Processing and Notifying Range Top-k Subscriptions

Albert Yu, Pankaj K. Agarwal, Jun Yang
Duke University
{syu,pankaj,junyang}@cs.duke.edu

***Abstract*— We consider how to support a large number of users over a wide-area network whose interests are characterized by range top-k continuous queries. Given an object update, we need to notify users whose top-k results are affected. Simple solutions include using a content-driven network to notify all users whose interest ranges contain the update (ignoring top-k), or using a server to compute only the affected queries and notifying them individually. The former solution generates too much network traffic, while the latter overwhelms the server. We present a geometric framework for the problem that allows us to describe the set of affected queries succinctly with messages that can be efficiently disseminated using content-driven networks. We give fast algorithms to reformulate each update into a set of messages whose number is provably optimal, with or without knowing all user interests. We also present extensions to our solution, including an approximate algorithm that trades off between the cost of server-side reformulation and that of user-side post-processing, as well as efficient techniques for batch updates.**

I. Introduction

Consider a range top-k query over a database of objects (e.g. stocks). The query examines a subset of the objects satisfying a range condition (e.g., stocks with risk rating between medium high and high), and picks the top k objects within this subset by some ranking criterion (e.g., stocks with the k lowest price-to-earning ratios). Over time, when the set of objects or their attribute values change, we wish to keep the query result up to date, as in the standard view maintenance and continuous query processing settings. We are interested in how to support hundreds of thousands or even millions of such queries simultaneously. Representing different user interests, these queries may have different range conditions and therefore different lists of k objects as their answers.

A challenging application setting is when a large number of these queries, which we shall refer to as *subscriptions*, are located across a wide-area network. For each event updating the database, we must notify all subscriptions whose results are affected. Notification messages should carry enough information so that the affected subscriptions can update their top-k lists accordingly. A naive approach would be to use a central server to maintain all objects and subscriptions, compute the list of affected subscriptions for each event, and notify each affected subscription with the change to its top-k list. Since an event may affect many subscriptions, this approach can easily overload the server with processing and messaging costs at least linear in the number of affected subscriptions.

A solution is to push some event processing and dissemination work into a more "intelligent" network, but at the cost of increasing system complexity. As demonstrated in previous work [8], [9], a *content-driven network* (*CN*) offers a good trade-off between functionality and complexity. CN is a class of overlay networks designed for efficient dissemination, with a clean message interface. Many off-the-shelf overlay networks are examples of CN, e.g., *content-based networks* [7] and *content-addressable networks* [24].[1] For the purpose of this paper, we regard CN as a black box for efficiently delivering a message to all subscriptions whose query parameters satisfy a selection condition carried by the message.[2] Instead of enumerating affected subscriptions one by one, the server would compute a compact description for the set of affected subscriptions, and then translate this description into a series of condition-carrying messages to be sent through CN. The number of such messages is usually far less than the number of affected subscriptions, thereby relieving the server bottleneck.

Range top-k subscriptions are challenging for several reasons. It is straightforward for CN to handle range subscriptions without top-k as in standard *publish/subscribe*: a message simply needs to list the updated object's attribute values, which can be interpreted as a condition testing whether a subscription range contains the object. However, such a message is not enough for range top-k subscriptions because they are "stateful": whether a subscription is affected depends on how the updated object ranks against others within the subscription range. Furthermore, if the updated object drops out of a subscription's top-k list, the new k-th ranked object must be sent to the subscription. While previous work [8] addresses the special case of $k = 1$ (i.e., range min/max subscriptions), the general case we handle in this paper is considerably more complex and has more practical applications.

A geometric framework. In this paper, we develop a geometric framework to support range top-k subscriptions. The geometric framework enables us to view the problem of generating notification messages intuitively as one of tiling a potentially complex region of affected subscriptions (in an appropriately defined subscription space) using simple geometric shapes. The set of tiles form a compact description of the region. Each tile corresponds to a CN message, whose condition selects all subscriptions covered by the tile. While one could first compute the list of affected subscriptions and then find the tiling, we develop algorithms (described below)

[1] CN is named after these popular examples, which should *not* be confused with *content delivery/distribution networks* [4] that serve the different purpose of replicating popular Web objects.

[2] An equivalent, dual view is that CN allows subscriptions to be selection conditions over message attributes, and CN efficiently delivers a message to all subscriptions whose conditions are satisfied by the message.

that avoid computing this potentially long list in the first place.

New algorithms. We propose new algorithms for message generation based on the framework above. These algorithms are scalable—they run in time dependent on the number of messages they generate, not the number of affected subscriptions (which could be substantially larger). Experiments confirm that this property translates into substantial savings in both server running time and network dissemination cost; furthermore, the performance lead over other approaches widens as the number of subscriptions increases.

We start with two algorithms. The first one, which we call *Paint-Dense*, is subscription-oblivious; it examines only the set of objects. This feature is attractive from both scalability and privacy perspectives, because it alleviates the need for a server to track a large number of subscriptions. *Paint-Dense* computes the optimal tiling assuming no knowledge of the subscriptions. The second version, *Paint-Sparse*, uses both the set of objects and the set of subscriptions. Intuitively, it produces a tiling sensitive to the subscription distribution; the size of the tiling is 2-approximate and often much smaller than that generated by *Paint-Dense*.

We also consider the case of batch updates, where a subscription needs to be notified of the net change in its result at the end of a batch. Simply processing this batch one event at a time generates more traffic than necessary. We show that by pre-processing the batch (coalescing and reordering updates), we can guarantee that subscribers receive the minimum number of messages needed.

Besides *Paint-Dense* and *Paint-Sparse*, we provide approximate algorithms that generate even fewer messages from the server at the expense of more "false positives"—notifications received by a subscription but not needed. False positives are discarded by each subscription with simple local post-processing, so our "approximate" algorithms still guarantee exact subscription results. Having fewer messages reduces processing and messaging loads on the server, but false positives bring higher last-hop traffic and extra post-processing. We allow the trade-off to be adjusted using a parameter $\varepsilon \leq 1$, while guaranteeing that subscriptions miss no notifications and receive no objects ranked below $(1+\varepsilon)k$.

Higher dimensions and beyond. For simplicity, we present our framework and algorithms in this paper assuming 1-d range top-k subscriptions, but they can be generalized to subscriptions whose range conditions involve multiple dimensions and more general constraints. Because of limited space, we briefly sketch out how extension to higher dimensions works. As a concrete illustration, in the technical report version [27] of this paper, we present the detailed algorithm and experimental evaluation for 1.5-d range top-k subscriptions.[3]

The subscription type we consider in this paper—orthogonal range top-k—is a standard one in most subscription/query languages. While there exist a plethora of proposals for other language features, little is known about how best to support this standard subscription type; our contributions fill this void. We also note that our techniques apply to top-k subscriptions with other types of conditions too. For example, conditions comparing categorical attributes against concepts drawn from a hierarchy can be mapped to range conditions with appropriate encoding of the hierarchy. For another example, range conditions subsume near-neighbor conditions under the L_∞ norm, and in low dimensions they can be effective as building blocks for supporting near-neighbor and nearest-neighbor conditions under other distance metrics.

This paper focuses on application settings with many geographically dispersed subscriptions to a central database (e.g., news aggregators and financial information services). However, our solution can be extended to other settings, ranging from simpler ones such as non-distributed continuous query systems with no need to deliver results over a network, to more complex ones such as publish/subscribe systems with multiple, distributed event publishers. We shall revisit this point when concluding the paper, and more details can be found in [27].

II. Overview and Our Framework

A. Problem Formulation

Consider a set $\mathcal{O}$ of n objects. For simplicity, assume each object has only two numeric attributes: x is used in range conditions, while y is used for ranking objects in ascending order of their y-values. Section IV-C discusses how to generalize the problem and our solutions to higher dimensions. For each object i ($1 \leq i \leq n$), let $x_i \in \mathbb{R}$ denote its x-value and $y_i \in \mathbb{R}$ denote its y-value. Without loss of generality, we assume all x_i's and y_i's are distinct.

We have a set $\mathcal{S}$ of m subscriptions over the network. Each subscription S_j ($1 \leq j \leq m$) specifies an x-value range of interest, denoted $\sigma_j = [\ell_j, r_j] \subseteq \mathbb{R}$. For some $k \ll n$, S_j wishes to track the top k objects (along their attribute values) in σ_j, i.e., those with the k smallest y-values. More precisely, S_j must maintain, at all times, the list $\mathrm{top}_k(S_j) = \{(x_i, y_i) \mid x_i \in \sigma_j \wedge |\{i' \mid x_{i'} \in \sigma_j \wedge y_{i'} < y_i\}| < k\}$.

A *(y-update) event*, denoted $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \rightarrow y_i^{\mathrm{new}})$, changes object i's y-value from y_i^{old} to y_i^{new}. Upon receiving an event δ, we must notify all *affected* subscriptions. A subscription S_j is affected by δ iff δ changes $\mathrm{top}_k(S_j)$; i.e., either the membership of this list changes or the y-value of some object in this list is updated as a result of δ. See Figure 1(a) for an example. For simplicity of presentation, we focus our discussion on y-update events.[4]

To notify all affected subscriptions, we follow the same overall approach as [8]—first using a server to *reformulate* the event into a sequence of messages, then using CN to *disseminate* these messages to subscriptions, and finally having subscriptions *post-process* received messages

[3] An example of a 1.5-d range top-k subscription would be "k stocks that have the lowest price-to-earning ratio among those with market capitalization above 50 billion US dollars and risk rating between medium high and high."

