- $L_{\min}[i]$ and $R_{\min}[i]$ track the smallest (in absolute value) negative/positive delta that will affect at least one subscription in the subtree rooted at i. Their values are accurate as of the last delta that affects at least one (but not all) subscription in the subtree rooted at i's parent.

- $L_{\max}[i]$ and $R_{\max}[i]$ track the smallest (in absolute value) negative/positive delta that will affect every subscription in the subtree rooted at i. Their values are accurate as of the last delta that affects at least one (but not all) subscription in the subtree rooted at i's parent. For a leaf node entry, $L_{\min}[i] = L_{\max}[i]$ and $R_{\min}[i] = R_{\max}[i]$. For all nodes, initially $L_{\min}[i] = R_{\min}[i] = r_{\min}[i]$ and $L_{\max}[i] = R_{\max}[i] = r_{\max}[i]$.

- $\gamma[i]$ is the *drift factor*, initially set to 0. It accumulates a sequence of consecutive deltas that have reached this index entry but not yet been propagated down (because no subscription in the subtree is affected by any delta in the sequence).
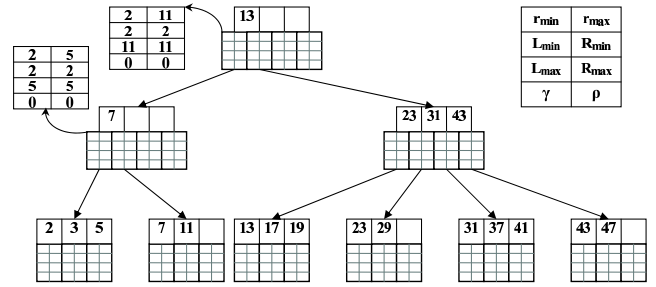

Figure 3: Example B^A-tree(r).

- $\rho[i]$ is the *reset bit*. When set to 1, it indicates that all subscriptions in this subtree were affected by the last delta that affected at least one subscription in this subtree.

Let $anc(i)$ be the set of ancestor index entries of i in the tree. There are two cases for how the reset bit is used to implicitly keep track of subscriptions in subtrees which are not directly updated.

- Case 1: For all $j \in anc(i)$, $\rho[j] = 0$. In this case, let $d = \sum_{k \in anc(i)} \gamma[k]$. Then, the *current* values of $L_{\min}[i]$, $R_{\min}[i]$, $L_{\max}[i]$, and $R_{\max}[i]$ should be $L_{\min}[i] + d$, $R_{\min}[i] - d$, $L_{\max}[i] + d$, and $R_{\max}[i] - d$, respectively.

- Case 2: There exists $j \in anc(i)$ such that $\rho[j] = 1$ and $\rho[k] = 0$ for all $k \in anc(j)$. Here, let $d = \gamma[j] + \sum_{k \in anc(j)} \gamma[k]$. Then, the *current* values of $L_{\min}[i]$, $R_{\min}[i]$, $L_{\max}[i]$, and $R_{\max}[i]$ are effectively $r_{\min}[i] + d$, $r_{\min}[i] - d$, $r_{\max}[i] + d$, and $r_{\max}[i] - d$, respectively.

Lookup (and Bookkeeping) On an incoming event Δ, we look up the B^A-tree(r) for the set of affected radius ranges by invoking LOOKUP($\mathcal{T}, \Delta, 0$), where $\mathcal{T}$ denotes the tree root. Algorithm 1 shows this procedure. Note that this "lookup" also updates the tree by performing necessary bookkeeping to track subscription movements; however, the structure of the tree remains unchanged.

During lookup, we consider three cases at each index entry of the node being visited. If the entire subtree is unaffected (Lines 7–12), we basically update the drift factor and do not recurse down. If the subtree is partially affected (Lines 13–15), we have to recurse down the subtree. If a previous lookup has set the reset bit for this entry, it implies that the child node has stale information. Hence, we invoke LOOKUP recursively with reset flag set to true. This flag indicates to the child index entries that they need to reset their information. If there is an accumulated drift factor, we need to add it to Δ before the recursive lookup. After the recursion returns, we can change the reset bit and drift factor to 0 for this index entry. Finally, if the subtree is completely affected (Lines 16–19), we set the reset bit for that index entry, change the drift factor to 0, and generate the affected radius range, and continue without recursing down. Resets propagated down by parents are processed in Lines 4–6. We incrementally compute the information to be returned to the parent in Lines 20–23.

It is not difficult to see that the I/O cost of lookup is $O(\min(m \log_B N, N/B))$, where m is the number of affected radius ranges it outputs. Lookup avoids visiting an entire subtree if the subtree is completely unaffected or entirely affected. Hence, to compute each affected radius range, lookup does not need to examine any nodes other than those on the paths from the root to the two endpoints of the range. Bookkeeping piggybacks on lookup and incurs no additional I/Os. With minor modifications, lookup can also produce the complete list of k affected subscriptions (for unicast) in $O(\min(m \log_B N + k/B, N/B))$ I/Os.

Producing Maximal Radius Ranges To ensure that m, the number of affected radius ranges, is as small as possible, we can mod-