[4] We can simply treat object insertion and deletion as y-update events $\mathsf{Upd}(x_i, \infty \rightarrow y_i)$ and $\mathsf{Upd}(x_i, y_i \rightarrow \infty)$, respectively. We can simulate an update to object i's x-value from x_i^{old} to x_i^{new} by a deletion of $(x_i^{\mathrm{old}}, y_i)$ followed by an insertion of $(x_i^{\mathrm{new}}, y_i)$. Alternatively, it is straightforward to extend our algorithms to handle these events directly.

to maintain their top-k lists. More specifically, the server maintains the set of objects $\mathcal{O}$, and reformulates each event into a sequence of constant-size CN messages of the format $\mathsf{Msg}(\ell_I, r_I, \ell_O, r_O, x_i, y_i)$, where $[\ell_I, r_I] \subseteq (\ell_O, r_O)$ are two nested ranges in $\mathbb{R}$, and (x_i, y_i) represents some object i (with its attribute values). Each message is interpreted as a condition over subscriptions' ranges of interest: CN delivers this message to subscription S_j iff $[\ell_I, r_I] \subseteq \sigma_j \subseteq (\ell_O, r_O)$ (see Figure 1(b)). Each subscription S_j maintains its own top-k list L_j. Upon receiving a message, S_j checks whether L_j currently contains object i (the one with x-value equal to x_i). If yes, S_j simply updates the y-value of this object to y_i. Otherwise, S_j updates its list L_j to contain the top k objects in $L_j \cup \{(x_i, y_i)\}$.

Our goal is to develop efficient algorithms for generating the sequence of CN messages for each event, such that every affected subscription will have its top-k list correctly updated by following the protocol above. We consider several performance measures in designing our algorithms: 1) the number of messages generated; 2) time spent by the server in generating them; and 3) the number of messages received by the subscriptions.[5] These measures present interesting trade-offs and must be considered jointly. For instance, minimizing (3) alone would not be sufficient; the naive approach of enumerating all affected subscriptions and unicasting to them one by one achieves this objective, but does poorly on the other criteria. A better goal is to keep (3) minimized and optimize other criteria as much as possible; our *exact* algorithms have this goal. If the server becomes a bottleneck, we could further reduce (1) and (2) at the expense of (3). In this case, we would allow an unaffected subscription to be notified; our *approximate* algorithms take this approach. These results are discussed further below.

B. Overview of Our Algorithms

Exact algorithms. With an *exact* algorithm, the server generates messages for each event such that only affected subscriptions are notified, and they each receive only one message (per y-update event). We consider two settings.

Subscription-oblivious. For the case where the server has no knowledge of the set of subscriptions (because of either scalability or privacy concerns), we develop an algorithm *Paint-Dense* with the following properties (Theorem 1):

- The algorithm is given $\mathcal{O}$, but not $\mathcal{S}$.
- It generates the minimum number of messages possible for any exact algorithm if $\mathcal{S}$ is *dense*: that is, given the set of objects $\mathcal{O}$, for any x-value range σ, there exists some subscription interested in precisely the objects within σ.
- Its running time depends on the number of messages generated, but not on $|\mathcal{S}|$ or the number of affected subscriptions, which can be much larger.

Subscription-aware. We call a set of subscriptions *sparse* if it is not dense. In this case, *Paint-Dense* may generate a message that does not reach any subscription, wasting both server processing and network dissemination efforts. Therefore, we develop *Paint-Sparse*, with the following properties (Theorem 2):

- The algorithm is given both $\mathcal{O}$ and $\mathcal{S}$.
- It generates at most twice the minimum number of messages possible for any exact algorithm, and it never generates any message that reaches no subscription.
- Its running time is sublinear in $|\mathcal{O}|$ and $|\mathcal{S}|$, and depends on the number of messages generated instead of the number of affected subscriptions, which can be much larger.

Extensions. We consider the batched version of the problem, in which subscriptions only need to have their top-k lists correctly updated at the end of an event sequence. We develop *Paint-Batch*, which pre-processes the event sequence before applying either algorithm above (with minor modifications) to each event. *Paint-Batch* operates well with the basic CN interface of Section II-A, and is able to guarantee that each subscriber receives the minimum number of messages possible (Theorem 3), which is far less than if we process all events in the sequence in order.

We also relax the requirement that only affected subscription may receive messages. By allowing unaffected subscriptions to receive unnecessary messages, an *approximate* algorithm further reduces the number of messages generated by the server. We develop approximate algorithms *Paint-Dense*(ε) and *Paint-Sparse*(ε), with parameter $\varepsilon \leq 1$ controlling this trade-off. Compared with their exact counterparts, they reduce the number of messages by a factor of εk while guaranteeing that unnecessarily received objects are ranked within $(1 \pm \varepsilon)k$ (Theorem 4). Furthermore, such objects are automatically ignored by subscriptions following the same protocol in Section II-A, so all results remain accurate at all times.

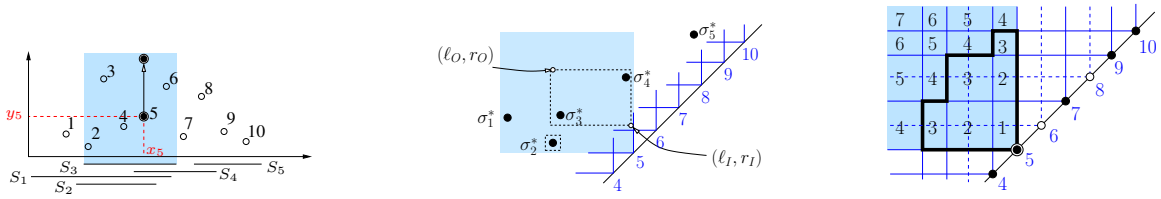
Both extensions above inherit the efficiency of *Paint-Dense* and *Paint-Sparse*, with running times dependent on the number of messages generated rather than subscriptions affected.

We also briefly discuss how to extend our problem and framework to higher dimensions (Section IV-C), where a subscription's range of interest becomes a d-dimensional region.

Data structures. For all our algorithms, the server maintains a data structure indexing the set of objects $\mathcal{O}$ by (x, y) as points in $\mathbb{R}^2$. This index supports the following operations:

- Events that update objects in $\mathcal{O}$.
- $\mathrm{first}_k(x_0, y_0, s)$: Here $(x_0, y_0) \in \mathbb{R}^2$ and $s \in \{\leftarrow, \rightarrow\}$. If s is $\leftarrow$ (resp. $\rightarrow$), then this query finds the first k objects in $\mathcal{O}$ in the southwest (resp. southeast) quadrant with (x_0, y_0) as the apex as we proceed in the $(-x)$-direction (resp. $(+x)$-direction) from (x_0, y_0). If the quadrant contains fewer than k objects, all of them are reported. Only the x-values of the objects are reported by first_k, and they are reported in the order encountered.
- $\min_y(\sigma, y_0)$: Given an x-value range σ and a y-value y_0, this query returns the object in $\mathcal{O}$ with the minimum y-value in the 3-sided rectangle $\sigma \times (y_0, \infty)$.

[5] For evaluation (Section V), especially comparison with approaches that do not use CN, we also consider the total traffic in the underlying IP network.



(a) (b) (c)

Fig. 1. **(a)** Event space $\mathbb{E}$. **(b)** Subscription space $\mathbb{S}$. **(c)** Partitioning of θ_5 into θ_5^1 , θ_5^2, $\cdots$. For $k = 3$, $\mathrm{IR}(5)$ is shown with thick outline.

We use $t(n)$ (where $n = |\mathcal{O}|$) in denoting the upper bounds on running times of the operations above: object updates and $\min_y$ all run in $O(t(n))$ time, while first_k runs in $O(t(n)+k)$ time. If we use kd-tree for the index, then the index size is linear and $t(n) = \sqrt{n}$. If we allow $O(n \log n)$ space for the index, then we can use a data structure based on *dynamic range trees* [20] to get $t(n) = \log^2 n$.

C. Geometric Framework

We now introduce a geometric framework essential to the understanding of the problem. Section III will reveal, with the help of this framework, the structure inherent in the seemingly arbitrary subset of affected subscriptions, which allows us to conveniently view the task of generating CN message as one of tiling a complex region using only rectangles.

Let $\mathbb{E} = \mathbb{R}^2$ denote the *event space*, where each object i is represented as a point $(x_i, y_i) \in \mathbb{R}^2$ (Figure 1(a)). Each subscription S_j is interested in objects that lie in the vertical strip $\sigma_j \times \mathbb{R}$; $\mathrm{top}_k(S_j)$ returns the k lowest among them.

Let $\mathbb{S} = \mathbb{R}^2$ denote the *subscription space*, where each subscription S_j with range of interest $\sigma_j = [\ell_j, r_j]$ is mapped to the point $\sigma_j^\star = (\ell_j, r_j) \in \mathbb{R}^2$ (Figure 1(b)). Object i is mapped to the northwest quadrant θ_i with apex at (x_i, x_i); i.e., $\theta_i = \{(\ell, r) \mid \ell \le x_i \le r\}$. S_j is interested in object i only if $\sigma_j^\star \in \theta_i$. A CN message $\mathsf{Msg}(\ell_I, r_I, \ell_O, r_O, x_i, y_i)$ corresponds to notifying, with (x_i, y_i), all subscriptions in the rectangle with southeast and northwest corners at (ℓ_I, r_I) and (ℓ_O, r_O), respectively.

We now introduce the concept of *influence regions*, which allows us to describe the answers to range top-k range queries geometrically. For an object i and a point $\xi \in \theta_i$, we define the *level* of ξ w.r.t. i, denoted by $\lambda(\xi, i)$, as the number of objects j s.t. $\xi \in \theta_j$ and $y_j \le y_i$. For $v > 0$, let $\theta_i^v = \{\xi \in \theta_i \mid \lambda(\xi, i) = v\}$. The set $\{\theta_i^v \mid i > 0\}$ partitions θ_i (Figure 1(c)). Let $\theta_i^{\le v} = \bigcup_{u \le v} \theta_i^u$. We call $\theta_i^{\le k}$ the *influence region* of object i, denoted $\mathrm{IR}(i)$. The following lemma establishes the connection between an object's influence region and its membership in the subscriptions' top-k lists. (Because of space constraints, please refer to [27] for all proofs.)

Lemma 1. $(x_i, y_i) \in \mathrm{top}_k(S_j)$ *iff* $\sigma_j^\star \in \mathrm{IR}(i)$.