Algorithm 1: Lookup algorithm for B^A-tree(r).

```
1  LOOKUP(node n, float Δ, bool ρ) begin
2      ⟨L_min^ret, R_min^ret, L_max^ret, R_max^ret⟩ ← ⟨∞, ∞, 0, 0⟩;
3      foreach index entry i of n do
4          if ρ = 1 then
5              ⟨L_min[i], R_min[i], L_max[i], R_max[i]⟩ ←
                   ⟨r_min[i], r_min[i], r_max[i], r_max[i]⟩;
6              ρ[i] ← 1;  γ[i] ← 0;
7          if −L_min[i] < Δ < R_min[i] then
                   // entire subtree unaffected
8              L_min[i] ← L_min[i] + Δ;
9              R_min[i] ← R_min[i] − Δ;
10             L_max[i] ← L_max[i] + Δ;
11             R_max[i] ← R_max[i] − Δ;
12             γ[i] ← γ[i] + Δ;
13         else if −L_max[i] < Δ < R_max[i] then
                   // subtree partially affected
14             ⟨L_min[i], R_min[i], L_max[i], R_max[i]⟩ ←
                   LOOKUP(child[i], γ[i] + Δ, ρ[i]);
15             ρ[i] ← 0;  γ[i] ← 0;
16         else
                   // entire subtree affected
17             ⟨L_min[i], R_min[i], L_max[i], R_max[i]⟩ ←
                   ⟨r_min[i], r_min[i], r_max[i], r_max[i]⟩;
18             ρ[i] ← 1;  γ[i] ← 0;
19             output ⟨r_min[i], r_max[i]⟩;
20         L_min^ret ← min(L_min^ret, L_min[i]);
21         R_min^ret ← min(R_min^ret, R_min[i]);
22         L_max^ret ← max(L_max^ret, L_max[i]);
23         R_max^ret ← max(R_max^ret, R_max[i]);
24     return ⟨L_min^ret, R_min^ret, L_max^ret, R_max^ret⟩;
25 end
```

ify Algorithm 1 to merge adjacent radius ranges with no unaffected subscriptions in between. Merging can be implemented in a streaming fashion because LOOKUP produces the ranges in order. The idea is to delay the output of affected ranges and instead keep a running range that can be extended until we encounter an unaffected leaf/subtree (or the end of processing); at this point, we can output the running range. This extension produces maximal radius ranges as output and incurs no additional cost.

Inserting and Deleting Subscriptions Subscriptions can be inserted and deleted using the standard B-tree procedures, with minor modifications to maintain the augmented fields. The I/O cost for these operations remains bounded by $O(\log_B N)$.

3.3 Notification Dissemination

Like Scan(r), B^A-tree(r) works with both unicast and CN(r) for notification dissemination. For unicast, B^A-tree(r) produces a list of k affected subscriptions; for CN(r), it produces a list of m maximal affected radius ranges, which are reformulated as messages that can be routed by a vanilla CN implementation that supports range searches. Since usually $m \ll N$, B^A-tree(r) should perform much better than Scan(r).

4 Subscriptions with Same Radius

We now come to the second subproblem as outlined in Section 1. Consider a set of N_r subscriptions with the same radius r. When a new subscription is created, it is centered at the current data value. Thus, subscriptions with the same radius may have different centers and receive different notification sequences. In this section, we present our solution, starting with two strawman solutions.

4.1 B-tree(c)+Unicast and CN(c): Strawman Solutions

B-tree(c)+Unicast While B-tree(lr)+unicast from Section 2.3 can be applied here, we can improve it by exploiting the fact that

all subscriptions have the same radius for this subproblem. Instead of indexing two endpoints, we index just the current center of each subscription in a standard B-tree (hence the name B-tree(c)). On an increasing update event $d(t') \to d(t)$ where $d(t) > d(t')$, we can compute the list of affected subscriptions for unicast simply by looking up subscriptions centered within $(d(t') - r, d(t) - r]$ (the case for decreasing updates is symmetric). Then, we delete these subscriptions and reinsert them into the tree at $d(t)$, because they are now recentered at this value. Deleting and reinserting them one at a time would result in $O(k_r \log_B N_r)$ I/Os, where k_r is the number of affected subscriptions. Instead, noting that affected subscriptions are to be deleted from a contiguous range of the B-tree and reinserted into another contiguous one, we can perform the deletions and insertions in batches, giving an overall I/O cost of $O(\log_B N_r + k_r/B)$ per event including both output computation and tree maintenance.

CN(c) An alternative, analogous to CN(lr) from Section 2.3, is to use a content-driven network to manage subscriptions identified by their centers (hence the name *CN(c)*). On each increasing update $d(t') \to d(t)$, we simply inject a message with attributes $D' = d(t') - r$ and $D = d(t) - r$ into the CN, which automatically route it to affected subscriptions, whose centers satisfy $D' < c_i(t) \le D$ (the case for decreasing updates is analogous). This approach does not require a server.

The main problem of CN(c), similar to CN(lr) and indeed common to all approaches that "index" dynamic subscription properties, is that recentering of affected subscriptions causes expensive maintenance overhead. In the case of CN(c), moving subscription centers lead to constant relocation of subscription and/or updates to routing information within the network. In Section 3, we were able to avoid this problem by indexing static radii. However, for subscriptions with the same radius, what static property can we use? The remainder of this section presents a solution.

4.2 Static Circular Ordering

On an incoming event, some subscriptions may be affected and need to be notified and recentered. Surprisingly, it turns out that while the subscription centers can move around, the relative ordering of these centers, when mapped onto a circle (we call this the *circular ordering*), remains static. Moreover, a newly created subscription does not break the circular ordering of existing subscriptions, but simply adds a new entry at some position in the ordering (similarly for removal of a subscription).

To explain the static nature of circular ordering, we start by proving an interesting property of subscription centers. It is clear that since all centers lie within a distance of r from $d(t)$, the distance between any two centers must be less than $2r$. We proceed to prove a stronger result.

Lemma 1. *The distance between any two subscription centers at any given time is less than the radius r; that is, $|c_i(t) - c_j(t)| < r$ for any i and j.*

Proof. Consider two subscriptions S_i and S_j, inserted at times t_i and t_j respectively. If $t_i = t_j$, then $c_i(t) = c_j(t)$ for all t and we are done. Let $t_j > t_i$. At time t_j, we have $c_j(t_j) = d(t_j)$ and $|c_i(t_j) - d(t_j)| < r$. Hence, $|c_i(t_j) - c_j(t_j)| < r$.