It turns out that $\mathrm{IR}(i)$ and $\theta_i^{\le v}$ in general have a regular shape that is easy to compute, as shown by the following lemma. Recall that $\mathrm{first}_v(x_i, y_i, \leftarrow)$ (resp. $\mathrm{first}_v(x_i, y_i, \rightarrow)$) denotes the set of x-values of the first v objects in $\mathcal{O}$ to the west (resp. east) of x_i with y-values less than y_i.

Lemma 2. *Consider object i with values (x_i, y_i). Let*

- $\ell_1 > \ell_2 > \cdots > \ell_v$ *denote the list returned by* $\mathrm{first}_v(x_i, y_i, \leftarrow)$ *(padded with $-\infty$ if fewer than v values are returned), and*
- $r_1 < r_2 < \cdots < r_v$ *denote the list returned by* $\mathrm{first}_v(x_i, y_i, \rightarrow)$ *(padded with ∞ if fewer than v values are returned).*

$\theta_i^{\le v}$ is an axis-aligned subregion of the quadrant θ_i, with vertices $(x_i, x_i), (\ell_v, x_i), (\ell_v, r_1), (\ell_{v-1}, r_1), (\ell_{v-1}, r_2), \ldots, (\ell_1, r_{v-1}), (\ell_1, r_v), (x_i, r_v)$ in clockwise order, ignoring degenerate vertices with $-\infty$ or ∞ coordinates.

Lemma 2 implies that $\mathrm{IR}(i) \subseteq \mathbb{S}$ is a staircase polygon with (x_i, x_i) as its apex and an ℓr-monotone rectilinear chain with no more than k "steps" (Figure 1(c)). We define the *influence interval* of object i, denoted $\mathrm{II}(i)$, to be the x-value range (ℓ_k, r_k), where ℓ_k and r_k are as defined above. By Lemma 2, $\mathrm{IR}(i)$ is bounded from the west by $\ell = \ell_k$ and from the north by $r = r_k$. Furthermore, the quadrants in $\{\theta_h \mid x_h \in \mathrm{first}_k(x_i, y_i, \leftarrow) \cup \mathrm{first}_k(x_i, y_i, \rightarrow)\}$ induce a partitioning of $\mathrm{IR}(i)$ into $O(k^2)$ (possibly open-sided) rectangles, denoted $\mathrm{IR}_\square(i)$. For each rectangle $\rho \in \mathrm{IR}_\square(i)$, all points in ρ have the same level w.r.t. i; we call this number the *level* of rectangle ρ, denoted $\lambda(\rho)$. Each θ_i^v $(v > 0)$ consists of up to v rectangles of level v, arranged along a diagonal and to the immediate southeast of θ_i^{v+1}'s rectangles (Figure 1(c)).

It also follows from Lemma 2 that given $\mathrm{first}_k(x_i, y_i, \leftarrow)$ and $\mathrm{first}_k(x_i, y_i, \rightarrow)$, $\mathrm{IR}(i)$ can be computed in time linear in the number of vertices of $\mathrm{IR}(i)$.

Example. Refer to Fig. 1. All subscriptions other than S_5 are interested in object 5. In $\mathbb{S}$, the northwest quadrant θ_5 contains σ_1^*, σ_2^*, σ_3^*, σ_4^*, but not σ_5^*. Suppose $k = 3$. The influence region of object 5 is an axis-aligned subregion of the quadrant with vertices $(x_5, x_5), (\ell_3, x_5), (\ell_3, r_1), (\ell_2, r_1), (\ell_2, r_2), (\ell_1, r_2), (\ell_1, r_3), (x_5, r_3)$ in clockwise order, where ℓ_1, ℓ_2, and ℓ_3 (resp. r_1, r_2, r_3) are the x-values of objects 4, 2, and 1 (resp. objects 7, 9, and 10), respectively. Since S_2 and S_3 rank object 5 at third, σ_2^* and σ_3^* lie in $\theta_5^3 \subset \mathrm{IR}(5)$. Similarly, S_4 ranks object 5 at second and σ_4^* lies in $\theta_5^2 \subset \mathrm{IR}(5)$. However, $\sigma_1^* \in \theta_5^4 \not\subset \mathrm{IR}(5)$ because S_1 ranks object 5 at fourth and therefore, object 5 is not in $\mathrm{top}_k(S_1)$. When object 5's y-value increases as shown, object 5 is replaced by object 3 in $\mathrm{top}_k(S_2)$, and by object 6 in $\mathrm{top}_k(S_3)$ and $\mathrm{top}_k(S_4)$. CN messages (dashed rectangles) are generated to notify S_2, S_3 and S_4.

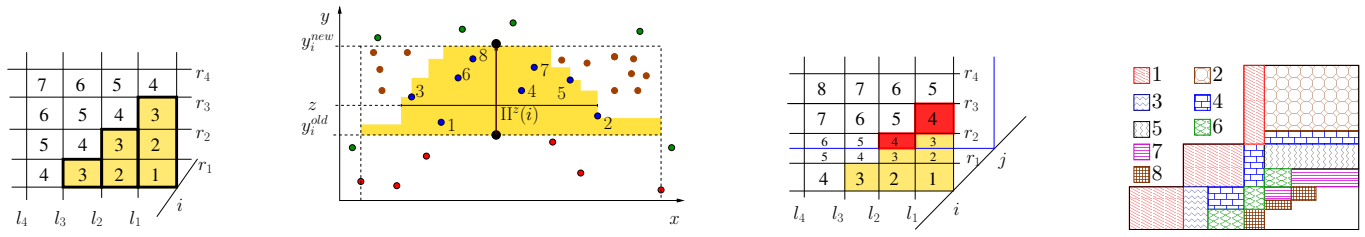


Fig. 2. **(a)** Tiling $\mathrm{IR}^{\mathrm{new}}(i)$ (shaded, $k=3$) by CN messages (shown with thick outlines). **(b)** Sweep in $\mathbb{E}$; $k=3$. **(c)** Effect on $\mathrm{IR}^{z}(i)$ of encountering exposed object j during the sweep. **(d)** Tiling $\nabla\mathrm{IR}(i)$ by CN messages.

III. Exact Algorithms

A. Subscription-Oblivious

Consider an event $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \to y_i^{\mathrm{new}})$. Let $\mathrm{IR}^{\mathrm{old}}(i)$ (resp. $\mathrm{IR}^{\mathrm{new}}(i)$) denote the influence region of object i before (resp. after) the update. There are two cases: $y_i^{\mathrm{old}} > y_i^{\mathrm{new}}$, which possibly raises object i's rank, and $y_i^{\mathrm{old}} < y_i^{\mathrm{new}}$, which possibly lowers object i's rank.

Rank-raising update. This case is simple. It can be easily seen that if $y_i^{\mathrm{old}} > y_i^{\mathrm{new}}$, then $\mathrm{IR}^{\mathrm{new}}(i) \supseteq \mathrm{IR}^{\mathrm{old}}(i)$. Every subscription S_j in $\mathrm{IR}^{\mathrm{old}}(i)$ (i.e., $\sigma_j^\star \in \mathrm{IR}^{\mathrm{old}}(i)$) must receive $(x_i, y_i^{\mathrm{new}})$ to update the y-value of object i in $\mathrm{top}_k(S_j)$. Every subscription S_j in $\mathrm{IR}^{\mathrm{new}}(i) \setminus \mathrm{IR}^{\mathrm{old}}(i)$ must receive $(x_i, y_i^{\mathrm{new}})$ as a new object in $\mathrm{top}_k(S_j)$, which would displace some other object from $\mathrm{top}_k(S_j)$. In sum, it suffices to notify all subscriptions in $\mathrm{IR}^{\mathrm{new}}(i)$ with $(x_i, y_i^{\mathrm{new}})$. Since each CN message reaches a rectangle in $\mathbb{S}$, and $\mathrm{IR}^{\mathrm{new}}(i)$ has up to k "steps," in the worst case we need k messages to tile $\mathrm{IR}^{\mathrm{new}}(i)$, as illustrated in Figure 2(a). The detailed algorithm, *Paint-Dense-IR*, is presented in [27]. Its running time, dominated by the two first_k calls to compute the new $\mathrm{II}(i)$, is $O(t(n)+k)$.

Rank-lowering update. This case is more complex. If $y_i^{\mathrm{old}} < y_i^{\mathrm{new}}$, then $\mathrm{IR}^{\mathrm{new}}(i) \subseteq \mathrm{IR}^{\mathrm{old}}(i)$. First, we notify all subscriptions in $\mathrm{IR}^{\mathrm{old}}(i)$ with $(x_i, y_i^{\mathrm{new}})$ using no more than k messages, in the same way as we tile $\mathrm{IR}^{\mathrm{new}}(i)$ for a rank-raising update. These messages allow subscriptions to update the y-value of object i in their top-k lists. For those in $\mathrm{IR}^{\mathrm{new}}(i)$, no more messages are needed.

Next, each subscription in $\nabla\mathrm{IR}(i) = \mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ needs to receive an object j that will replace i in its top-k list.[6] Clearly, such objects have their influence regions expanded. We identify these objects by introducing the notion of *exposed* objects, and we generate messages for each such object to notify appropriate subscriptions in $\nabla\mathrm{IR}(i)$, as described below.

Imagine sweeping the y-value of i continuously from y_i^{old} to y_i^{new}. Let $\mathrm{IR}^{z}(i)$, $\mathrm{II}^{z}(i)$, and $\theta_i^{k|z}$ denote the influence region, influence interval, and θ_i^k, respectively, of i when its y-value is set to z. An object j with y-value $y_j \in (y_{\mathrm{old}}, y_{\mathrm{new}})$ is called *exposed* if the quadrant θ_j intersects $\mathrm{IR}^{y_j}(i)$. It can be shown [27] that $\Delta\mathrm{IR}(j) = \mathrm{IR}^{\mathrm{new}}(j) \setminus \mathrm{IR}^{\mathrm{old}}(j)$, the gain in j's influence region as the result of the update, is

$$\Delta\mathrm{IR}(j) = \mathrm{IR}^{y_j-\epsilon}(i) \setminus \mathrm{IR}^{y_j+\epsilon}(i) = \theta_i^{k|y_j-\epsilon} \cap \theta_j, \quad (1)$$

where ϵ is an arbitrarily small value.[7] Lemma 2 suggests that $\Delta\mathrm{IR}(j)$ consists of a series of up to k rectangles along a diagonal in the northeast direction (e.g. darkly shaded rectangles in Figure 2(c)). Intuitively, any subscription $S_a \in \Delta\mathrm{IR}(j)$ is interested in both i and j. Furthermore, when the y-value of i is $y_j-\epsilon$, S_a ranks i at k and j at $k+1$, so $i \in \mathrm{top}_k(S_a)$ and $j \notin \mathrm{top}_k(S_a)$. When the y-value of i increases from $y_j-\epsilon$ to $y_j+\epsilon$, i and j swap their ranks and now $j \in \mathrm{top}_k(S_a)$ and $i \notin \mathrm{top}_k(S_a)$. It can be checked that the top-k list does not change for any subscription in $\mathrm{IR}^{y_j-\epsilon}(i) \setminus \theta_i^{k|y_j-\epsilon}$—such a subscription is either not interested in j or the ranks of both i and j are less than k, in which case swapping their ranks does not affect the top-k list.

During the sweep, we maintain the list $\mathcal{L}^z$ (resp. $\mathcal{R}^z$), which is initialized by $\mathrm{first}_k(x_i, y_i^{\mathrm{old}}, \leftarrow)$ (resp. $\mathrm{first}_k(x_i, y_i^{\mathrm{old}}, \rightarrow)$) and always contains the x-values of the first k objects in $\mathcal{O}$ to the west (resp. east) of x_i with y-values at most z, padded with $-\infty$ (resp. ∞) if there are fewer than k such objects. By Lemma 2, $\mathcal{L}^z$ and $\mathcal{R}^z$ allow us to readily obtain $\mathrm{IR}^{z}(i)$, $\mathrm{II}^{z}(i)$, and the tiling $\mathrm{IR}^{z}_{\square}(i)$ of $\mathrm{IR}^{z}(i)$ as needed. The next exposed object above z is the object with the minimum y-value in the (open) 3-sided rectangle $\mathrm{II}^{z}(i) \times (z, \infty) \subseteq \mathbb{E}$, and can be found by $\min_y(\mathrm{II}^{z}(i), z)$; see Figure 2(b). Once a new exposed point is found, the lists $\mathcal{L}^z$, $\mathcal{R}^z$ are updated and at most k new CN-messages are generated to tile the region defined by Eq. (1). Together, messages generated during the sweep for all exposed objects tile $\nabla\mathrm{IR}(i)$ (Figure 2(d)).

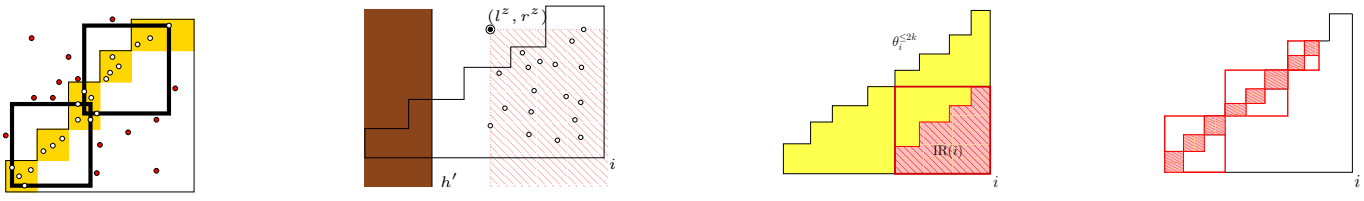
Example. Suppose $k=3$ and object i's y-value increases as shown in Figure 2(b). Exposed objects are numbered in the order encountered. During the sweep, when an exposed object j is encountered with y-value y_j, $\mathrm{IR}(j)$ will expand as shown in Figure 2(c). $\Delta\mathrm{IR}(j)$ is darkly shaded, and $\mathrm{IR}^{y_j+\epsilon}(i)$ (lightly shaded) becomes $\mathrm{IR}^{y_j-\epsilon}(i) \setminus \Delta\mathrm{IR}(j)$. A set of CN messages is generated to cover $\Delta\mathrm{IR}(j)$ for the exposed object j. The CN messages for all exposed objects together tile $\nabla\mathrm{IR}(i)$ as shown in Figure 2(d) [8]. Rectangles with the same fill pattern are for the same exposed object.

The complete algorithm, *Paint-Dense*, with additional details, is presented in [27]. We conclude with the following:

[6] Note that this subscription must receive $(x_i, y_i^{\mathrm{new}})$ before receiving the replacement object; otherwise, the replacement object would appear to be out of the top-k list because of the stale y-value of i.

[7] $\Delta\mathrm{IR}(j)$ is empty if $\theta_i^{k|y_j-\epsilon}$ is empty, which happens when $\mathrm{IR}^{y_j-\epsilon}(i) = \theta_i$ and $\lambda(\xi, i) < k$ for all $\xi \in \mathrm{IR}^{y_j-\epsilon}(i)$.

[8] The sweep is demonstrated step-by-step in [27].



(a) (b) (c) (d)

Fig. 3. **(a)** Reducing the number of rectangles covering $\mathcal{P}$. **(b)** Finding the next essential exposed object. **(c)** Covering $\mathrm{IR}(i)$ (a staircase in dark shade) using fewer rectangles (shown with thick outlines); $\varepsilon = 1$. **(d)** Covering a change in influence region (a diagonal chain in dark shade) using fewer rectangles (shown with thick outlines); $\varepsilon = \frac{1}{3}$.

Theorem 1. Paint-Dense *generates the minimum number of CN messages for any exact subscription-oblivious algorithm. For a rank-raising update, it generates $O(k)$ messages in time $O(t(n) + \mu + k)$. For a rank-lowering update, it takes $O(\nu t(n) + \mu + k)$ time, where μ is the number of messages generated and $\mu/k - 1 \leq \nu < \mu$ is the number of exposed objects. If the rank-lowering update chooses an object to update uniformly at random, then the expected value of μ is $O(k^2)$; in the worst case, $\mu \leq nk$.*

The bounds on μ in the above theorem are tight in the sense that there is a sequence of updates for which the worst-case bound is $\Omega(nk)$ in the worst case and $\Omega(k^2)$ in the expected case. As our experiments indicate, the value of μ is closer to its expected value on the data sets we tested the algorithm.

B. Subscription-Aware

If we give the server the knowledge about the distribution of subscriptions, the number of CN messages generated by the server can be reduced. In particular, we can avoid sending messages whose corresponding rectangles in $\mathbb{S}$ contain no subscriptions, and can combine multiple messages into one as long as their bounding rectangles contain no extraneous subscriptions and they carry the same object values.

The general problem can be formulated as a geometric optimization problem: *Given a subset of subscriptions $\mathcal{P} \subseteq \mathcal{S}$ (to notify with the same object values), find a set of rectangles $\mathcal{G}$ in $\mathbb{S}$ such that every point of $\mathcal{P}$ lies in exactly one rectangle of $\mathcal{G}$ and no point of $\mathcal{S} \setminus \mathcal{P}$ lies in any rectangle of $\mathcal{G}$. The goal is to minimize the number of rectangles in $\mathcal{G}$.* Figure 3(a) illustrates this problem. Subscriptions in $\mathcal{P}$ are shown as circles while those in $\mathcal{S} \setminus \mathcal{P}$ are shown as dots. Rectangles in $\mathcal{G}^{\mathrm{dense}}$ are shaded, and rectangles in the optimal covering are shown with thick outlines. A brute-force approach is to compute the set $\mathcal{P}$ and then solve the standard rectangular covering problem on $\mathcal{P}$. However, doing so requires us to enumerate potentially large sets of affected subscriptions, which we would like to avoid, and this problem is NP-complete in general [1].

We describe an algorithm *Paint-Sparse* that modifies *Paint-Dense* to optimize CN message generation. We call an exposed object j *essential* if its gain in influence region, $\Delta\mathrm{IR}(j) = \mathrm{IR}^{\mathrm{new}}(j) \setminus \mathrm{IR}^{\mathrm{old}}(j)$, contains some subscription in $\mathcal{S}$; i.e., we must generate some message with j. We iterate through the list of essential exposed objects as follows. Suppose the sweep is currently at position z. Let $\ell^z = \min\{\ell \mid (\ell, r) \in \mathcal{S} \cap \mathrm{IR}^z(i)\}$ and $r^z = \max\{r \mid (\ell, r) \in \mathcal{S} \cap \mathrm{IR}^z(i)\}$; i.e., (ℓ^z, r^z) is the apex of the smallest southeast quadrant $Q^z \subseteq \mathbb{S}$ enclosing all subscriptions in $\mathrm{IR}^z(i)$ (e.g. dashed region in Figure 3(b)). It can be shown that given z, the next essential exposed object j is the point in the 3-sided rectangle $[\ell^z, r^z] \times (z, \infty)$ in $\mathbb{E}$ with the minimum y-coordinate. Using this observation, *Paint-Sparse* finds the next essential exposed object j.

Example. In Figure 3(b), the staircase is the current $\mathrm{IR}^z(i)$. Circles represent subscriptions inside $\mathrm{IR}^z(i)$, and are enclosed by the dashed quadrant with apex at (ℓ^z, r^z). Object h', with the darkly shaded quadrant, is an example of an exposed, but unessential object.