Now, consider an incoming event at some later time $t_k > t_j$. If this event affects both S_i and S_j, we have $c_i(t_k) = c_j(t_k) = d(t_k)$, so $|c_i(t_k) - c_j(t_k)| = 0 < r$. If this event affects neither of the two subscriptions, their centers remain unchanged, and so is the distance between them. Finally, if the event affects just one of them, say S_i, we have $c_i(t_k) = d(t_k)$ and $|c_j(t_k) - d(t_k)| < r$. Hence, $|c_i(t_k) - c_j(t_k)| < r$. $\quad\square$
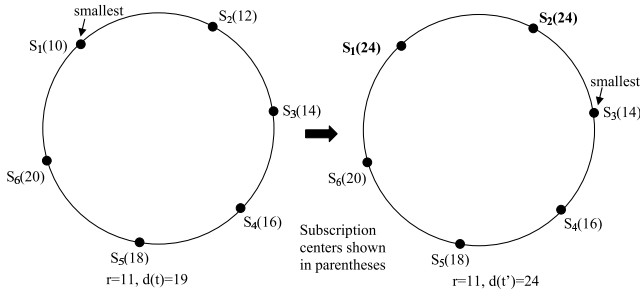
Figure 4: Static circular ordering of subscriptions.

Figure 5: Example B^W-tree(lid).

Theorem 1. *Suppose that at time t, all subscriptions are arranged in increasing order of their centers. No future event or subscription insertion/deletion can change the circular ordering of these subscriptions.*

Proof. Case I (effect of events): An event that affects no subscriptions does not change any centers and cannot break their ordering. Consider an increasing update event $d(t_1) \rightarrow d(t_2)$ that affects at least one subscription. Such an event affects a set of k_r subscriptions with centers in the range $(d(t_1) - r, d(t_2) - r]$. However, note that there exists no subscription with a center of $d(t_1) - r$ or less; therefore, this event affects subscriptions with the k_r smallest centers. The new center for these subscriptions is $d(t_2)$. This center is at a distance of no less than r to the right of the smallest subscription center, because the right endpoint of that subscription, denoted S_0, was violated. However, from Lemma 1 we know that all other subscriptions are centered at a distance of less than r to the right of S_0's center. Hence, all affected subscriptions, which were those with smallest centers, now effectively become the subscriptions with largest center, and the circular ordering is not violated. Similarly, on a decreasing event, a set of subscriptions with largest centers effectively become those with the smallest center, preserving the circular ordering.

Case II (effect of subscription insertion/deletion): A subscription insertion or deletion at time t does not modify the current center of any existing subscription. Therefore, existing subscriptions retain their circular ordering. Insertion or deletion only changes the immediate neighbor for each of the two subscriptions whose centers are the closest to and on either side of point $d(t)$ in the circular ordering. □

To illustrate the static nature of circular ordering, Figure 4 shows six subscriptions of radius 11, arranged in increasing order of their centers, clockwise in the circle. If an update event of $19 \rightarrow 24$ arrives, the two subscriptions with the smallest centers (S_1 and S_2) are affected and are assigned the new center of 24. The circular ordering of subscriptions is unbroken, with S_3 now being the subscription with the smallest center (14).

4.3 B^W-tree(lid): Circular Augmented Weight-Balanced B-Tree on Labels

To index a set of subscriptions with the same radius, we assign an increasing, unique number to each subscription in the order of its current center. We call this number the *label* of the subscription. By Theorem 1, the circular ordering of labels does not change with data update events, even though centers can move. Also, as we have seen in the previous section, the set of affected subscriptions can always be expressed as a single range in the circular subscription space. Since the numeric label space is linear, a range in the circular space may be either a single interval (e.g., $[lid_1, lid_2]$) or the union of two intervals (e.g., $(-\infty, lid_2] \cup [lid_1, +\infty)$) on nu-
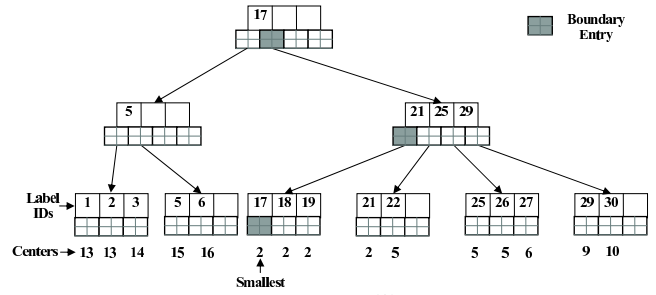
meric labels; in either case, two labels are needed to encode the range. Our goal is develop a data structure that is easy to maintain and can help us efficiently compute the two labels encoding the affected subscription range.

Our data structure is called B^W-*tree(lid)*, for *circular augmented weight-balanced B-tree on labels*. We first describe the basic data structure, and then discuss the modifications needed to maintain the labels in the face of subscription insertions and deletions. We start with a B-tree that indexes subscriptions by their labels. Every index entry i is augmented with the following:

- $c_{\min}[i]$ is the smallest subscription center in i's subtree.
- $c_{\max}[i]$ is the largest subscription center in i's subtree.
- $\rho[i]$ is a *reset bit* indicating that all subscriptions in i's subtree were affected by a single update and therefore have identical centers.

Over time, as the data item of interest is updated, any subscription can have the smallest center. We keep track of the subscription currently with the smallest center by remembering its label, called the *boundary label*. The index entries lying on the path from the root to the boundary label are called *boundary entries*. For an internal-node boundary entry, we maintain two sets of $c_{\min}$ and $c_{\max}$, for the two subranges separated by the boundary label. Figure 5 shows an example B^W-tree(lid). Subscriptions are stored in leaves in the increasing order of their centers, starting with the boundary label and wrapping around at the end.