For each essential exposed object j, *Paint-Sparse* first computes $\Delta\mathrm{IR}(j)$ and its partitioning into at most k rectangles as in *Paint-Dense*. Next, we use a simple greedy algorithm that processes the rectangles in order and merges the current rectangle into the last one generated as long as doing so does not cover any subscription not originally covered. This simple approach is within a factor of 2 optimal in the number of messages generated. Omitting the details, which can be found in [27], we obtain the following:

Theorem 2. *Given any set of subscriptions, the number of CN messages generated by* Paint-Sparse *is at most twice the minimum possible for any exact algorithm.* Paint-Sparse *runs in in time $O(t(n) + k \log m)$ for a rank-raising update, and $O(\check{\nu}(t(n) + k \log m))$ time for a rank-lowering update, where $\check{\nu}$ is the number of essential exposed objects; $\check{\nu} < \check{\mu} \leq (\check{\nu} + 1)k$, where $\check{\mu}$ is the number of messages generated by* Paint-Sparse.

Remark. If the entire $\mathcal{S}$ is too expensive to maintain for the server, it can maintain a small sketch of $\mathcal{S}$, e.g., a cover of $\mathcal{S}$ by B rectangles in $\mathbb{S}$ for a parameter B, and use this cover instead of $\mathcal{S}$ itself in *Paint-Sparse*. This approach would provide a continuous trade-off between the cost of maintaining and utilizing information about subscriptions and the number of CN messages generated.

Paint-Sparse's optimization of merging multiple messages, while reducing the number of messages, increases the areas of rectangles in $\mathbb{S}$ corresponding to messages. Larger areas may, for some CN implementations, imply higher dissemination costs. Nonetheless, we note that message merging in *Paint-Sparse* is done in a careful way to avoid false positives, so these larger areas do not reach any more subscriptions and traffic to subscriptions remains minimized. Furthermore, reducing the number of messages is effective in relieving the bottleneck at the server and message injection point.

IV. Extensions

A. Batch Processing

If subscriptions only need to have their top-k lists correctly updated at the end of the batch, we give an algorithm, *Paint-Batch*, which achieves the minimization of the number of messages delivered to the subscriptions within the problem setting of Section II-A without assuming new dissemination interfaces or capabilities. To process individual events in batched sequence $\mathcal{E}$, *Paint-Batch* can use any algorithm $\mathcal{A}$ (either *Paint-Dense* or *Paint-Sparse*), with only minor modifications. The key idea is to pre-process $\mathcal{E}$ in a way such that event-at-time processing by $\mathcal{A}$ (with some modifications) will minimize the number of messages subscriptions receive. Let $\mathrm{IR}^{\mathrm{old}}(i)$ (resp. $\mathrm{IR}^{\mathrm{new}}(i)$) denote the influence region of object i before (resp. after) $\mathcal{E}$. *Paint-Sparse* proceeds in four steps:

1) ***Pre-process.*** First, if multiple events in $\mathcal{E}$ update the same object, we coalesce them into one. Next, we split the set $\mathcal{E}$ into two, $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$, where $\mathcal{E}^{\downarrow}$ (resp. $\mathcal{E}^{\uparrow}$) contains all events that decrease (resp. increase) y-value.
2) ***Apply*** $\mathcal{E}^{\downarrow}$ ***to*** $\mathcal{O}$**.** Let $\mathcal{T}$ denote the data structure we maintain for $\mathcal{O}$. We update $\mathcal{T}$ with using events in $\mathcal{E}^{\downarrow}$. We do not generate any messages in this step.
3) ***Generate messages for*** $\mathcal{E}^{\uparrow}$ ***and apply*** $\mathcal{E}^{\uparrow}$ ***to*** $\mathcal{O}$**.** We process events in $\mathcal{E}^{\downarrow}$ in descending order of the new values using $\mathcal{A}$, but with the following modifications. 1) If $\mathcal{A}$ generates messages for an exposed object that is updated in $\mathcal{E}^{\uparrow}$ or will be later updated in $\mathcal{E}^{\downarrow}$, we discard such messages. 2) If $\mathcal{A}$ is processing a ranking-lowering update for an object i whose messages have been discarded earlier, we notify the region $\mathrm{IR}^{\mathrm{new}}(i) \cup \mathrm{IR}^{\mathrm{pre}}(i)$ with i's updated values, where $\mathrm{IR}^{\mathrm{pre}}(i)$ denotes i's influence region right before we start processing $\mathcal{E}^{\uparrow}$. This can be accomplished using $O(n)$ space.
4) ***Generate messages for*** $\mathcal{E}^{\downarrow}$**.** For each object i updated in $\mathcal{E}^{\downarrow}$, we notify the region $\mathrm{IR}^{\mathrm{new}}(i)$ with i's new value. We simply follow $\mathcal{A}$ to compute the messages by querying $\mathcal{T}$.

Intuition behind the design of *Paint-Batch* is included in [27]. To conclude, we have the following result.

Theorem 3. Paint-Batch *minimizes the number of messages each subscriber receives. Given an event sequence* $\mathcal{E}$*,* Paint-Batch *based on* Paint-Dense *runs in* $O(|\mathcal{E}| \log |\mathcal{E}| + \bar{\nu} t(n) + \bar{\mu})$ *time, and* Paint-Batch *based on* Paint-Sparse *runs in* $O(|\mathcal{E}| \log |\mathcal{E}| + \bar{\nu}(t(n) + k \log m))$ *time, where* $\bar{\mu}$ *is the number of messages generated by* Paint-Batch *and* $\bar{\nu}$ *is the number of objects in these messages.*

B. Approximate Algorithms

To further alleviate the potential message injection bottleneck, more reduction in the number of CN messages generated by the server is possible with approximate algorithms. They allow subscriptions to receive unnecessary messages containing false positive updates to top-k lists, which are discarded by post-processing at the subscriptions. The basic idea is to simplify the boundaries of regions to notify by judiciously including some additional subscriptions. As a simple example, Figure 3(c) shows that if we can use a single message with rectangle $\mathrm{MEB}(\mathrm{IR}(i))$ instead of tiling a staircase-shaped $\mathrm{IR}(i)$ with multiple messages, then any subscriptions receiving i would still rank within top $2k$ for these subscriptions, because we can prove that $\mathrm{MEB}(\mathrm{IR}(i)) \subseteq \theta_i^{\leq 2k}$. This idea can be extended to approximate θ_i^v for any $v \leq k$ with $\lceil 1/\epsilon \rceil$ rectangles so that for any point ξ lying in one of these rectangles $\lambda(\xi, i) \in [(1-\epsilon)k, (1+\epsilon)k]$; see Figure 3(d). We prove that both *Paint-Dense* and *Paint-Sparse* can be made approximate by using this idea, as described in [27].

Theorem 4. *With the approximate algorithms, a subscription receives a message with object* i *only if 1)* i *ranks between* $(1-\varepsilon)k$ *and* $(1+\varepsilon)k$*, or 2)* i *is already in the top* $(1+\varepsilon)k$ *but its value has changed. A rank-raising update generates at most* $1/\varepsilon$ *messages. If a rank-lowering update chooses an object to update uniformly at random, the expected number of messages generated is* $O(k/\varepsilon)$*. The running times of the approximate algorithms are similar to the exact algorithms.*

C. Range Conditions in Higher Dimensions

Our geometric framework is quite general and extends to high dimensions and different types of ranges. Each object i now has d numeric attributes $\{X^{(1)}, \cdots, X^{(d)}\}$ for *selection* by subscriptions, and an additional numeric attribute Y for *ranking*. The event space $\mathbb{E}$ is now $\mathbb{R}^{d+1}$, and each object i is represented as a point $(x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(d)}, y_i) \in \mathbb{E}$. Each subscription S_j specifies a region $\sigma_j \subseteq \mathbb{R}^d$, which can be a d-dimensional box, halfspace, ball, or simplex, or any other shape, and contains the top-k objects among the ones in which it is interested. Each subscription is mapped to a point σ_j^* and each object i to a region θ_i in the subscription space $\mathbb{S}$ so that S_j is interested in object i iff $\sigma_j^* \in \theta_i$. The exact mapping depends on the shape of subscriptions. If each σ is a d-dimensional box $\prod_{h=1}^{d} [\ell_j^{(h)}, r_j^{(h)}]$, then $\mathbb{S} = \mathbb{R}^{2d}$, $\sigma_j^* = (\ell_j^{(1)}, r_j^{(1)}, \cdots, \ell_j^{(d)}, r_j^{(d)})$, and θ_i is the orthant $\{\xi \in \mathbb{R}^{2d} \mid \xi^{(2i-1)} \leq x_i, \xi^{(2i)} \geq x_i, 1 \leq i \leq d\}$. If each σ_j is a halfplane $x^{(d)} \geq a_j^{(1)} x^{(1)} + \cdots + a_j^{(d-1)} x^{(d-1)} + a_j^{(d)}$, then $\mathbb{S} = \mathbb{R}^d$, $\sigma_j^* = (a_j^{(1)}, \ldots, a_j^{(d)})$, and θ_i is a halfspace $\xi^{(d)} \leq -x_i^{(1)} \xi^{(1)} - \cdots - x_i^{(d-1)} \xi^{(d-1)} + x_i^{(d)}$. If $d = 2$ and each σ_j is a disk of radius r_j centered at (a_j, b_j), then $\mathbb{S} = \mathbb{R}^3$, $\sigma_j^* = (a_j, b_j, a_j^2 + b_j^2 - r_j^2)$ and θ_j is the halfspace $\xi^{(3)} \leq 2x_i^{(1)} \xi^{(1)} + 2x_i^{(2)} \xi^{(2)} - x_i^{(1)} - x_i^{(2)}$. It can be verified that, in each case, S_j is interested in i iff $\sigma_j^* \in \theta_i$. The notion of influence region $\mathrm{IR}(i) \subseteq \theta_i$ can be extended to high dimensions. When the y-value of an object i is updated, we update $\mathrm{IR}(i)$ from $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ (if y_i is increased) into constant-size regions, and send one $O(1)$-size message for each such region. Computing the decomposition of $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ or $\mathrm{IR}^{\mathrm{new}}(i)$ becomes more challenging and the number of regions increases, typically exponentially in the worst case, with dimension. However, many of these regions are empty, so *Paint-Sparse* is more effective in high dimensions. In many cases, it is possible to analyze the number of messages generated by the algorithm. The theorem below

gives such a result for the case of rectangles.