To ensure efficient lookup and maintenance, we use an idea similar to B^A-tree(r) in Section 3: If subscriptions in a subtree are either all affected or all unaffected, we do not enter the subtree, but instead record the effect in the augmented fields of the subtree root. Recall from Section 4.2 that an increasing update event $d(t') \rightarrow d(t)$ affects all centers no greater than $d(t) - r$ (starting with the current boundary label) and relocates them to $d(t)$, which will become the largest centers located immediately before the new boundary label. To this end, we only need to take two paths down the B^W-tree(lid): P_1, towards the current boundary label, and P_2, towards the smallest center greater than $d(t) - r$. In this process, for each node on P_1 (and P_2), we simply go through each index entry j located to the right (and left, respectively) of the entry leading to the next node on the path, and set $\rho[j]$ and update $c_{\min}[j]$ and $c_{\max}[j]$ to $d(t)$. This method avoids updating subtrees that are completely affected. Maintenance of the boundary label and boundary entries also can be done during this process; we omit the details. Decreasing update events are handled similarly. Note that the entire process involves no changes to the tree structure.

Lookup on B^W-tree(lid) returns two subscription labels encoding the range of affected subscriptions in the circular subscription space. The overall I/O cost, including any maintenance overhead, is $O(\log_B N_r)$ per event.

Inserting/Deleting Subscriptions A new subscription may be added between two adjacent existing subscriptions in the circular

ordering. If there is a gap in the label space, a new label can directly be assigned to the new subscription. Otherwise, we need to relabel some existing subscriptions to make space for the new subscription.

A simple scheme such as relabeling all elements equally spaced over the label domain suffers from the problem of frequent relabeling, especially for skewed insertion patterns where new subscriptions are inserted repeatedly into the smallest gap. This *order maintenance* problem has been studied extensively in algorithms and database literature. We adopt the I/O-efficient relabeling scheme used in *W-BOX* [22], which was inspired by Dietz [9]. The relabeling scheme is easily accommodated in B^W-tree(lid) by using a weight-balanced B-tree [1] instead of a standard B-tree. The insertion cost (including relabeling) is $O(\log_B N_r)$ I/Os, amortized. Further improvements may be possible using alternative relabeling schemes such as [10], which bounds the number of relabelings per insertion by a constant.

4.4 Notification Dissemination

We use a content-driven network that manages subscriptions identified by their labels; we call this approach *CN(lid)*. For each incoming event, we use B^W-tree(lid) to compute the two labels encoding the range of subscriptions to be notified, and inject into CN(lid) a single message containing these two labels as attributes. As before, any incarnation of CN supporting range queries is able to efficiently route this message to relevant subscriptions.

As discussed in the previous section, although events do not change subscriptions labels, we may need to occasionally relabel subscriptions in case we run out of labels when adding new subscriptions. For CN(lid), details and costs of relabeling depend on the particular CN used. Our implementation of CN(lid) in experiments, for example, uses a distributed B-tree. A relabeling operation, based on the W-BOX relabeling scheme we use (Section 4.3), can be concisely represented as a range of labels to be relabeled and the magnitude of shift, which is easily supported by our distributed B-tree. Relabeling does not alter routing paths because the circular ordering of subscriptions remains unchanged during relabeling. Only routing information covering relabeled ranges need to be updated. We investigate the overhead of relabeling in CN(lid) experimentally in Section 7.2.2, and show that it is low even at high subscription creation rates.

Alternatively, we can use unicast for notification dissemination in conjunction with B^W-tree(lid). The lookup procedure of B^W-tree(lid) can be modified to generate the list of affected subscriptions instead of a label range, which would take $O(\log_B N_r + k_r/B)$ I/Os, where k_r is the number of affected subscriptions.

5 Putting the Pieces Together

We now discuss how to support the most general case where subscriptions have arbitrary radii, and subscriptions with the same radius can have different centers. Consider the set of all subscriptions with the same radius. An event can affect (subscriptions with) this radius in three ways: 1) The radius can be completely affected, i.e., the event affects all subscriptions with this radius; 2) the radius can be completely unaffected; or 3) the radius can be partially affected.

The basic idea is to use a two-layer system. A *primary server* generates ranges of radii that are either completely or partially affected. A set of *secondary servers* are responsible for individual radii. Conceptually, these servers are organized as a CN(r) as in Section 3, where secondary servers are indexed by their designated radii. For each radius, its designated secondary server forms a CN(lid) with other brokers in the system, where each subscription is identified by its label, as in Section 4. We call this two-

level network substrate *CN(r,lid)*. Although we have described CN(r,lid) conceptually as consisting of many content-driven networks, in practice they can be implemented by one network, with additional attributes encoding message types and additional predicates that distinguish them.

With CN(r), the primary server can send an update event to all affected secondary servers using messages containing the affected radius ranges. In order to compute these radius ranges, the primary server uses a two-level B-tree called the *B^2-tree(r,lid)*, composed of one upper-level B^A-tree(r) and a B^W-tree(lid) for each radius at the leaves. The processing cost for the primary server is $O(q \log_B N' + s \log_B N)$ per event, where q is the number of completely affected radius ranges, N' is the number of unique radii, s is the number of partially affected radii, and N is the total number of subscriptions. It may appear that the primary server needs to only maintain the B^A-tree(r). However, when a radius r is partially affected, the B^A-tree(r) would not be able to adjust L_{min}, L_{max}, R_{min}, and R_{max} fields for r because their values depend on the new smallest and largest centers among subscriptions with radius r, and this information is only available in the B^W-tree(lid) for r.

Each secondary server maintains a B^W-tree(lid) for each radius r it is responsible for. Upon receiving from the primary server an event affecting r, the secondary server uses the B^W-tree(lid) for r to compute a label range identifying the set of affected subscriptions with radius r. This label range is injected into CN(lid) and routed to affected subscriptions.

6 Extensions

6.1 Generalizing Notification Semantics

We can extend our subscription language and semantics to allow for a range of radii $[r^-, r^+]$ specifying the acceptable range of deviation from the last-notified value. For a subscription S_i, the radius range $[r_i^-, r_i^+]$ replaces the rigid r_i threshold. We refer to r_i^- as the *may-notify* radius, and r_i^+ as the *must-notify* radius. S_i *must* be notified if $|c_i(t) - d(t)| \geq r_i^+$. In addition, S_i *may* be notified if $|c_i(t) - d(t)| \geq r_i^-$. Finally, S_i should definitely *not* be notified if $|c_i(t) - d(t)| < r_i^-$. The additional flexibility provided by this type of notification conditions opens up optimization possibilities.

For subscriptions with unique, different radii (as considered in Section 3), we seek to reduce the number of affected radius ranges. If two or more radius ranges are separated only by subscriptions that may be notified, we can concatenate these ranges into a single radius range. Disseminating a lesser number of larger radius ranges means that we pay the network overhead of reaching a radius range, less often. This reduces the overall network traffic. To this end, we change the B^A-tree(r) to sort all subscriptions by their may-notify radii, and set $r_{min} = r_i^-$ and $r_{max} = r_i^+$ for S_i's leaf node entry (as opposed to setting both to r_i in Section 3). Generation of radius ranges works the same way as in Algorithm 1. However, we do not output any isolated radius range consisting entirely of subscriptions that may, but do not have to, be notified.

For subscriptions with the same may-notify radii and the same must-notify radii (as considered in Section 4), we seek to notify all subscriptions that may be notified, as long as at least one subscription must be notified. The underlying intuition is that a notification will cause all notified subscriptions to have the same center, which means that they will behave identically from now onwards and therefore can be effectively treated as a single subscription, thereby reducing the number of unique subscriptions in the system. We omit the details of modifying B^W-tree(lid) to implement this heuristic. Note that with this heuristic, Theorem 1 still holds because notifications are only made with respect to radius $r^- \leq r^+$,

and notifications are performed only when some r^+ is violated.

6.2 Relative Notification Conditions

We now consider another extension of our subscription language to support *relative notification conditions*, where the radius is specified not in absolute terms but relative to the last-notified value. We discuss two types of such notifications.

The first type uses an *additive-relative radius*, which forms the subscription interval by subtracting from and adding to the subscription center. For instance, users may be interested in receiving a stock price update if the current price has deviated from the last received price by at least 10%. Formally, the subscription interval is given by $(c_i(t) - c_i(t) \cdot p_i, c_i(t) + c_i(t) \cdot p_i)$, where $p_i > 0$ is the *additive-relative radius* parameter. The subscription needs to be updated if $|d(t) - c_i(t)| \geq p_i \cdot c_i(t)$. In the case of subscriptions with unique, different p_i's, B^A-tree(r) from Section 3 works similarly, using p_i instead of r_i. The main change is that if an entire subtree rooted at index entry j is affected, we have to set $L_{\min}[j]$ and $R_{\min}[j]$ values to $p_{\min}[j] \cdot d(t)$ instead of $r_{\min}[j]$ (likewise for $L_{\max}[j]$ and $R_{\max}[j]$). In the case of subscriptions with the same p, however, we cannot directly use B^W-tree(lid). The reason is that Theorem 1 no longer holds due to the fact that the radius is dependent on the current center. Nevertheless, we can still use the B^A-tree(r)-based solution if we relax the subscription creation semantics to ensure that subscriptions with the same radius have the same center, as discussed in the beginning of Section 3.

The second type of relative notification conditions uses a *multiplicative radius*, which forms the subscription interval by dividing and multiplying the subscription center. Formally, the subscription interval is given by $(c_i(t)/f_i, c_i(t) \cdot f_i)$, where $f_i > 1$ is the *multiplicative radius* parameter, and we assume that the data value of interest is always positive. This type of relative notification conditions can be handled by transforming the values by taking their logarithms. In the logarithmic domain, the subscription interval becomes $(\log c_i(t) - \log f_i, \log c_i(t) + \log f_i)$, which is just a standard, non-relative value-based notification condition with radius $\log f_i$. Therefore, we can directly apply the techniques developed in the previous sections in the logarithmic domain.

7 Evaluation

7.1 Metrics, Workload, and Setup

For the *all-rad* case, where subscriptions can have arbitrary radii, and subscriptions with the same radius can have different views of the data, we we have implemented the following techniques for server-side processing:

- B-tree(lr): A simple B-tree constructed on left and right endpoints of current subscription intervals (Section 2.3).
- B^2-tree(r,lid): Our two-level data structure indexing subscriptions first by radius, and then by labels (Section 5).

For the *diff-rad* case (Section 3), where subscriptions with the same radius have identical view of the data, i.e., unique subscriptions all have different radii, we have implemented the following techniques (in addition to B-tree(lr), which is also applicable in this case):

- Scan(r): Storing subscriptions sorted by radii, and performing a linear scan to output radii or radius ranges (Section 3.1).
- B^A-tree(r): Our augmented B-tree constructed on subscription radii (Section 3.2).

Finally, for the *same-rad* case (Section 4), where subscriptions have the same radius but different views of the data, we have implemented the following techniques (in addition to B-tree(lr), which is also applicable in this case):

- B-tree(c): A simple B-tree constructed on subscription centers for subscriptions with the same radius (Section 4.1).
- B^W-tree(lid): Our augmented weight-balanced B-tree on the subscription labels (Section 4.3).

Although our data structures have been designed to be I/O-efficient, our current implementation uses main memory. The server processing time we measure in experiments mostly reflect CPU and memory access costs.

For the network substrate, we have implemented a simulator for large-scale networks. The first phase of network simulation generates application-level routing traces, which are further analyzed by a second phase to produce detailed costs. The second phase performs a link-level simulation using a topology produced by *INET* [5], a generator of Internet-like network topologies. We consider both unicast and content-driven network (CN) for notification dissemination. The CN we have implemented is structured as a distributed B-tree with support for load balancing. The root of the B-tree is hosted by a server, and the nodes are hosted by brokers. We experiment with the following dissemination methods:

- Unicast from the central server. This method can be used in conjunction with B-tree(lr), B^2-tree(r,lid) (for all-rad), Scan(r) (for diff-rad), B^A-tree(r) (for diff-rad), B-tree(c) (for same-rad), and B^W-tree(lid) (for same-rad).
- CN(lr), serverless content-driven network where subscriptions are identified by the left and right endpoints of their intervals (Section 2.3). This method works in all three cases.
- CN(c), serverless content-driven network where subscriptions are identified by their current centers (Section 4.1). This method works for same-rad.
- CN(r), content-driven network where subscriptions are identified by their radii (Section 3.3). This method works in conjunction with Scan(r) and B^A-tree(r) for diff-rad.
- CN(lid), content-driven network where subscriptions are identified by their labels (Section 4.4). This method works in conjunction with B^W-tree(lid) for same-rad.
- CN(r,lid), two-level content-driven network (Section 5). This method works in conjunction with B^2-tree(r,lid) for all-rad.