Theorem 5. *If the input objects are i.i.d. in $\mathbb{R}^d$ with their attributes being independent and each subscription is an axis-aligned rectangle, then* Paint-Dense *generates $O((k \ln^{d-1} n)^{d+1})$ expected number of CN messages to process an update event.*

V. Evaluation

Network setup. For message dissemination, we use a CN based on *Meghdoot* [14] and the *content addressable network* [24]. This CN uses a network of *brokers* to deliver CN messages of the format described in Section II-A. It partitions the subscription space $\mathbb{S}$ into *zones*, each owned by a broker responsible for all subscriptions within this zone; we call this broker the *gateway* broker of these subscriptions. Each zone can forward messages to its adjacent zones, so messages may travel over multiple hops to their destinations. We use INET [10] to generate a 20,000-node IP network, and randomly pick 1,000 nodes as brokers. Subscriptions are located randomly within the network, and object update events also originate from random locations.

For our approaches, we designate the broker whose zone covers the center of $\mathbb{S}$ as the server, which maintains the database of all objects $\mathcal{O}$. In the case of sparse subscriptions, the server additionally maintains the database of all subscriptions $\mathcal{S}$ (but not how they are assigned to brokers). Events are first routed to the server, where they are reformulated into a sequence of CN messages.

Approaches compared. Our approaches all use CN for message dissemination and only differ in their message generation algorithms. Hence, we use the names of these algorithms to refer to these approaches: exact ones include *Paint-Dense* and *Paint-Sparse*, and approximate ones include *Paint-Dense*(ε) and *Paint-Sparse*(ε) with different ε settings. We compare them with two less sophisticated approaches:

Unicast An event is first sent to the server, which in this case tracks all objects, all subscriptions, and how subscriptions are assigned to gateway brokers. The server computes the set of affected subscriptions. For each affected subscription S_j, the server unicasts to S_j's gateway broker the id j and the change to $\text{top}_k(S_j)$ (which can be captured by one object). This approach is exact in that it notifies only affected subscriptions.

For comparison, we consider the following algorithm for computing unicast messages, which uses some but not all insights from our algorithms.[9] Given $\mathsf{Upd}(x_i, y_i^{\text{old}} \to y_i^{\text{new}})$, the server first computes $\text{II}^{\text{old}}(i) \cup \text{II}^{\text{new}}(i) = (\ell, r)$, and finds all subscriptions in $(\ell, x_i] \times [x_i, r) \subseteq \mathbb{S}$. Next, the server processes each such subscription S_j in turn. For a rank-lowering update, the exposed object has y-value between y_i^{old} and y_i^{new}, and can be found by $\min_y(\sigma_j, y_i^{\text{old}})$.

[9] Alternatively, we may simply use *Paint-Dense* to obtain the list of affected tiles in $\mathbb{S}$, and then look up affected subscriptions within these tiles. In this case, the server processing cost becomes that of *Paint-Dense* plus a term linear in the number affected subscriptions, which is strictly (much) less efficient than *Paint-Dense* and does not offer an interesting comparison.

CN-Relax This approach uses the same CN as our approaches, but does not need a server. An event $\mathsf{Upd}(x_i, y_i^{\text{old}} \to y_i^{\text{new}})$ directly enters the CN as $\mathsf{Msg}(x_i, x_i, -\infty, \infty, x_i, y_i^{\text{new}})$, which reaches all subscriptions whose ranges include x_i. In effect, *CN-Relax* treats each range top-k subscription simply as a range subscription. Each subscription must maintain all objects within its range at all times, from which the top k can be computed. This approach is approximate in that it may notify unaffected subscriptions.

Metrics. We consider the following metrics in evaluation:

Outgoing traffic from the server Measured by the total number of bytes sent by the server. A larger number means higher network stress at the server.

Traffic in the broker network Measured by the total number of bytes sent across network hops, excluding those from gateway brokers to their subscriptions (which are accounted for by the *redundancy* metric discussed below). Depending on what we consider a "hop," there are two metrics: *overlay traffic* treats each overlay link (i.e., a link between two brokers without going through other brokers) as a hop, while *IP traffic* treats each underlying IP link as a hop. IP traffic better reflects physical reality but it depends heavily on the CN implementation; overlay traffic better reflects how we use the CN (as a black box). Well-designed CNs try to make overlay routes as efficient as IP routes, which helps close the gap between these two metrics.

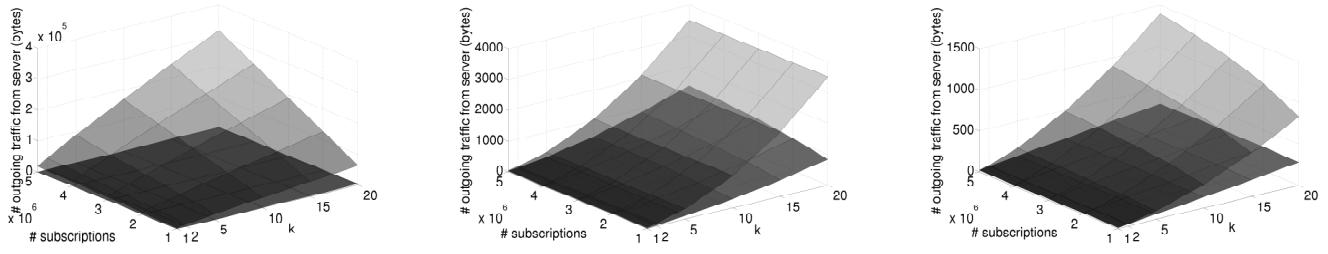
Redundancy in messages received by subscriptions Measured by $\widehat{N}/N - 1$, where $\widehat{N}$ denotes the number of messages received by subscriptions and N denotes the number of messages received by subscriptions under an exact approach. A larger redundancy means higher last-hop traffic and more work for subscriptions. Exact approaches have 0 redundancy.

Server processing cost Measured by the number of calls (by type, as discussed in Section II-B) against the underlying data structures when generating messages. We choose to count the number of calls because the running time depends on the choice of data structures. Our implementation uses data structures that are easier to implement and efficient in practice, but not asymptotically optimal.

Workloads. Most results in this section use synthetic workloads, which allow us to vary their characteristics. Unless specified otherwise, there are 10,000 objects, whose x-values follow one of two distributions: 1) *Uniform*: The x-values are uniformly distributed over the possible x-value range. 2) *Clustered*: The x-values lie in 10 clusters, whose centers partition the possible x-value range into 11 segments of length w. Each cluster gets 10% of the objects. For each object in a cluster, the distance between its x-value and the cluster center follows a Gaussian distribution with standard deviation $w/8$.

To generate an event, we pick an object to update uniformly at random. Its y-value is increased or decreased, each with 0.5 probability. The new y-value is then chosen uniformly random from the possible range of y-values.

Unless specified otherwise, the number of subscriptions is 2 million. We consider the following subscription distributions:



(a) *Unicast* (top) vs. *Paint-Dense* (bottom) (b) *Paint-Dense* (top) vs. *Paint-Sparse* (bottom) (c) *Paint-Sparse* (top) vs. *Paint-Sparse*(1) (bottom)

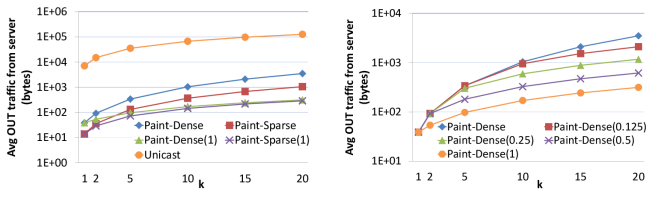
Fig. 4. Average outgoing traffic (# bytes) from server per event.

- *Uniform*: The subscriptions are uniformly distributed in $\mathbb{S}$.
- *Clustered*: Most subscriptions lie in 10 clusters in $\mathbb{S}$. Let P be a set of 10,000 $\times$ 10,000 grid points. We first randomly pick a set C of 10 centers in $\mathbb{S}$ and use a mixture model to assign probability to each point $p \in P$. A parameter λ controls the standard deviation of each cluster $c_i \in \mathcal{C}$. Let σ be $(\max - \min)\lambda/4$, where max and min are the maximum and minimum values in the domain. For each point $p \in P$, $F(p) = \sum_{i=1}^{10} F_i(p)$, where $F_i(p) = \exp(-0.5\|c_i - p\|/\sigma^2)$. The probabilities are then normalized such that they sum to 1.
- *Correlated* (to clustered object distribution): Subscriptions are generated from the 10 clusters of the clustered object distribution. For each subscription in a cluster, the distance between its endpoints from the cluster center follows a Gaussian distribution with standard deviation $w/8$.
- *Anti-correlated* (to clustered object distribution): As with the correlated case above, we generate subscriptions using the clusters of the clustered object distribution. However, we shift each cluster center by $w/2$ and ignore the last cluster, such that each subscription cluster center is located midway between two consecutive object cluster centers for the object distribution.

In addition to synthetic workloads, we have obtained information on 2,031 stocks from Yahoo! Finance. For each stock, we collected data for earnings per stock (EPS), the average recommendation (RECO, which varies from 1, strong buy, to 5, strong sell, over the past month), as well as the open and close prices over 30 days. EPS is then used to convert each price to price-to-earning ratios (PER). Thus, we have a trace of events, each being an update of PER with a RECO constant. 400,000 subscriptions are generated and each requests the k lowest PER over a RECO range.

We first present results for the uniform object distribution and uniform subscription distribution.

Outgoing traffic from server. Figure 4 shows the outgoing traffic from the server per event, averaged over all events in the workload, as we vary k and the number of subscriptions (m). For clarity, we compare only two approaches per plot. Note that *CN-Relax* is not compared because it is a serverless approach. In Figure 4(a), we see that *Paint-Dense*'s outgoing traffic is invariant to m, but *Unicast*'s outgoing traffic is not scalable in m and k. When $m = 5{,}000{,}000$ and $k = 20$, *Unicast* and *Paint-Dense* generate 316,158 and 3,501 bytes, resp. Figure 4(b) shows that by taking into account $\mathbb{S}$, *Paint-Sparse* incurs even lower outgoing traffic than *Paint-Dense*; the gap is wider with fewer (sparser) subscriptions. Figure 4(c) shows that approximation further relieves any potential message injection bottleneck at the server.



(a) Various approaches (b) Varying ε for *Paint-Dense*(ε)

Fig. 5. Average outgoing traffic (# bytes) from server per event.

Figure 5(a) provides more details on the outgoing traffic produced by different approaches. Although outgoing traffic increases for all approaches as k increases, our approaches clearly outperform *Unicast*. *Paint-Dense* generates 1.5 orders of magnitude less outgoing traffic than *Unicast*, whereas *Paint-Sparse*, *Paint-Dense*(1), and *Paint-Sparse*(1) generate between 2 and 2.5 orders of magnitude less. Since the number of messages generated by *Paint-Dense* and *Paint-Dense*(1) is invariant to m, their lead over *Unicast* can widen arbitrarily as subscription density increases. The same trend holds for *Paint-Sparse* and *Paint-Sparse*(1); they always produce no more messages than *Paint-Dense* and *Paint-Dense*(1), resp.

For approximation algorithms, Figure 5(b) shows that increasing ε effectively decreases server outgoing traffic.

Figures above only show average outgoing traffic. When we look at the maximum amount of outgoing traffic from the server per event (which reveals bottlenecks better than the average), in Figure 6(a), we see an even bigger (multiple orders of magnitude) advantage of our approaches over *Unicast*. For *Unicast*, the maximum ongoing traffic is proportional to m, but remains the same when k varies because the number of affected subscriptions does not depend on k in the worst case (e.g., when the most popular object's y-value is dramatically changed). When $m = 5{,}000{,}000$, *Unicast*'s maximum outgoing traffic is 39,998,000 bytes, compared with only 31,752 bytes for *Paint-Sparse* (with $k = 20$).

Traffic in broker network. Figures 7(a) and 7(b) show the amounts of overlay and IP traffic (resp.) incurred per event in

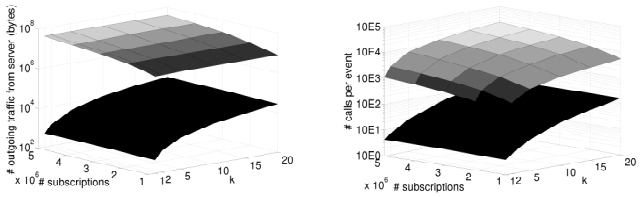
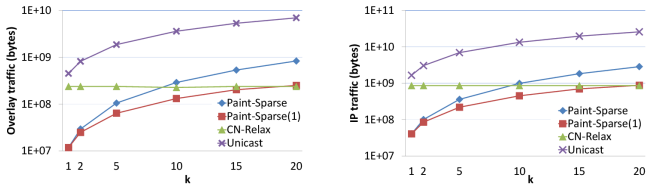


Fig. 6. *Unicast* (top) vs. *Paint-Sparse* (bottom). (a) Maximum outgoing traffic from server for an event. (b) Number of calls per event.



(a) Overlay (b) IP

Fig. 7. Traffic in broker network per event.

the broker network, averaged over all events in the workload. Trends in these two figures are consistent. We see that *Unicast* performs worst among all approaches for all values of k tested and that *Paint-Sparse* leads *Unicast* by an order of magnitude. Furthermore, approximation is effective for reducing in-network traffic, as evidenced by *Paint-Sparse*(1). *CN-Relax* generates the same amount of in-network traffic for all k because it ignores ranking. While it may appear here that *CN-Relax* is attractive when $k > 10$ (largely because *CN-Relax* needs not be concerned with exposed objects), bear in mind that 1) *CN-Relax* requires subscriptions to maintain all objects within their ranges, which is expensive; and 2) *CN-Relax* generates excessive last-hop traffic, as we will see next.

Redundancy in messages received by subscriptions. Table I shows the total number of messages received by subscriptions per event (averaged over the workload) for *Paint-Sparse* (or any exact algorithm). Table II shows the overall redundancy in messages received by subscriptions (averaged over the workload) for the approximate approaches. Note that all exact approaches would have 0 redundancy, and an approximate approach would effectively be exact if $1/\varepsilon \geq k$. Clearly, *CN-Relax* sends a lot of unnecessary messages to subscriptions, negating the advantages in its serverless approach and its relatively lower broker network traffic when $k > 10$. For our approximate approaches, we see that as ε increases, their reduction in traffic from the server and within the broker network comes at the expense of higher redundancy. Still, they offer a spectrum of user-controllable trade-offs that are more attractive than the two extremes: exact algorithms on one hand and *CN-Relax* on the other.

Server processing cost. Figure 6(b) gives a high-level view of the average number of calls per event to the underlying data structures made by *Unicast* and *Paint-Sparse*. Tables III (varying k) and IV (varying m, the number of subscriptions) offer a more detailed breakdown and comparison. As k or m increases, both *Paint-Sparse* and *Unicast* make more calls, but *Unicast* makes orders of magnitude more than *Paint-Sparse*.

k	1	2	5	10	15	20
Paint-Sparse	444.141	939.76	2211.14	4190.21	6082.94	7904.26

Table I. Total number of messages received by subscriptions per event.

Approaches	$k = 1$	2	5	10	15	20
Paint-Sparse$(.125)$	0	0	0	0.015	0.0234	0.034
Paint-Sparse$(.25)$	0	0	0.027	0.061	0.070	0.080
Paint-Sparse$(.5)$	0	0	0.11	0.14	0.16	0.17
Paint-Sparse(1)	0	0.16	0.29	0.30	0.32	0.34
CN-Relax	1440.2	680.14	288.49	151.76	104.23	79.98

Table II. Redundancy in messages received by subscriptions.

k	*Paint-** # first_k	*Paint-Dense* # $\min_y$	*Paint-Sparse* # $\min_y$	*Paint-Sparse* # snap^{10}	*Unicast* # $\min_y$
1	2	1.12	0.72744	1.2284	444.141
2	2	1.73	1.08988	2.95254	1086.44
5	2	3.57	2.59596	12.06538	2846.1
10	2	6.60	5.5568	40.69192	5453.64
15	2	9.63	8.57692	84.65234	8032.39
20	2	12.63	11.58048	143.20416	10587.4

Table III. Average number of calls per event; increasing k.

Paint-Sparse

m ($\times 10^5$)	# $\min_y$	# first_k	# snap^{10}
2	3.52	2	29.87
8	4.84	2	37.34
40	5.94188	2	42.0509
100	6.26988	2	42.86982
Dense	6.60	2	43.36

Unicast; $k = 10$

m ($\times 10^5$)	# $\min_y$
2	545.23
8	2181.49
40	10906.4
100	27267.5

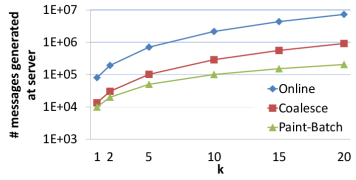
Table IV. Average number of calls per event; increasing m.

Table IV shows [10] that the number of $\min_y$ calls by *Unicast* is linear in m and *Paint-Dense* is invariant to m. For *Paint-Sparse*, when m increases, there are fewer inessential exposed objects, so *Paint-Sparse* needs to examine more exposed objects during a rank-lowering update. However, our experiments show that the number of calls is increased only by a factor of roughly 2 even with dense subscriptions; therefore, our approach is much more scalable.

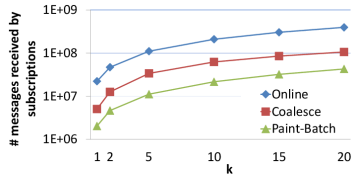
Batch processing. Next, we evaluate the effectiveness of our batch processing algorithm, *Paint-Batch*, by comparing it with *Online*, which simply processes the batched event sequence one event at a time, and *Coalesce*, which coalesces events updating the same object into one before processing, but does not sort or group them into $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$. The sequence contains 50,000 events, and we vary k. Figure 9(a) compares the total number of messages generated over the sequence; Figure 9(b) compares the total number of messages received by all subscriptions; Figure 9(c) compares the total number of $\min_y$ calls. In all figures, *Paint-Batch*, with both coalescing and sorting optimizations, dominates the other approaches. The savings provided by sorting (between *Paint-Batch* and *Coalesce*, especially in the number of messages received by subscriptions) are significant, though they are dwarfed by the savings provided by coalescing.

Trends across synthetic workloads. Results for other workloads are similar, and exhibit trends that confirm intuition. Figure 8(a) shows the ratio between the number of messages generated by *Paint-Sparse* and *Paint-Dense* for various workloads. With knowledge of $\mathcal{S}$, *Paint-Sparse* (as well as

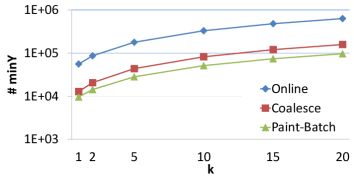
[10] Included here for completeness, $\text{snap}()$ is function used by *Paint-Sparse* to find the smallest rectangle containing all subscriptions inside a given rectangle. See [27] for details.



(a) Total number of messages generated

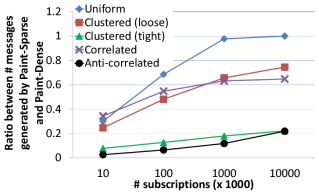


(b) Total number of messages received by subscriptions



(c) Total number of $\min_y$ calls

Fig. 9. Batch processing approaches.