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 duration 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*, which measures the total number of messages sent between overlay nodes (for CN), or between the server and brokers hosting subscriptions (for unicast). 2) *Number of IP message hops*, which measures the number of hops over IP-level links. An overlay hop may involve traversing a number of IP-level links on its path. 3) *Network traffic*, which measures the total number of bytes transferred between overlay nodes (for CN) or between the server and brokers (for unicast). 4) *Maximum node stress*, which measures the number of messages originating from a node. The maximum node stress for an event is the highest node stress among all nodes while processing that event. Besides subscription processing and notification dissemination, these metrics also account for the overhead of recentering notified subscriptions, inserting/deleting subscriptions, and load balancing (for CN).

Workload We generate synthetic subscription radii using a truncated normal distribution to model skewed interests. The radii lie in the range $[1, 200k]$. In case of same-rad, we choose the initial centers of subscriptions with the same radius from a truncated normal

Table 1: Summary of parameters.

parameter	value
number of events	100000
diff-rad	
number of subscriptions	10k–100k
subscription radii	$N(25000, 50000)$
distribution of events	$AR_1(400, 1, 1000)$
same-rad	
number of subscriptions	5k–50k
subscription radius	51000
distribution of events	$AR_1(500, 1, 800)$
all-rad	
subscriptions	See Sec. 7.2.3
distribution of events	$AR_1(500, 1, 800)$

# subs.	20k	40k	60k	80k	100k
CN(r)	30	33	34	35	36
Unicast	499	982	1426	1850	2201

Table 2: Average number of outgoing messages from server (diff-rad).

# subs.	10k	20k	30k	40k	50k
CN(lid)	0.04	0.04	0.04	0.04	0.04
Unicast	97	193	290	386	483

Table 3: Average number of outgoing messages from server (same-rad).

Figure 6: Processing time; increasing number of subscriptions (diff-rad).

Figure 7: Network traffic; increasing number of subscriptions (diff-rad).

Figure 8: Processing time; increasing number of subscriptions (same-rad).

Figure 9: Network traffic; increasing number of subscriptions (same-rad).

distribution.

We experiment with both synthetic and real event data. Synthetic event data is derived using an order-1 autoregressive model, $AR_1(c, \phi, \sigma)$. The new value set by the i-th update is derived as: $U_i = c + \phi U_{i-1} + N(0, \sigma)$, where $N(0, \sigma)$ represents a normally distributed error with mean 0 and standard deviation σ. In this paper, we present results for $\phi = 1$, which represents a drifting random walk. In addition, we experiment with historical stock prices collected from Yahoo! Finance [24]. Details of the real workload are described later.

Table 1 summarizes the main experimental parameters.

Experimental Setup We perform link-level simulation of an INET topology with 20000 nodes. Of these, 1000 nodes are chosen as brokers participating in the publish/subscribe system. All subscriptions are hosted by brokers. We do not model the last-hop cost for brokers to notify subscribers, because such costs are uniform across all approaches being compared, and heavily depend on the end-user delivery mechanisms such as emails and instant messages. Also, we do not model message hops from publishers to the server. Accordingly, to ensure fair comparison, we disregard the hop from the publisher to the network entry point for serverless approaches such as CN(lr) and CN(c).

In experiments, we have found results on IP-level costs (e.g., the number of IP message hops) to follow similar trends as those on node-level costs (e.g., the number of overlay message hops). Therefore, we only show results on node-level metrics.

7.2 Experiments and Results

7.2.1 Subscriptions with Different Radii (diff-rad)

We experiment with 100k synthetic events and from 10k to 100k subscriptions generated using the parameters shown in Table 1. Subscriptions with the same radius are constrained to have the same center in this subsection.

Processing Time We increase the number of subscriptions and measure the server-side processing time for the various approaches
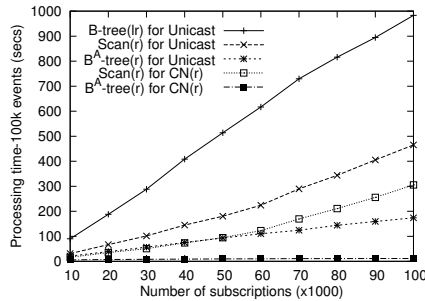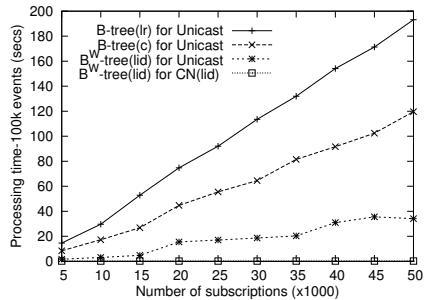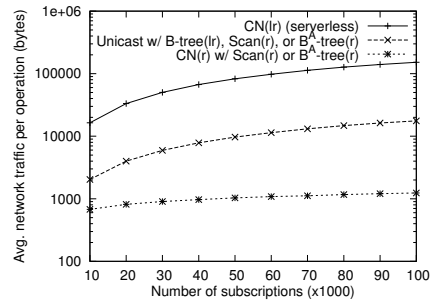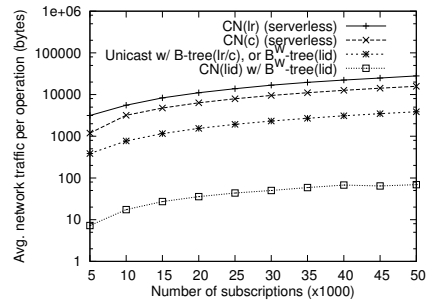
that we implemented for diff-rad. From Figure 6, we see that B-tree(lr) really suffers because it indexes dynamic endpoints and incurs substantial overhead of recentering subscriptions. Although CN(r) is the ideal match for B^A-tree(r), B^A-tree(r) still outperforms B-tree(lr) and Scan(r) significantly even for unicast.

Other factors being equal, server-side processing techniques intended for unicast are generally slower than techniques intended for a content-driven network, because the latter techniques only need to generate a concise description of affected subscriptions instead of a long list. Table 2 shows the average number of outgoing messages from the server, for unicast and CN(r). We see that the semantic descriptions generated for CN(r) (radius ranges) are much more concise than lists of affected subscriptions, even after 100k events.

Back to Figure 6, we see that to produce radius ranges for CN(r), B^A-tree(r) performs much better than the naive Scan(r). B^A-tree(r) takes negligible time to process even 100k subscriptions.

Network Traffic We next compare the average network traffic generated per event in bytes for different notification dissemination methods, as we increase the total number of subscriptions. Figure 7 shows the results. Note that the y-axis uses a logarithmic scale. There is an order of magnitude reduction in network traffic when using CN(r), as compared with unicast. CN(lr) does even worse than unicast because of the overhead of recentering subscriptions as well as the added cost of load balancing necessitated by movements of subscriptions. Results on the number of overlay message hops reveal a similar trend, and are omitted.

Maximum Node Stress The maximum node stresses over all events (for 10000 subscriptions) are 3969, 1984, and 282 for CN(lr), unicast, and CN(r), respectively. CN(lr) is the worst because of the high reorganization cost caused by subscription recentering. Unicast bottlenecks the server with a large number of outgoing messages. CN(r) is the lowest because of the smaller number of radius ranges and the static content-driven network organization. For CN(r), 93% of events have maximum node stress less than 100.
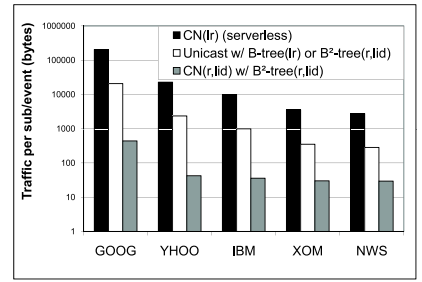
To summarize, the combination of B^A-tree(r)+CN(r) offers the best solution to diff-rad. The serverless approach CN(lr) is too
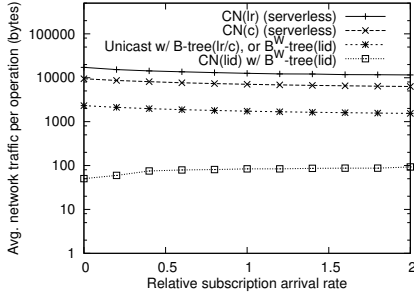
Figure 10: Increasing relative subscription arrival rate (same-rad).
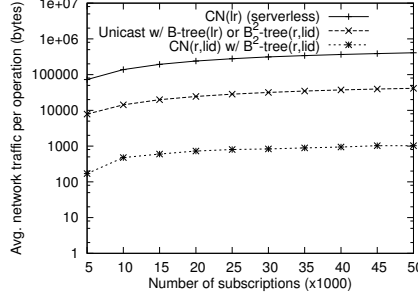
Figure 11: Average network traffic per operation (all-rad).

Figure 12: Average network traffic per operation, real workload (all-rad).

expensive in terms of network costs, while the other approaches are inferior to B^A-tree(r)+CN(r) by all performance metrics we have experimented with.

7.2.2 Subscriptions with Same Radius (*same-rad*)

We next experiment with a large number of subscriptions having the same radius; experimental parameters are shown in Table 1. We interleave arrivals of new subscriptions with events, so subscriptions can have different centers. During the bootstrap phase, we insert subscriptions that have initial centers normally distributed around the first event.

Processing Time We vary the total number of subscriptions from 5k to 50k, and measure processing time for 100000 events. The results are shown in Figure 8. We see that B^W-tree(lid) is very efficient at generating label ranges, taking negligible time regardless of the number of subscriptions. B^W-tree(lid) also does very well at generating subscription lists for unicast, much better than B-tree(lr) and B-tree(c), which suffer because they index dynamic properties of subscriptions. Table 3 shows the average number of outgoing messages generated by the server, for unicast and CN(lid). We see that the semantic descriptions generated for CN(lid) (label ranges) are extremely concise compared to lists of affected subscriptions. This factor also contributes to the lower server processing cost of B^W-tree(lid)+CN(lid).

Network Traffic In Figure 9, we show the average network traffic per event while increasing the total number of subscriptions for *same-rad*. Again, the y-axis uses logarithmic scale. Dissemination using CN(lr) is quite inefficient due to subscription movement and load balancing. CN(c) is slightly better as it indexes only the centers. Unicast, despite its simplicity, turns out to be better than both these alternatives. However, CN(lid) does orders of magnitude better, taking less than 100 bytes of traffic per event on average. The reason is that CN(lid) only needs to disseminate at most only two label ranges per event. Again, results on the number of overlay message hops reveal a similar trend, and are omitted.

Maximum Node Stress The maximum node stress over all events (for 10000 subscriptions) are 18051, 9026, 9025, and 2 for CN(lr), CN(r), unicast, and CN(lid), respectively. CN(lr) and CN(c) perform poorly due to reinsertion of all affected subscriptions. Unicast requires the server to send out a large number of messages. CN(lid) has the lowest node stress because it sends very concise semantic descriptions of affected subscriptions.