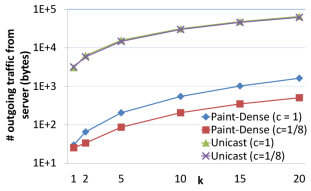
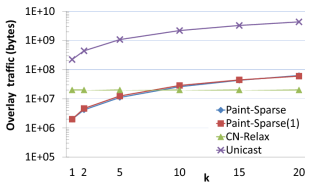
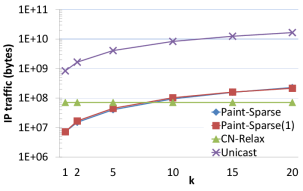


Fig. 8. (a) *Paint-Sparse* vs. *Paint-Dense* for various workloads, with # objects = 1,000. (b) Average outgoing traffic from server per event, with y-value changes following a Gaussian distribution.



(a) Overlay



(b) IP

Fig. 10. Traffic in broker network per event; Yahoo! workload.

Paint-Sparse(ϵ), which is not shown here) generates less traffic with more clustered subscriptions, because of more opportunities for skipping empty regions in $\mathbb{S}$. The ratio is 1 with ten million uniformly distributed subscriptions, which are basically dense. Furthermore, *Paint-Sparse* skips a greater number of inessential exposed objects for the anti-correlated workload than for the correlated one.

In practice, y-values of objects rarely change in a completely random fashion. To see how this observation impacts the performance of our algorithms, instead of choosing new y-values uniformly at random, we let the difference between the new and old y-values follow a Gaussian distribution with standard deviation set to $c/8$ times the length of the range of possible y-values. A smaller c means changes are less volatile. Figure 8(b) shows the traffic from the server for two settings of c. We see that *Paint-Dense* generates fewer messages when c is smaller because fewer objects are exposed by less volatile value (and hence rank) changes. The traffic under Unicast is approximately the same for both $c = 1$ and $c = 1/8$.

Yahoo! Finance data. Results for Yahoo! Finance workload are largely consistent with other results presented in this section, so we sample some here comparing *Paint-Sparse*, *Paint-Sparse*(1), *CN-Relax*, and *Unicast*. In terms of outgoing traffic from the server, this workload allows *Paint-Sparse* and *Paint-Sparse*(1) to inject a significantly fewer number of messages into CN than other workloads, because the y-values (price-to-earning ratios) only change slightly for most events; consequently, most rank-lowering updates expose only a few objects. In terms of traffic in the broker network, Figures 10(a) and 10(b) show that *Paint-Sparse* and *Paint-Sparse*(1) generate two orders of magnitude less traffic than *Unicast*. While *CN-Relax* again seems attractive around $k = 10$, it does poorly with the next metric, redundancy in messages received by subscriptions, shown in Table V. Here, *CN-Relax* results in far more unnecessary traffic to subscriptions with double- and triple-digit redundancy, compared with less than 0.4 for *Paint-Sparse*(1) (and 0 for *Paint-Sparse* because it is exact). Finally, in terms of server processing cost, Table VI shows that *Paint-Sparse* makes few calls. On the other hand, the number of $\min_y$ calls remains huge for *Unicast*, because it still checks all subscriptions in $(\ell, x_i] \times [x_i, r) \subseteq \mathbb{S}$ even though the majority of events affect no subscriptions.

Approaches	$k = 1$	2	5	10	15	20
Paint-Sparse(1)	0	0.19	0.25	0.35	0.39	0.38
CN-Relax	706.51	361.22	147.13	72.67	48.38	36.11

Table V. Redundancy in messages received; Yahoo! workload.

Paint-Sparse

k	# $\min_y$	# first_k	# snap^{10}
1	0.5	2	1
2	0.5	2	2
5	0.51	2	5.05
10	0.54	2	10.35
15	0.57	2	16.02
20	0.61	2	22.18

Unicast

k	# $\min_y$
1	176.01
2	409.44
5	1050.48
10	2278.27
15	3510.16
20	4614.24

Table VI. Average number of calls per event; Yahoo! workload.

VI. Related Work

Much work on scalable processing and notification of subscriptions has been done in the context of publish/subscribe systems (e.g., [6], [22]), but traditionally they consider only selection queries over message attributes. Recent work seek to extend them to support more complex subscriptions (e.g., [8], [11]), or use them for scalable implementation of distributed stream processing [28] and query result caching [13]. The work most relevant to this paper is [8], which discusses scalable processing and dissemination of range top-1 subscriptions. We build on their approach of leveraging CN for efficient dissemination. However, as demonstrated in this paper, the case of $k > 1$ is considerably more complex and requires new algorithms data structures; we also consider batch updates and approximate solutions.

Other recent work on publish/subscribe has also addressed ranking, but with various different subscription semantics; little is known about how best to support standard range top-k subscriptions. Drosou et al. [12] consider ranking events by relevance and diversity. Machanavajjhala et al. [18] consider the reverse problem—finding most relevant subscriptions for a published event. In the sliding window model, Pripuzic et

al. [23] maintains a buffer to store relevant events that have a high probability of entering a top-k result in the future, and Haghani et al. [15] continuously monitor top-k queries over incomplete data streams. Lu et al. [17] consider an approximate top-k real-time publish/subscribe model, in which each subscriber approximately receives the k most relevant publications before a deadline.

Range top-k querying is well studied in the database literature, both in terms of access method design (e.g., [25]), and integration with relational query processing and optimization (e.g., [16]). The key difference is that we focus on a different dimension of scalability here: instead of making a single range top-k query scale over a large dataset, we consider how to scale over a large number of ongoing range top-k queries.

Our problem is related to that of *reverse top-k queries* [26], where, given a data update, affected queries are identified and their results are updated. Their definition of top k is different from ours, however: queries do not specify range conditions but instead vectors of weights that customize relative importance of different ranking criteria. Also, the issue of efficiently notifying affected queries over a network is not considered.

There also has been much research on top-k processing in a distributed setting, e.g., [2], [5], [19], [21]. Most previous work focuses on computing or monitoring the result for a single top-k query over a set of distributed sources, where each source provides either individual object scores or partial scores that must be aggregated across sources before being used for ranking. Our problem setting is inverted—instead of having one query over many distributed objects, we have many distributed subscriptions over one stream of object updates, which call for different techniques.

VII. Conclusion

In this paper we have tackled the problem of supporting a large number of range top-k subscriptions in a wide-area network. Our techniques are based on a geometric framework, enabling us to characterize the subset of subscriptions affected by an event as a region in an appropriately defined space, and solve the problem of notifying affected subscriptions as one of tiling the region with basic shapes. The array of techniques we have developed speak to the power of this framework.

As mentioned in Section I, our techniques can be applied to other application settings. In essence, we have devised an effective way to divide the problem of supporting a large number of stateful subscriptions into two tasks: one that computes a compact description of the changes, and one that further uses this description to update affected subscriptions. This paper uses CN to scale up dissemination for the second task, but there are more possibilities. 1) In settings where we need not deliver result updates over a network, we can scale up the second task of updating subscriptions in parallel, without duplicating the effort of the first task or requiring each processing node to maintain the set of objects. 2) Instead of using a single server to perform the first task, we can distribute the database of objects across multiple nodes, which process incoming events and generate outgoing messages in a distributed fashion. This extension allows us to handle the general publish/subscribe setting where events originate from multiple, distributed publishers.

References

[1] P. K. Agarwal and S. Suri. Surface approximation and geometric partitions. *SIAM J. Comput.*, 27:1016–1035, 1998.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD'03*, pp. 28–39.

[3] C. Buchta. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33:63–65, November 1989.

[4] R. Buyya, M. Pathan, and A. Vakali. *Content Delivery Networks*. Springer, Berlin, Germany, 2008.

[5] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC'04*, pp. 206–215.

[6] A. Carzaniga, D. S. Rosenblum, , and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 2001.

[7] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *IMWS'02*, pp. 59–68.

[8] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD'06*, pp. 587–598.

[9] B. Chandramouli, J. M. Phillips, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems. In *VLDB'07*, pp. 878–889.

[10] H. Chang, R. Govindan, S. Jamin, S. J. Shenker, and W. Willinger. Towards capturing representative as-level internet topologies. In *SIGMETRICS'02*, pp. 280–281.

[11] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB'04*, pp. 612–623.

[12] M. Drosou, K. Stefanidis, and E. Pitoura. Preference-aware publish/subscribe delivery with diversity. In *DEBS'09*, pp. 1–12.

[13] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, Christopher Olston, and Anthony Tomasic. Scalable query result caching for web applications. *Proc. VLDB Endow.*, 1:550–561, August 2008.

[14] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Middleware'04*, pp. 254–273.

[15] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM'09*, pp. 877–886.

[16] C. Li, K. C. Chang, I. F. Ilyas, and S. Song. Ranksql: query algebra and optimization for relational top-k queries. In *SIGMOD'05*, pp. 131–142.

[17] X. Lu, X. Li, T. Yang, Z. Liao, W. Liu, and H. Wang. Rrps: A ranked real-time publish/subscribe using adaptive qos. In *ICCSA'09*, pp. 835–850.

[18] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *Proc. VLDB Endow.*, 1(1):451–462, 2008.

[19] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *TODS*, 29:319–362, 2004.

[20] Y. Nekrich. Space efficient dynamic orthogonal range reporting. *Algorithmica*, 49:94–108, October 2007.

[21] T. Neumann, M. Bender, S. Michel, R. Schenkel, P. Triantafillou, and G. Weikum. Distributed top-k aggregation queries at large. *Distrib. Parallel Databases*, 26:3–27, August 2009.

[22] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *ICDE'05*.

[23] K. Pripužić, I. P. Žarko, and K. Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w. In *DEBS'08*, pp. 127–138.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, August 2001.

[25] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *TKDE*, 16:1555–1570, December 2004.

[26] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. In *ICDE'10*, pp. 365–376.

[27] A. Yu, P. K. Agarwal, and J. Yang. On processing and notifying range top-k subscriptions. Technical report, Duke University, 2011. `http://www.cs.duke.edu/dbgroup/papers/rangetopksub.pdf`.

[28] Y. Zhou, A. Salehi, and K. Aberer. Scalable delivery of stream query result. *Proc. VLDB Endow.*, 2:49–60, August 2009.