Performance of Insertion When subscriptions come and go over time, B^W-tree(lid)+CN(lid) may need to perform subscription relabeling. Here, we test how relabeling costs degrade the performance of this combination. We allow subscriptions to be continuously inserted during the experiment, and vary the *relative subscription arrival rate (RSAR)*. RSAR of 2 means that two new subscriptions are inserted for every event arrival. We see from Fig-

ure 10 that the average network traffic per operation (subscription insertion or event dissemination) increases for CN(lid) with increasing RSAR. For the other approaches, subscription insertion is less expensive than event dissemination, which explains why the average cost of an operation decreases slightly with increasing RSAR for these approaches. Regardless, CN(lid) is orders of magnitude better than them, even for an unusually high RSAR of 2. In practice, we would expect the event arrival rate to be higher than the subscription insertion rate.

To summarize, the combination of B^W-tree(lid)+CN(lid) offers the best solution to *same-rad*. Although serverless approaches incur no server-side processing cost, their high network costs make them infeasible. Other approaches are outperformed by B^W-tree(lid)+CN(lid) in terms of both server- and network-side performance metrics.

7.2.3 Subscriptions with All Radii (*all-rad*)

Finally, we experiment with a large number of subscriptions with arbitrary radii and full personalization of event views. In this set of experiments, we generate subscriptions on stock prices as follows. To generate s subscriptions, we first have a baseline uniform random distribution of $0.1s$ subscriptions over radii (defined at cent boundaries) ranging from $0 to $100. To model the observation that subscriptions with radii at whole dollar amounts may be more popular (because they are simpler and more natural for users to specify), we overlay the remaining subscriptions as a truncated normal distribution $N(20, 20)$ constrained at dollar boundaries over the baseline set of subscriptions. We show performance only in terms of network traffic; the server processing cost of using B^2-tree(r,lid) for content-driven dissemination was found to be at least an order of magnitude lower than using B-tree(lr) for unicast.

Network Traffic We use the event distribution shown in Table 1. We first vary the total number of subscriptions from 5k to 50k, and plot the average network traffic per operation (new subscription or event) in Figure 11. The y-axis uses a logarithmic scale. We see that CN(r,lid) performs an order of magnitude better than unicast. As expected, the serverless CN(lr) does not do well because of subscription movements, performing even worse than unicast.

Results of Real Workload We use a real event workload in this experiment. Traces of historical daily stock prices were obtained from Yahoo! Finance [24], for 5 leading stocks from various industries. Each trace has between 600 and 11000 events. We replay the stock price variations in these traces multiple times, to get a dataset with 50k events per stock. We randomly interleave new subscriptions with incoming events.

Figure 12 shows the performance in terms of average network traffic per operation for this real workload. The y-axis uses a logarithmic scale. Again, the serverless CN(lr) does worse than the other approaches. Unicast does better, but our techniques show an

order of magnitude reduction in traffic while adhering to the strict semantics of personalized notifications.

8   Related Work

Dynamic Data Dissemination   As discussed in Section 2.3, Shah et al. [21, 20] address the problem of disseminating dynamic data over a network of repositories. However, they target weaker subscription semantics and build a customized dissemination structure. We support stricter notification semantics efficiently, and leverage standard dissemination substrates.

Bounded Approximate Caching   Bounded approximate cache maintenance [15] has been studied extensively in database literature. Olston et al. [17] use bounded approximate caching to maintain caches on a best-effort basis, which does not adhere to precise subscription semantics. In [16], the focus is on how to set these bounds adaptively based on query workloads. However, in our setting, notification conditions are specified by the subscriber and cannot be changed by the system. Also, related work in this area does not consider scalable indexing or update dissemination for bounded approximate caches.

Continuous Query Systems   Continuous query systems [13, 7, 23] 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. *OpenCQ* [13] supports notification conditions that refer to current and previous database states, and *NiagaraCQ* [7] supports timer-based notification conditions. In other words, *NiagaraCQ* allows control over the staleness of subscriptions, but not over their accuracy, in terms of user-defined metrics. All these systems use simpler processing techniques for notification conditions and do not optimize notification dissemination. On the other hand, we propose efficient processing techniques and address dissemination issues as well.

Other Related Work   A number of publish/subscribe systems built by the database community have made the subscription language more powerful [14, 8]. Many of them have added language support for notification conditions. For instance, *Xyleme* [14] supports *monitoring queries*, which are analogous to notification conditions. *SMILE* [12] supports SQL queries over the event history. However, none of them address the efficiency issues in processing notification conditions. Our techniques can be employed by such systems to support notification conditions in a scalable manner.

9   Conclusion

We address the problem of adding scalable support for subscriptions with personalized value-based notification conditions in a large-scale wide-area publish/subscribe system. Our first step was to efficiently process and disseminate events for subscriptions with varying radii, where subscriptions with the same radius share the same view of the data. We next showed how to efficiently support subscriptions with the same radius but different views of the data. Finally, we showed how to put the two pieces together to build a publish/subscribe infrastructure capable of handling the precise notification semantics in a scalable manner. We also discussed a number of extensions such as support for may-notify and must-notify conditions, and relative notification conditions.

We maintain a clean interface between the server and the network, and leverage established dissemination components. This design allows us to quickly build and deploy a robust wide-area publish/subscribe system. Our techniques were shown to be much more efficient than less sophisticated solutions, with an order of magnitude reduction in network traffic and server processing cost.

References

[1] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *IEEE Symp. on Foundations of Computer Science*, 1996.

[2] 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.

[3] 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.

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

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

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

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

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

[9] P. F. Dietz. Maintaining order in a linked list. In *ACM Symp. on Theory of Computing*, 1982.

[10] P. F. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *ACM Symp. on Theory of Computing*, 1987.

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

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

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

[14] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. *SIGMOD Record*, 2001.

[15] C. Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.

[16] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, 2001.

[17] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, 2002.

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

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

[20] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *VLDB*, 2003.

[21] S. Shah, K. Ramamritham, and P. J. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *VLDB*, 2002.

[22] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In *ICDE*, 2005.

[23] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, 1992.

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